DOMAIN-GROUNDED LARGE LANGUAGE MODELS FOR RELIABLE AND EFFICIENT PARALLEL SOFTWARE ENGINEERING: ERROR DETECTION, REPAIR, AND META-LEARNED SKILL OPTIMIZATION

by

SCOTT PIERSALL
M.S. University of North Florida, USA, 2001
B.S. University of North Florida, USA, 1999

A dissertation proposal submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2026

Major Professor: Liqiang Wang

© 2026 Scott Piersall

ABSTRACT

Large language models (LLMs) have demonstrated substantial potential to support parallel software engineering by detecting defects, repairing programs, and generating optimized parallel code. However, general-purpose LLMs often lack the specialized knowledge and execution awareness required to reason reliably about the Message Passing Interface (MPI), where correctness and performance depend on interactions among communication operations, synchronization, data distribution, and process-level behavior. This dissertation investigates the central thesis that grounding LLMs in domain-specific knowledge and empirical program evidence can substantially improve the reliability and efficiency of LLM-assisted parallel software engineering without modifying the underlying model parameters. The research develops a unified approach spanning MPI error detection, automated program repair, parallel code generation, and performance optimization. Domain grounding is provided through structured MPI bug references, representative examples, reasoning guidance, retrieval mechanisms, compiler and static-analysis diagnostics, correctness testing, runtime measurements, and process-level load-imbalance profiles.

The first part of this dissertation examines how bug-referenced prompting, few-shot learning, chain-of-thought reasoning, and retrieval-augmented generation can improve LLM-based detection and repair of MPI defects. The second part introduces METAFORGE, a meta-learning framework that iteratively optimizes persistent natural-language skill specifications using execution-grounded feedback. METAFORGE converts correctness, code-quality, runtime, and utilization measurements into reward-guided textual feedback and maintains an external memory that distills and retrieves optimization knowledge across tasks. These LLM-centered studies are evaluated across diverse MPI programs, workload sizes, experimental conditions, and language models, with comparisons against direct prompting, zero-shot generation, retrieval-based methods, and textual-gradient baselines. They also investigate whether optimization knowledge encoded in learned skills and

external memory transfers to previously unseen parallel-programming tasks. A third study extends the broader principle of domain-grounded learning to autonomous vehicle motion planning. This work integrates explicit vehicle-dynamics and tire–pothole impact models with reinforcement learning to coordinate lateral maneuvering and longitudinal speed control under structural safety, ride-comfort, traffic-safety, and efficiency objectives. Collectively, the three studies investigate how explicit domain knowledge, independently measured system behavior, and persistent learned knowledge can improve the reliability and effectiveness of machine-learning systems operating in technically constrained domains. Building on these results, the proposed future work investigates the trustworthiness of reward-validated persistent optimization knowledge and develops an ontology-grounded validation framework for determining whether learned skill updates should be admitted into persistent agent knowledge.

To my family.

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to the many people who have supported and encouraged me throughout my Ph.D. journey. Pursuing this degree while balancing research, teaching, student mentorship, and professional responsibilities has been both demanding and deeply rewarding. This work would not have been possible without the guidance, patience, encouragement, and sacrifices of many individuals.

First and foremost, I am profoundly grateful to my advisor, Dr. Liqiang Wang, for his exceptional mentorship, unwavering support, and insightful guidance throughout my doctoral studies. His vision, encouragement, and confidence in my work have been instrumental in shaping both this research and my development as a scholar. I am especially thankful for the intellectual freedom he has provided me to explore the intersection of machine learning, software quality, and high-performance computing and to develop these ideas into a coherent research program.

I would also like to sincerely thank my dissertation committee members, Dr. Chen Chen, Dr. Sherif Elfayoumy, and Dr. Wei Zhang, for generously contributing their time, expertise, and thoughtful feedback. Their questions, recommendations, and diverse perspectives have strengthened this research and helped me consider its broader technical and scholarly implications.

My deepest appreciation goes to my family. To my mother and father, thank you for your lifelong love, encouragement, and belief in me. The values of perseverance, responsibility, and hard work that you instilled in me have guided me throughout my personal, professional, and academic life. To my son Benjamin and daughter Stephanie, thank you for your patience, understanding, joy, and unwavering support. You continually remind me that achievement is most meaningful when it is connected to family, purpose, and the opportunity to create a better future for others.

I would also like to thank Dr. Sherif Elfayoumy for encouraging me to return to academia to pursue a PhD after my success in industry.

I am also grateful to the students I have had the privilege to teach and mentor. Their curiosity, determination, and accomplishments have continually renewed my commitment to education and have reminded me why advancing computing research and preparing the next generation of professionals are so important. Many of the ideas underlying this dissertation were strengthened by my experiences teaching systems programming, parallel computing, and other applied computing subjects and by helping students connect academic knowledge with meaningful professional opportunities.

I would also like to acknowledge the colleagues, collaborators, educators, and computing professionals who have encouraged me throughout a career spanning software development, entrepreneurship, and higher education. The lessons I learned while developing production software, building a technology company, working with industry partners, and teaching future computing professionals have shaped how I approach research: practical problems matter, empirical evidence matters, and technical innovation is most valuable when it improves the work and opportunities of others.

Finally, I am grateful to the University of Central Florida and the Department of Computer Science for providing the academic environment, research resources, and opportunities that made this work possible. To everyone who offered guidance, encouragement, friendship, or support along the way, please accept my sincere appreciation. This accomplishment reflects not only my efforts, but also the influence and generosity of the many people who helped me reach this point.

Contents

List of Figures xvii

List of Tables xxi

Chapter 1: INTRODUCTION 1

1.1 Background and Motivation 1

1.2 Domain-Grounded Large Language Models 3

1.3 Limitations of Existing LLM-Based Approaches 4

1.4 Problem Statement 6

1.5 Dissertation Thesis 6

1.6 Research Objectives 7

1.6.1 Improve Domain-Specific Defect Detection 8

1.6.2 Improve LLM-Based Program Repair 8

1.6.3 Improve Parallel Code Generation 8

1.6.4 Develop Persistent Skill Optimization 9

1.6.5 Support Cross-Task Knowledge Transfer 9

1.6.6 Develop Trustworthy Persistent Skill Evolution 9

1.7 Research Questions 10

1.8 Overview of the Dissertation Research 11

1.8.1 Bug-Referenced Defect Detection and Repair 11

1.8.2 Meta-Learned Skill Optimization 12

1.8.3 Held-Out Cross-Task Transfer 13

1.8.4 Future Work: Trustworthy Persistent Skill Evolution 13

1.9 Established and Expected Contributions 14

1.10 Significance of the Research 16

1.11 Scope and Limitations 17

1.12 Reproducibility and Artifact Availability 18

1.13 Pothole-Aware Autonomous Driving and Vehicle Dynamics 19

1.13.1 Navigating Potholes: Learning Safe Motion Planning with Vehicle Dynamics 19

1.13.1.1 Background and Motivation 19

1.13.1.2 Problem Statement 21

1.13.1.3 Research Objectives 22

Chapter 2: LITERATURE REVIEW 24

2.1 Classic Error Detection and Repair in MPI Programs 24

2.2 Modern Software Defect Detection and Repair with Large Language Models . . . 27

2.2.1 LLM-Based Debugging and Fault Localization 28

2.2.2 LLM-Augmented Static Analysis and Defect Detection 29

2.2.3 LLM-Based Software Testing 30

2.2.4 LLM-Based Automatic Program Repair 31

2.2.5 Prompt Engineering and Code-Oriented Language Models 32

2.3 Execution-Grounded Code Optimization 33

2.4 Memory-Augmented Agents 35

2.5 Performance Analysis and Optimization in HPC 36

2.6 Trustworthy Persistent Skill Evolution and Ontology-Grounded Validation 38

2.7 Pothole-Aware Autonomous Driving and Vehicle Control 40

2.7.1 Vehicle Control Technologies for Pothole-Hazard Driving 40

2.7.2 Reinforcement Learning for Autonomous Driving and Vehicle Control . . . 42

2.7.3 Pothole Hazards as Frequent Pre-Crash Events 44

2.8 Synthesis of Research Gaps 45

Chapter 3: Improving MPI error detection and repair with large language models and bug references 48

3.1 Introduction 48

3.2 Methodology 51

3.2.1 Experimental Overview 51

3.2.2 MPI Program Dataset 52

3.2.3 Evaluation Metrics 53

3.2.4 Zero-Shot Baseline 54

3.2.5 Domain-Informed Bug References 54

3.2.6 Few-Shot Bug Referencing 55

3.2.7 MPI-Guided Chain-of-Thought Reasoning 56

3.2.8 Retrieval-Augmented MPI Context 57

3.2.9 Automatic MPI Program Repair 58

3.2.10 Cross-Model Generalization 59

3.3 Evaluation 60

3.3.1 Baseline MPI Defect Detection 60

3.3.2 Effect of Bug References and Structured Reasoning 61

3.3.3 Stability Across Independent Trials 63

3.3.4 Retrieval-Augmented Generation 64

3.3.5 Comparison with Conventional MPI Defect Detectors 66

3.3.6 Automatic Program Repair 67

3.3.7 Generalization Across Large Language Models 68

3.3.8 Cross-Model Repair Behavior 70

3.3.9 Cross-Model RAG Behavior 71

3.3.10 Discussion 72

Chapter 4: A Meta-Learning Framework for Harness Optimization in Automated Code Generation for Parallel Computing 75

4.1 Introduction 75

4.2 Methodology 77

4.2.1 Framework Overview 77

4.2.2 Notation and Framework State 79

4.2.3 Execution-Grounded Evaluation 79

4.2.4 Meta-Learning Control Flow 81

4.2.5 Reward-Guided Textual-Gradient Refinement 82

4.2.6 Retrieval-Augmented Optimization Memory 83

4.2.7 Cross-Task Knowledge Persistence 86

4.2.8 Prompt Specifications 86
4.2.8.1 Code-Generation Prompt 87
4.2.8.2 Textual-Gradient Prompt 87
4.2.8.3 Skill-Rewrite Prompt 88
4.2.8.4 Knowledge-Distillation Prompt 89
4.3 Evaluation 89
4.3.1 Experimental Design 89
4.3.1.1 Compilation and Execution Environment 90
4.3.1.2 Baselines and Experimental Controls 91
4.3.1.3 Metrics 92
4.3.2 Overall Results 92
4.3.3 Task-Level and Trial-Level Results 94
4.3.3.1 Calculate π 94
4.3.3.2 Matrix Multiplication 96
4.3.3.3 Mandelbrot Sets 98
4.3.3.4 Heat Transfer Using Partial Differential Equations 101
4.3.3.5 Distributed Principal Component Analysis 103

4.3.4 Run-Level Token Consumption 105

4.3.5 Retrieval-Memory Ablation 106

4.3.6 Held-Out Cross-Task Transfer 107

4.3.7 Zero-Update Skill Transfer 111

4.3.8 Cross-Model Ablation: Claude and Codex 112

4.3.9 Limitations and Interpretation 113

Chapter 5: Navigating Potholes: Learning Safe Motion Planning with Vehicle Dynamics . 116

5.1 Introduction 116

5.2 Methodology 118

5.2.1 Framework Overview 118

5.2.2 Impact Mechanics Modeling 119

5.2.2.1 Quarter-Car Suspension Dynamics 119

5.2.2.2 Wheel–Pothole Impact Dynamics 121

5.2.3 Hierarchical Tactical Planning 123

5.2.4 Multi-Objective Reward Design 125

5.2.4.1 Passage Efficiency 126

5.2.4.2 Traffic Safety 126

5.2.4.3 Ride Comfort 127
5.2.4.4 Structural Durability 127
5.2.4.5 Reward Calibration 127
5.2.5 Curriculum Learning 128
5.2.6 Integrated Multi-Physics Simulation Environment 129
5.2.7 Empirical Pothole Parameterization 130
5.3 Evaluation 131
5.3.1 Evaluation Design 131
5.3.2 Learning Dynamics and Convergence 133
5.3.3 Emergent Pothole-Handling Strategies 134
5.3.4 Overall Comparative Performance 135
5.3.5 Avoidable Pothole Scenarios 136
5.3.6 Unavoidable Pothole Scenarios 137
5.3.7 Representative Case Studies 139
5.3.7.1 Case I: Lateral Avoidance Under Rear-End Pressure 139
5.3.7.2 Case II: Impact-Constrained Traversal of an Unavoidable Pothole 139
5.3.8 Generalization to Sequential Hazards 141

5.3.9 Evaluation Summary and Limitations 142

Chapter 6: Future Work 144

6.1 Future Work: Trustworthy Skill Evolution through Ontology-Grounded Validation . 144

6.1.1 Research Gap and Proposed Direction 145

6.1.2 Research Questions 147

6.1.3 Ontology-Grounded Skill Validation Framework 148

6.1.4 Semantic and Constraint-Based Skill Validation 149

6.1.5 Proposed Research Methodology 151

6.1.5.1 Phase 1: Characterize Endogenous Reward Skill Hacking 151

6.1.5.2 Phase 2: Develop the Trustworthiness Ontology and Knowledge Graph 152

6.1.5.3 Phase 3: Develop Semantic Skill Extraction and Formal Constraints 152

6.1.5.4 Phase 4: Integrate Independent Executable Verification 153

6.1.5.5 Phase 5: Evaluate Persistence, Transfer, and Generalization . . . 154

6.1.5.6 Phase 6: Evaluate Trustworthiness and Optimization Utility . . . 155

6.1.6 Scope and Anticipated Limitations 156

6.1.7 Expected Contributions 157

List of Figures

3.1 Comparison of Zero-Shot, Few-Shot, Few-Shot+CoT, and retrieval-augmented prompting for ChatGPT. The strongest detection performance is obtained when explicit MPI bug references are combined with structured MPI reasoning. 62

3.2 ChatGPT classification performance across retrieval corpora containing different proportions of correct and defective MPI programs. 64

3.3 ChatGPT repair outcomes for successful compilation, resource-leak removal, and deadlock removal using the domain-grounded Few-Shot+CoT repair strategy. 68

3.4 Confusion-matrix outcomes across prompting strategies for Llama 2, Code Llama, and Qwen2.5-Coder. Few-Shot bug references combined with MPI-specific CoT reasoning produce the largest increase in true positives and the strongest reduction in false negatives across all three models. 69

3.5 Accuracy, precision, recall, and F1 across prompting strategies for the three additional LLMs. 70

3.6 Repair success and failure rates across Llama 2, Code Llama, and Qwen2.5-Coder. 71

3.7 Accuracy, Precision, Recall, and F1 Score by Retrieval Corpus Composition using Llama 2, Code Llama, and Qwen2.5-Coder, showing repeated trends for each retrieval corpus by LLM. 72

4.1 MetaFORGE execution-grounded skill-refinement architecture. 78

4.2 MetaFORGE code-generation prompt. 87

4.3 MetaFORGE textual-gradient prompt. 88

4.4 MetaFORGE skill-rewrite prompt. 88

4.5 MetaFORGE knowledge-distillation prompt. 89

4.6 Average optimization improvement versus per-iteration token consumption. . 94

4.7 MPI π calculation results comparing MetaFORGE and TextGrad [114]. MetaFORGE achieves higher code-quality, runtime, and load-balance improvements with lower token consumption across small, medium, and large workloads over 15 trials. (5 trials per size) 95

4.8 MPI matrix multiplication results comparing MetaFORGE and TextGrad across 15 trials. MetaFORGE achieves higher code-quality, runtime, and load-balance improvements. 97

4.9 Mandelbrot-set generation results comparing METAFORGE and TEXTGRAD across 15 trials. METAFORGE achieves stronger static-analysis warning reduction, runtime improvement, and load-balance improvement, with modestly higher aggregate per-iteration token consumption. 100

4.10 Heat-transfer PDE results comparing MetaFORGE and TextGrad. MetaFORGE achieves stronger code-quality, runtime, and load-balance improvements, with modestly higher token consumption over 15 trials. 102

4.11 Distributed PCA results comparing MetaFORGE and TextGrad. MetaFORGE achieves higher code-quality, runtime, and load-balance improvements with lower per-iteration token consumption over 15 trials. 105

4.12 Comparison of MetaFORGE with and without retrieval context memory on the π calculation task. Each point represents one optimization trial across all iterations, while diamonds indicate the mean value. The three subfigures report (left) static analysis improvement, (center) runtime improvement, and (right) load imbalance improvement. Incorporating retrieval context consistently improves runtime optimization while maintaining comparable static analysis and load balancing performance. 109

4.13 **Held-out cross-task transfer protocol.** For each leave-one-task-out fold, METAFORGE is trained sequentially on the two non-held-out tasks across all workload sizes. The resulting transferred skill and cross-task memory are frozen before evaluation. Zero-update transfer isolates knowledge retained in the transferred skill, whereas adaptation transfer evaluates refinement with and without read-only access to the frozen cross-task memory. The held-out task is excluded from all Phase 1 skill updates and memory construction. . . . 115

5.1 Overall framework of the proposed vehicle-dynamics-oriented hierarchical autonomy for pothole hazards. 119

5.2 Proposed mechanism-informed model for tire–pothole edge impact analysis. . 122

5.3 Architecture of the hierarchical decision-making and low-level execution layers. 124

5.4 Representative pothole samples from the two data sources used for geometry parameterization. 131

5.5 Training curves of episodic return. Solid lines denote the mean over five random seeds, and shaded regions visualize across-seed variability. 134

5.6 Visualization of emergent in-lane pothole avoidance behaviors learned by the proposed method. (a) Left nudge. (b) Right nudge. (c) Straddling (center pass). 135

5.7 Distributional comparison of pothole-handling metrics. 138

5.8 Case I: avoidable pothole scenario. 140

5.9 Case II: unavoidable pothole scenario. 141

5.10 Sequential-hazard generalization. 142

List of Tables

3.1 Confusion Matrix of Core Definitions in Software Defect Detection 53

3.2 Nomenclature and description of retrieval databases used in RAG studies. . . 58

3.3 Classification performance across ChatGPT prompting strategies. RAG conditions vary the composition of the retrieval corpus. Bold indicates the best value for each metric across the evaluated configurations. 61

3.4 Variance (σ^2) of confusion-matrix counts across five ChatGPT trials. Lower variance indicates greater stability; bold marks the lowest variance for each count. 63

3.5 Mean ChromaDB embedding-space retrieval distance for each retrieval-corpus composition. Lower values indicate closer proximity under the embedding representation. 65

3.6 MPI defect-detection performance of conventional analysis tools on the evaluation dataset. 66

3.7 McNemar tests comparing Zero-Shot detection with the Few-Shot+CoT bug-referencing approach across the four evaluated LLMs. 70

4.1 MetaFORGE notation. 79

4.2 Context-retrieval notation. 80

4.3 Context-retrieval scoring weights. Structural similarity receives the greatest weight, while reward improvement and recency provide secondary ranking signals. 86

4.4 Aggregate METAFORGE and TEXTGRAD results across five trials per workload. Static-analysis, runtime, and load-imbalance improvements are computed relative to zero-shot Codex; higher is better. Workload-level percentages are computed from pooled raw measurements and may therefore differ from the arithmetic means of the trial-level percentages reported in the detailed tables. Tokens denote mean per-iteration consumption; lower is better. Task-average rows summarize the three workload-level aggregate results. Avg. best iter. reports the mean iteration at which the best reward was obtained and is descriptive rather than a quality metric. Bold marks the better result. 93

4.5 Per-trial π-calculation results for the small workload. 96

4.6 Per-trial π-calculation results for the medium workload. 96

4.7 Per-trial π-calculation results for the large workload. 97

4.8 Per-trial matrix-multiplication results for the small workload. 98

4.9 Per-trial matrix-multiplication results for the medium workload. 99

4.10 Per-trial matrix-multiplication results for the large workload. 99

4.11 Per-trial Mandelbrot-set results for the small workload. 101

4.12 Per-trial Mandelbrot-set results for the medium workload. 101

4.13 Per-trial Mandelbrot-set results for the large workload. 102

4.14 Per-trial heat-transfer PDE results for the small workload. 103

4.15 Per-trial heat-transfer PDE results for the medium workload. 104

4.16 Per-trial heat-transfer PDE results for the large workload. 104

4.17 Per-trial distributed PCA results for the small workload. 106

4.18 Per-trial distributed PCA results for the medium workload. 106

4.19 Per-trial distributed PCA results for the large workload. 107

4.20 Run-level optimization cost across five independent trials per workload. Values are mean ± sample standard deviation. Prompt and processing tokens sum to total tokens. Lower token counts indicate lower measured token consumption. 108

4.21 Retrieval-memory ablation for Calculate π. 109

4.22 Held-out cross-task adaptation with and without frozen-memory retrieval. . . 111

4.23 Zero-update results averaged across trials. All generated programs passed task-specific correctness testing. 112

4.24 Cross-model ablation on distributed PCA over five trials per workload. Values are mean improvements and per-iteration token consumption. **Bold** indicates the better result. 113

5.1 Quarter-car model parameters. 121

5.2 Parameters used in the wheel–pothole impact model. 123

5.3 Configuration of reward function weights. 128

5.4 Follower IDM and reaction-time parameters. 130

5.5 Main hyperparameters used for SAC training. 132

5.6 Performance Statistics Across All 240 Evaluation Episodes 136

5.7 Performance Statistics in 120 Avoidable Pothole Scenarios 137

5.8 Performance Statistics in 120 Unavoidable Pothole Scenarios 137

Chapter 1: INTRODUCTION

1.1 Background and Motivation

Large language models (LLMs) have demonstrated substantial potential for assisting with software engineering tasks, including code generation, program explanation, defect detection, debugging, repair, testing, and optimization [21]. By learning statistical regularities from large collections of natural language and source code, these models can produce plausible implementations from high-level descriptions and offer recommendations that resemble those of human developers. As LLMs become increasingly integrated into software development pipelines, their role is expanding toward more autonomous forms of program analysis and synthesis.

Despite these advances, the reliability of LLM-generated or LLM-modified software remains a significant concern. A generated program may compile and appear reasonable while still containing subtle semantic defects, violating application programming interface requirements, producing incorrect results, or exhibiting unacceptable performance. These limitations are particularly consequential in high-performance and parallel computing, where program behavior depends not only on local source-code statements but also on interactions among concurrent processes, communication operations, synchronization points, data distributions, and hardware resources.

Parallel software engineering presents challenges that differ substantially from those encountered in conventional sequential programming. Developers must reason about multiple executing processes, distributed data, message ordering, synchronization, collective communication, non-blocking operations, resource lifecycles, and process-level performance. Defects such as mismatched communication operations, inconsistent collective calls, invalid communicator usage, incomplete non-blocking requests, unsafe synchronization, and incorrect data decomposition may

result in deadlock, nondeterministic behavior, silent corruption, or failures that appear only under particular process counts or workload configurations. Even when a parallel program is functionally correct, inefficient communication, unnecessary synchronization, poor memory access, or imbalanced work distribution can significantly reduce performance and resource utilization.

The Message Passing Interface (MPI) provides a widely adopted programming model for distributed-memory parallel computing [57, 25, 70]. MPI offers flexible primitives for point-to-point communication, collective operations, derived data types, process synchronization, remote-memory access, and parallel input/output. This flexibility enables the development of scalable scientific and engineering applications but also creates a large space of possible correctness defects and performance problems. MPI therefore provides a rigorous evaluation domain for studying whether LLMs can reason about software systems whose behavior emerges from interactions across multiple processes.

General-purpose LLMs do not guarantee conformance to MPI semantic rules or optimization for a particular execution environment. Although a model may recognize common MPI functions and generate syntactically plausible code, direct prompting can still miss MPI-specific defects, motivating the use of explicit domain knowledge and external validation [73].

This dissertation is motivated by the premise that LLMs for parallel software engineering should not operate solely from general pretrained knowledge. Instead, they should be grounded in domain-specific knowledge and empirical evidence obtained from the programs they analyze or generate. Domain grounding can provide explicit descriptions of known defect patterns, representative examples, API constraints, specialized reasoning guidance, and retrieved cases. Execution grounding can provide compiler diagnostics, static-analysis findings, correctness-test results, runtime measurements, and process-level performance profiles. Together, these forms of grounding can transform an LLM from an unconstrained code generator into a component of a broader evidence-driven

software-engineering process.

1.2 Domain-Grounded Large Language Models

In this dissertation, *domain grounding* refers to the deliberate augmentation of an LLM with specialized knowledge that is directly relevant to the target software-engineering problem. For MPI programming, this knowledge may include descriptions of MPI correctness rules, known bug patterns, annotated examples, repair strategies, communication semantics, static-analysis findings, and optimization principles. The grounding information may be supplied directly within a prompt, selected through retrieval, encoded in a persistent skill specification, or accumulated in an external memory across multiple problem-solving episodes.

Domain grounding differs from relying exclusively on the latent knowledge contained in a pretrained model. The model's internal representations may contain partial knowledge of MPI and parallel programming, but that knowledge is difficult to inspect, may be incomplete, and may not be activated consistently by a task prompt. Supplying explicit domain information makes the reasoning context more transparent and allows the software-engineering process to be shaped by explicit domain-specific technical knowledge.

This dissertation also considers *execution grounding*, in which LLM reasoning is informed by evidence produced through compilation, analysis, testing, and execution. Execution grounding is particularly important for parallel programming because source-level plausibility does not guarantee correct or efficient runtime behavior. A program must be compiled, tested, executed under representative conditions, and evaluated using measurements that reflect both correctness and performance.

Domain and execution grounding are complementary. Domain knowledge can guide an LLM

toward likely defects and appropriate implementation strategies, whereas execution evidence can determine whether those strategies succeed in practice. A retrieved description of an MPI bug may help a model detect a defect, but compiler diagnostics and correctness tests provide additional evidence about the generated repair. Similarly, a natural-language skill may instruct a model to distribute work evenly, but process-level runtime measurements are needed to establish whether load imbalance was actually reduced.

1.3 Limitations of Existing LLM-Based Approaches

Existing research has shown that domain-informed examples and structured reasoning, retrieval-augmented code generation, and iterative textual feedback can improve performance in software-engineering and code-generation settings [73, 117, 114].

First, many code-generation evaluations emphasize compilation or functional correctness while giving limited attention to domain-specific defects. A program that passes general tests may still violate MPI usage rules, leak communication resources, rely on unsafe ordering assumptions, or fail under a different number of processes. Domain-specific analysis is therefore necessary to evaluate properties that are not captured by conventional syntax or unit tests.

Second, many approaches treat defect detection, program repair, and code generation as separate tasks. In practice, these activities are closely related. Effective repair requires accurate defect detection, and effective generation requires the model to avoid the same defect patterns that appear in defective programs. Knowledge learned from detection and repair may therefore be valuable for subsequent code generation and optimization.

Third, one-time prompting provides no persistent mechanism for improving the instructions that guide an LLM. A model may receive feedback about a particular program, but the resulting insight

is often discarded after the interaction. Without a persistent representation of successful strategies, the system must rediscover similar lessons across tasks and runs.

Fourth, existing textual-feedback methods may produce lengthy or weakly structured optimization instructions. Feedback that merely restates compiler output or performance measurements may not communicate the relationship between an observed problem and the changes needed to resolve it. An effective refinement framework must translate heterogeneous evidence into concise and actionable language that can improve future generations.

Fifth, performance optimization introduces objectives that may conflict with one another. A change that reduces runtime may decrease readability, introduce a correctness risk, or worsen load balance. Similarly, eliminating a static-analysis warning may produce additional communication overhead. A parallel code-generation system must therefore evaluate multiple dimensions of program quality rather than optimizing a single metric in isolation.

Finally, comparatively little work has examined whether language-encoded optimization knowledge learned from execution feedback can persist and transfer across distinct parallel-programming tasks. Existing work demonstrates execution-guided optimization and agentic code generation for parallel programs, but these systems do not establish persistent cross-task natural-language skill transfer for MPI code generation [100, 40, 93]. A skill refined for numerical integration, for example, may encode general principles concerning data decomposition, collective reduction, synchronization, and load distribution. Whether such principles improve generation for previously unseen parallel-programming tasks therefore remains an important research question.

1.4 Problem Statement

LLMs can generate and analyze parallel programs, but their outputs are limited by incomplete domain knowledge, insufficient awareness of execution behavior, and the absence of persistent mechanisms for retaining reward-validated optimization knowledge. Direct prompting may produce plausible responses without ensuring that MPI-specific defects are detected, repairs preserve program behavior, or generated programs are both correct and efficient.

The central problem addressed by this dissertation is therefore:

> *How can large language models be systematically grounded in domain knowledge and execution evidence to improve the detection, repair, generation, and optimization of parallel software?*

Addressing this problem requires methods that integrate specialized knowledge with empirical software-engineering tools. It also requires an approach that can retain successful strategies across iterations and determine whether those strategies generalize to previously unseen tasks.

1.5 Dissertation Thesis

The central thesis of this dissertation is that:

> *Grounding large language models in explicit domain knowledge, program-analysis evidence, and runtime feedback can substantially improve their reliability and efficiency for parallel software engineering, while persistent natural-language skills and external memory can preserve and transfer reward-validated optimization knowledge across tasks without modifying the underlying model parameters.*

This thesis is investigated primarily through two complementary LLM-centered research directions, with a third study providing a broader demonstration of domain-grounded machine learning in a safety-critical physical system. The first direction examines how explicit MPI bug knowledge, representative examples, structured reasoning, and retrieval can improve LLM-based error detection and program repair. The second investigates how compiler feedback, static analysis, correctness testing, runtime measurements, and load-imbalance profiles can guide the iterative refinement of persistent natural-language skills for parallel code generation. The third study investigates vehicle-dynamics-oriented reinforcement learning for autonomous pothole handling, demonstrating how explicit physical models, safety constraints, and execution outcomes can similarly ground learned decision making.

Together, these studies examine a common principle across two technically constrained settings: explicit domain knowledge and independently measured evidence can ground learned systems in the semantics and requirements of the domain in which they operate. Within parallel software engineering, the research progresses from MPI defect detection and repair to execution-grounded code generation, persistent skill optimization, and cross-task knowledge transfer. The autonomous-driving study provides a complementary evaluation of the same principle in a safety-critical physical system, where learning is grounded in vehicle dynamics, impact mechanics, and traffic-safety constraints. Building on the persistent-skill results, the proposed future work then asks whether reward-validated optimization knowledge should be independently evaluated for trustworthiness before it is permitted to become persistent.

1.6 Research Objectives

The primary objective of this dissertation is to develop and evaluate domain-grounded methods that improve the reliability, effectiveness, and trustworthiness of LLM-based parallel software en-

gineering. The completed studies address defect detection, program repair, parallel code generation, persistent skill optimization, and cross-task knowledge transfer, while the proposed future work extends these objectives to trustworthy persistent skill evolution.

1.6.1 Improve Domain-Specific Defect Detection

The first goal is to determine how explicit MPI knowledge affects an LLM's ability to identify defects in parallel programs. This work investigates the use of bug references, few-shot examples, structured reasoning guidance, and retrieval-augmented generation. The objective is not merely to determine whether a model labels a program as defective, but whether it identifies the relevant defect, explains the underlying MPI rule, and distinguishes true defects from valid but uncommon programming patterns.

1.6.2 Improve LLM-Based Program Repair

The second goal is to evaluate whether domain grounding improves the correctness and precision of LLM-generated repairs. A useful repair must resolve the identified defect while preserving intended program behavior and avoiding the introduction of new defects. This goal therefore requires evaluation procedures that combine known defect labels, compilation, testing, and domain-specific analysis.

1.6.3 Improve Parallel Code Generation

The third goal is to develop methods for generating parallel programs that satisfy both correctness and performance requirements. The generated programs should compile, pass task-specific correctness tests, minimize domain-specific static-analysis findings, execute efficiently, and distribute

work effectively among processes.

1.6.4 Develop Persistent Skill Optimization

The fourth goal is to investigate whether natural-language skill specifications can be iteratively improved using execution-grounded feedback. Rather than fine-tuning model parameters, the approach modifies a persistent textual artifact that instructs the LLM how to approach future generation tasks. This allows the optimization process to remain model-agnostic, inspectable, and comparatively lightweight.

1.6.5 Support Cross-Task Knowledge Transfer

The fifth goal is to determine whether optimization knowledge learned from one collection of MPI tasks can improve performance on previously unseen tasks. This goal examines both direct transfer, in which a learned skill is applied without further updates, and adaptation transfer, in which the transferred skill is refined using new task evidence. The role of external memory in retaining and retrieving cross-task knowledge is also evaluated.

1.6.6 Develop Trustworthy Persistent Skill Evolution

The sixth, proposed goal is to determine how reward-validated optimization knowledge can be independently evaluated for semantic and evaluation-integrity trustworthiness before it becomes persistent. This work will characterize the emergence, persistence, and transfer of endogenous reward- or evaluation-hacking strategies and will develop an ontology-backed knowledge graph for representing skill updates, provenance, trustworthiness constraints, and independent verification evidence. The objective is to evaluate whether untrustworthy skill updates can be identified and

prevented from entering persistent agent knowledge while preserving legitimate improvements in program correctness, performance, code quality, and resource utilization.

1.7 Research Questions

This dissertation is guided by the following research questions:

1. **RQ1:** To what extent do domain-specific bug references, few-shot examples, and structured reasoning improve LLM-based detection of MPI programming defects?
2. **RQ2:** To what extent does domain grounding improve the correctness and precision of LLM-generated repairs for defective MPI programs?
3. **RQ3:** How effectively can retrieval-augmented generation select and apply relevant MPI knowledge during defect detection and repair?
4. **RQ4:** Can compiler diagnostics, static-analysis findings, correctness tests, runtime measurements, and load-imbalance profiles be combined to guide iterative improvement of natural-language skills for parallel code generation?
5. **RQ5:** Does execution-grounded skill optimization improve generated MPI programs across correctness, code quality, runtime, and process utilization compared with direct generation and textual-gradient baselines?
6. **RQ6:** Can language-encoded optimization knowledge and external memory transfer improvements to MPI tasks that were excluded from the skill-learning process?
7. **RQ7:** What limitations, failure modes, and tradeoffs are observed in the completed domain-grounded LLM studies for MPI defect detection, repair, and execution-grounded skill optimization?

1.8 Overview of the Dissertation Research

1.8.1 Bug-Referenced Defect Detection and Repair

The first component of this dissertation investigates domain-grounded LLM methods for MPI defect detection and repair. The approach uses a curated corpus of MPI programs containing known correct and incorrect behaviors. The programs represent a range of MPI defect classes, including point-to-point communication defects, collective-operation mismatches, incorrect parameters, communicator misuse, invalid resource management, and unsafe synchronization patterns.

Several LLM prompting and grounding strategies are compared. Direct prompting serves as a baseline in which the model receives a program and a general instruction to identify or repair errors. Few-shot prompting augments the request with representative labeled examples. Structured reasoning encourages the model to analyze relevant communication relationships, process roles, and MPI semantics before producing a conclusion. Bug-referenced prompting supplies explicit descriptions of known MPI error patterns. Retrieval-augmented generation selects bug references or examples that are semantically related to the target program.

The evaluation considers more than raw response accuracy. Detection is assessed using true-positive, false-positive, and false-negative behavior across known defect categories. Repair is evaluated through compilation, testing, and verification that the original error has been removed without introducing additional defects. This component establishes whether explicit domain knowledge can compensate for weaknesses in general-purpose LLM reasoning.

1.8.2 Meta-Learned Skill Optimization

The second component introduces METAFORGE, a meta-learning framework for optimizing persistent LLM skill specifications. A skill specification is a natural-language document that provides reusable instructions, constraints, strategies, and domain knowledge for parallel code generation. The skill is distinct from the task specification: the task defines what program should be generated, whereas the skill describes how the model should reason about and implement parallel programs.

For each iteration, METAFORGE uses the current skill and a fixed task specification to generate a candidate MPI program. The candidate is then compiled, analyzed, tested, and executed. The resulting evidence includes compiler diagnostics, domain-specific static-analysis findings, correctness outcomes, median runtime measurements, and process-level load-imbalance information.

These signals are combined into a composite reward representing the quality of the generated candidate. A textual-gradient process converts the observed evidence and reward trend into recommendations for refining the skill. Rather than modifying the candidate program directly, the framework updates the reusable skill that will guide subsequent generations. This distinction allows the framework to improve the model's persistent problem-solving instructions across iterations.

METAFORGE also maintains a lightweight external memory. The memory stores compressed optimization knowledge supported by prior evidence, such as successful decomposition strategies, communication patterns, performance observations, and warnings against ineffective approaches. Relevant memory entries can be retrieved during later iterations or tasks, allowing the system to reuse previously reward-validated optimization knowledge.

The framework was evaluated on five MPI programming tasks representing different computation and communication characteristics: numerical integration for calculating π, distributed matrix multiplication, Mandelbrot Sets generation, heat-transfer simulation using partial differential equa-

tions, and distributed principal component analysis. Each task was evaluated at small, medium, and large workload sizes to determine whether the learned skill remained effective as computational and communication demands changed.

1.8.3 Held-Out Cross-Task Transfer

A central component of the METAFORGE study evaluated whether learned skills and external memory generalized beyond the tasks used during optimization. A leave-one-task-out protocol excluded each target task from the skill-learning process while the remaining tasks were used to refine a transferred skill and populate the external memory.

The held-out task was then evaluated under complementary transfer conditions. In zero-update transfer, the transferred skill was used without task-specific rewriting, retrieval, feedback, or subsequent updating, isolating the optimization knowledge encoded directly in the skill. In adaptation transfer, the transferred skill was further refined on the held-out task, both with retrieval disabled and with read-only access to the frozen cross-task memory. This comparison isolated the contribution of previously learned external memory to held-out adaptation.

Together, these experiments evaluated whether natural-language optimization knowledge captured reusable principles of parallel programming rather than merely overfitting to the tasks on which the skill and memory were learned.

1.8.4 Future Work: Trustworthy Persistent Skill Evolution

The limitations and failure modes characterized under RQ7 motivate a distinct prospective trustworthiness question: whether optimization knowledge should be trusted solely because it improves a measured reward. A reward-guided optimizer may discover strategies that improve an observ-

able metric while weakening the semantic correctness or evaluation integrity of the intended task. When such a strategy is encoded into a reusable skill, the resulting failure can persist beyond the generation in which it originated.

Section 6.1 therefore proposes future research on trustworthy skill evolution through ontology-grounded validation. The proposed framework will represent learned skill updates, optimization objectives, evaluators, evidence, provenance, and trustworthiness constraints within an ontology-backed knowledge graph and will evaluate proposed skill modifications against explicit semantic, provenance, evaluation-integrity, and evidence requirements before they are admitted into persistent knowledge. Independent executable evidence will provide an additional verification layer for properties that can be measured externally. This work extends the dissertation from determining whether persistent natural-language skills can improve future generations to determining whether reward-validated optimization knowledge satisfies independently defined trustworthiness requirements before it becomes persistent.

1.9 Established and Expected Contributions

The completed studies establish several dissertation contributions, while the proposed future work extends these contributions toward trustworthy persistent skill evolution:

1. **A domain-grounded methodology for MPI defect detection.** The dissertation develops and evaluates prompting and retrieval strategies that expose LLMs to explicit MPI bug knowledge, representative examples, and structured reasoning guidance.

2. **An empirical evaluation of LLM-based MPI program repair.** The research assesses whether detected errors can be repaired accurately while preserving intended behavior and avoiding newly introduced defects.

3. **An execution-grounded framework for skill refinement.** METAFORGE integrates compilation, static analysis, testing, runtime measurement, and load profiling into an iterative process for optimizing persistent natural-language skills.

4. **A multi-objective evaluation of parallel code generation.** The dissertation evaluates generated programs across correctness, code quality, runtime, load balance, and model-use cost rather than relying on a single outcome measure.

5. **A persistent external memory for optimization knowledge.** The METAFORGE memory mechanism distills evidence-supported lessons and retrieves them across iterations and tasks.

6. **Evidence regarding cross-task transfer.** Held-out experiments determine whether learned skills and memory improve generation for previously unseen parallel programming tasks.

7. **A model-agnostic alternative to parameter fine-tuning.** Because the approach optimizes external natural-language skills rather than model weights, it can be applied to proprietary or rapidly changing LLMs for which parameter-level training is unavailable.

8. **Insights into the limitations of LLMs for parallel software engineering.** The dissertation characterizes failure modes involving false positives, incomplete repairs, invalid optimizations, performance variability, model dependence, and the tradeoffs among competing software-quality objectives.

9. **An expected ontology-grounded framework for trustworthy persistent skill evolution.** The proposed future work will extend reward-validated skill optimization by representing learned optimization knowledge, provenance, trustworthiness constraints, and independent verification evidence within an ontology-backed knowledge graph. The framework will evaluate whether proposed skill updates satisfy explicit semantic and evaluation-integrity requirements before they are admitted into persistent agent knowledge.

1.10 Significance of the Research

The significance of this research extends beyond the specific task of generating MPI C programs. Parallel software is essential to scientific computing, machine learning, simulation, engineering, finance, and large-scale data processing. However, developing reliable parallel programs requires specialized expertise that is not uniformly available. Tools that help developers detect defects, repair implementations, and improve performance could reduce the expertise barrier associated with parallel computing.

This research also contributes to the broader study of trustworthy LLM-based software engineering. It treats the LLM as one component within a controlled analysis and evaluation loop rather than as an unquestioned source of code. Generated outputs are evaluated using external evidence, and METAFORGE retains optimization knowledge when supported by measured reward outcomes. Such evidence establishes reward validation under the defined optimization objective, but does not necessarily establish semantic or evaluation-integrity trustworthiness. Section 6.1 therefore investigates an additional validation layer for determining whether reward-improving optimization knowledge should be permitted to become persistent.

The dissertation further demonstrates how natural language can serve as an optimization medium. Conventional meta-learning typically modifies model parameters, internal representations, or learned policies. In contrast, METAFORGE optimizes a persistent, human-readable skill specification. This textual representation can be inspected, edited, transferred, and compared across models. The resulting approach may be applicable to other software-engineering domains in which generated artifacts can be evaluated using compilers, analyzers, tests, simulators, or execution metrics.

Although MPI provides the evaluation domain, the underlying methods are intended to general-

ize to other forms of execution-grounded code generation. Potential applications include shared-memory parallelism, accelerator programming, distributed data processing, database query optimization, systems programming, and performance-sensitive scientific software. In each case, domain knowledge and empirical feedback could be converted into persistent instructions that improve future model behavior.

1.11 Scope and Limitations

The primary LLM-centered studies in this dissertation focus on MPI programs written in C. MPI is selected because it provides a mature parallel programming standard, a rich set of domain-specific correctness requirements, and measurable performance characteristics. A complementary study extends the broader principle of domain-grounded learning to autonomous vehicle motion planning, where physical vehicle dynamics and independently measured safety outcomes provide the grounding mechanism.

The research does not claim to guarantee the correctness of arbitrary generated parallel programs. Correctness is evaluated relative to task definitions, available test cases, compiler behavior, and the coverage of the static-analysis tools. A program may pass the evaluation harness while still containing an undetected defect. The dissertation therefore treats LLM assistance as a mechanism for improving software quality rather than replacing formal verification or expert review.

The performance results are also specific to the experimental hardware, MPI implementation, compiler configuration, process count, workload, and runtime environment. Runtime improvement in one environment does not guarantee equivalent improvement on a different architecture. Multiple trials and workload sizes are used to reduce the influence of measurement noise, but the results should be interpreted within the stated experimental context.

The proposed approach optimizes prompts, skills, and external memory rather than the parameters of the underlying LLM. This supports model-agnostic experimentation but introduces dependence on the capabilities and stability of externally provided models. Model updates, deprecations, non-deterministic generation, and differences among providers may affect reproducibility.

The proposed trustworthiness research is intentionally narrower than the general problems of reward hacking or self-improving agents. It focuses on code-generation systems in which optimization knowledge is represented as an explicit, persistent, and inspectable natural-language skill or procedural rule. The ontology-backed framework will evaluate such knowledge against represented semantic, provenance, evaluation-integrity, and evidence requirements; satisfying those requirements will not constitute a proof of universal skill trustworthiness or guarantee detection of behaviors that fall outside the encoded constraints.

Finally, the dissertation focuses on technical correctness, software quality, execution performance, and computational cost. Broader human factors, including developer trust, long-term maintainability, and the use of LLM-generated code in production organizations, are important but remain outside the primary experimental scope.

1.12 Reproducibility and Artifact Availability

Reproducibility is an important consideration throughout this dissertation because the evaluated systems combine stochastic LLM generation, externally provided model APIs, compiler and analysis tools, parallel execution environments, and workload-dependent performance measurements. Accordingly, the completed studies retain the experimental artifacts necessary to reproduce the evaluation procedure and independently inspect the reported results.

For the MPI defect-detection and repair study, these artifacts include the evaluated MPI programs,

defect labels and bug references, experimental prompts, retrieval corpora and configurations, trial-level detection results, repair inputs and outputs, and evaluation scripts. For METAFORGE, the reproducibility materials include the initial natural-language skill specification, complete task and workload definitions, correctness validators, harness implementation, textual-gradient and skill-rewrite prompts, TEXTGRAD configuration, compilation and execution commands, memory storage and retrieval implementation, raw trial logs, token-consumption records, and scripts used to aggregate the reported results.

The final dissertation and associated publications will provide persistent repository references for the releasable artifacts. Credentials, API keys, licensed model components, and machine-specific secrets will not be distributed. Because some experiments depend on externally hosted commercial models, exact bit-for-bit reproduction may not remain possible after model updates or deprecations; the released artifacts are therefore intended to make the experimental procedure, inputs, evaluation logic, and recorded outputs independently auditable even when the original model endpoint is no longer available.

1.13 Pothole-Aware Autonomous Driving and Vehicle Dynamics

1.13.1 Navigating Potholes: Learning Safe Motion Planning with Vehicle Dynamics

1.13.1.1 Background and Motivation

Autonomous vehicles (AVs) are progressing from research prototypes toward large-scale real-world deployment. Commercial robotaxi services have expanded in multiple markets [99, 80, 74, 75], while SAE Level 3 conditional-automation systems have received regulatory approval in Japan, Germany, China, and the United States [30, 56, 9, 79]. Much of autonomous driving re-

search has accordingly emphasized interaction-centric objectives, including collision avoidance, traffic-rule compliance, and efficient navigation. These objectives are essential, but they do not fully address vehicle-dynamics challenges created by off-nominal road surfaces, including low-adhesion conditions and physically degraded infrastructure such as potholes.

Potholes represent a particularly important road-surface hazard because they are both common and capable of producing substantial mechanical and traffic consequences. Approximately 44 million drivers in the United States are affected by potholes annually [1], and pothole-related vehicle damage contributes nearly $3 billion in annual repair costs, with individual incidents often resulting in substantial repair expenses [1, 2]. The mechanical severity of a pothole encounter arises from the abrupt geometric discontinuity at the road surface. When a wheel strikes the pothole edge, the tire–road interaction can generate high-magnitude transient loads that propagate through the wheel and suspension system [77, 119]. These impulsive forces can damage tire and wheel components and can impose large loads on the suspension structure, making pothole handling a vehicle-dynamics problem as well as a geometric path-planning problem.

Modern perception systems provide increasingly rich information about these hazards. Cameras and LiDAR can identify potholes and estimate attributes such as location, dimensions, and depth [18, 39, 110, 3, 105], while vehicle-to-cloud and cooperative-perception mechanisms can extend this awareness beyond the immediate sensing range [27, 105]. Consequently, the principal challenge is increasingly shifting from whether a pothole can be detected to how an automated vehicle should respond once the hazard has been characterized.

Existing control-side responses include predictive and active suspension systems designed to reduce body vibration and suppress impact-induced excitation [18, 6, 64, 8]. Although these systems can improve ride quality and attenuate some road disturbances, their effectiveness is constrained by actuator bandwidth, suspension travel, cost, and limited deployment, particularly for severe

high-energy impacts. Advances in steer-by-wire and brake-by-wire technology have therefore motivated greater interest in motion-planning and control strategies that can respond directly through vehicle trajectory and speed selection [20, 118, 33].

A key limitation of many existing pothole-handling strategies is their emphasis on longitudinal speed reduction or suspension response. Braking can reduce impact severity, but strong deceleration may increase travel time and can create rear-end interaction risk when another vehicle follows closely. Additionally, a pothole is often a localized road defect rather than a lane-spanning obstacle. This creates intra-lane lateral redundancy that may permit the vehicle to bypass the hazard, shift the wheel path away from the pothole, or straddle it while remaining within the lane. Existing methods seldom exploit this lateral maneuvering opportunity despite its potential to substantially reduce direct wheel–pothole interaction [119].

Deep reinforcement learning (DRL) provides a promising mechanism for coordinating these competing actions because it can learn control policies through interaction with an environment and optimize multiple objectives simultaneously [5, 20, 123, 7, 98]. However, vehicle-dynamics-oriented reinforcement-learning problems introduce strict physical failure boundaries and sparse, high-severity outcomes. Unstructured exploration can therefore destabilize learning or drive the policy toward conservative local optima, such as excessive braking [5, 7]. These characteristics motivate training strategies that progressively increase task difficulty while retaining physically interpretable safety constraints.

1.13.1.2 Problem Statement

The central problem considered in this study is that conventional pothole-handling methods do not jointly reason about the physical consequences of tire–pothole impact, the available lateral maneuvering space, longitudinal speed regulation, ride comfort, and interaction safety with surrounding

traffic. Perception systems may identify the location and geometry of a pothole, yet detection alone does not determine the safest vehicle response. Likewise, suspension-oriented mitigation and braking-centered motion planning address only part of the problem. A strategy that minimizes vertical impact through aggressive braking can increase rear-end collision risk, while a purely geometric avoidance maneuver can ignore structural loading and vehicle-dynamics constraints.

The problem is therefore inherently multi-objective and physics constrained: an autonomous vehicle must determine whether to avoid, straddle, or traverse a pothole and must simultaneously select an appropriate speed such that the resulting maneuver remains kinematically feasible, structurally safe, comfortable for occupants, efficient, and compatible with surrounding traffic. Solving this problem requires a decision-making framework that elevates impact-load limits and vertical-dynamics information from low-level vehicle-control considerations into the tactical planning process.

This study addresses that gap by formulating pothole handling as coordinated lateral–longitudinal tactical decision-making informed by explicit vehicle dynamics. The proposed framework combines a physics-based representation of wheel–pothole impact and vertical vehicle response with a learning-based decision policy so that structural safety, ride comfort, traffic interaction, and passage efficiency can be evaluated within a common closed-loop environment.

1.13.1.3 Research Objectives

The objective of this research is to develop and evaluate a vehicle-dynamics-oriented hierarchical framework for autonomous pothole handling that integrates physical impact constraints directly into tactical motion planning. Rather than relying exclusively on braking or low-level suspension response, the framework coordinates lane-bounded lateral offsets with longitudinal speed selection so that the vehicle can exploit geometric avoidance opportunities when feasible and regulate impact

severity when avoidance is not possible.

The specific research objectives are:

1. **Integrate vehicle-dynamics constraints into tactical motion planning.** Develop a hierarchical planning framework that incorporates physical impact-load limits and vertical-dynamics objectives into high-level decisions while producing kinematically feasible trajectories for downstream execution.

2. **Construct a unified multi-physics evaluation environment.** Develop a Gymnasium-based simulation testbed that combines vehicle vertical dynamics, structural tire–pothole impact mechanics, and interactive car-following behavior so that structural safety, ride comfort, passage efficiency, and traffic safety can be evaluated within a single closed-loop framework.

3. **Learn coordinated lateral–longitudinal pothole-handling strategies.** Apply a maximum-entropy Soft Actor-Critic (SAC) policy to select target speed and lateral offset jointly, enabling the emergence of avoidance, straddling, and impact-limited traversal strategies rather than prescribing these behaviors through fixed rules.

4. **Improve training stability across heterogeneous pothole conditions.** Employ curriculum learning to address sparse impact feedback, strict physical failure boundaries, and substantial variation in pothole geometry by progressively increasing task complexity during training.

Together, these objectives extend autonomous motion planning beyond geometric obstacle avoidance by explicitly connecting tactical decisions to the mechanical consequences of road-surface interaction. Within the broader dissertation, this study provides a complementary example of domain-grounded machine learning in which physically meaningful models, safety boundaries, and execution outcomes are incorporated directly into the learning and decision-making process.

Chapter 2: LITERATURE REVIEW

This chapter reviews the research foundations underlying the dissertation's investigation of domain-grounded learning in technically constrained systems. The literature is organized into four related strands. The first examines classical and large-language-model-based approaches to detecting and repairing defects in Message Passing Interface (MPI) programs. The second reviews execution-grounded code optimization, persistent language-model memory, and performance engineering for high-performance computing (HPC), providing the foundation for persistent skill optimization and cross-task knowledge transfer. The third examines emerging concerns surrounding reward hacking, self-evolving agents, persistent learned skills, and semantic validation, motivating the proposed future work on trustworthy skill evolution through ontology-grounded validation. The fourth reviews pothole-aware vehicle control and reinforcement learning as a complementary study of domain-grounded learning in a safety-critical physical system. Together, these areas establish the technical context for the completed studies and the proposed future research presented in the remainder of this dissertation.

2.1 Classic Error Detection and Repair in MPI Programs

MPI is a foundational programming model for distributed-memory HPC, supporting large-scale scientific applications and distributed machine-learning workloads. Its flexibility, however, introduces correctness challenges that are substantially more difficult than many defects encountered in sequential software. MPI programs may exhibit non-deterministic execution, wildcard communication, process-dependent control flow, collective synchronization, and communication relationships whose correctness depends on operations performed by multiple ranks. These characteristics make debugging difficult because a defect may manifest only for particular process counts, mes-

sage orderings, or runtime interleavings. The MPI Bugs Initiative provides a systematic framework for evaluating MPI verification tools and illustrates the breadth of defects that arise from these communication and synchronization semantics [43].

Static analysis examines source code without executing the target program and can identify defects such as incorrect type usage, invalid buffer operations, request-lifecycle errors, and some communication mismatches. MPI-CHECK extends Clang-based static analysis with MPI-specific reasoning, including checks for non-blocking requests and selected point-to-point communication properties [19]. Static analysis is attractive because it can be applied without enumerating runtime executions, but its ability to represent dynamic communication behavior is necessarily limited. In particular, wildcard communication, runtime-dependent control flow, and complex process interactions can produce false positives or allow execution-dependent defects to escape detection [43]. Hybrid techniques have therefore been investigated to improve precision for narrowly defined defect classes. Ma *et al.*, for example, reformulate thread-safety violations in hybrid MPI/OpenMP software as race conditions and use targeted instrumentation around MPI calls, achieving efficient detection while remaining focused on a specific class of concurrency error [54].

Dynamic analysis evaluates program behavior during execution and is therefore well suited to defects that depend on synchronization or runtime communication ordering. ISP applies model-checking concepts to MPI executions and uses dynamic partial-order reduction to explore relevant schedules [92]. MUST performs runtime analysis of MPI programs and includes mechanisms for detecting deadlocks and other incorrect MPI usages [29]. Generic event-based tool infrastructures such as GTI support scalable runtime-analysis tools by organizing the collection and processing of events from parallel executions [28]. These approaches improve visibility into actual execution behavior, but their coverage is limited to the paths and schedules exercised or systematically explored, and instrumentation or schedule exploration can introduce substantial runtime overhead as process counts increase [48, 28]. Consequently, dynamic techniques provide valuable evidence for

particular executions without guaranteeing practical coverage of every behavior in a large parallel program.

Symbolic execution and model checking attempt to reason more systematically about alternative program paths and communication schedules. TASS applies symbolic execution and state-space exploration to MPI programs and can verify safety properties and functional equivalence for bounded configurations [85]. CIVL similarly transforms C/MPI programs into a verification representation that can be analyzed for functional correctness, deadlock freedom, and conformance with selected MPI rules [53]. MPI-SV combines symbolic verification techniques to reason about MPI programs [14], while subsequent work has extended symbolic reasoning to one-sided MPI communication [32]. Related research also combines symbolic execution and model checking to improve MPI verification coverage [112]. These approaches offer stronger reasoning over unexplored inputs and schedules than conventional testing, but the number of states and paths can grow combinatorially, making scalability a persistent challenge.

Concolic testing combines concrete execution with symbolic reasoning to reduce the cost of exhaustive symbolic exploration. COMPI applies concolic testing to MPI applications and uses concrete executions to guide symbolic input generation [48]. Later work improves COMPI through input tuning and support for floating-point behavior, increasing coverage while reducing unnecessary exploration [49]. Although concolic methods mitigate some of the path-explosion problem, they remain constrained by non-deterministic execution, large input domains, and the increasing communication-state space of realistic MPI programs.

Collectively, classical MPI analysis has established effective techniques for specific correctness properties, defect classes, and bounded execution scenarios. Nevertheless, no single technique provides comprehensive, low-overhead coverage of the heterogeneous defects found in real MPI software. Static techniques are limited by their approximations of dynamic behavior and high

false-positives, runtime techniques are constrained by executed paths and instrumentation cost, and symbolic or concolic techniques face state-space growth as programs scale [92, 49, 85, 14, 32, 17]. These limitations motivate complementary techniques that can reason from source-code context, domain knowledge, and heterogeneous analysis evidence rather than relying on a single fixed analysis mechanism.

2.2 Modern Software Defect Detection and Repair with Large Language Models

Recent advances in large language models (LLMs) have substantially expanded their application to software-engineering tasks, including code generation, program understanding, debugging, fault localization, software testing, vulnerability detection, code refinement, and automated program repair [21]. Unlike conventional software-analysis techniques, which generally rely on predefined rules, program abstractions, or explicit execution models, LLMs learn statistical representations of both natural language and source code and can reason jointly over program syntax, surrounding context, natural-language descriptions, diagnostic information, and examples. These capabilities have motivated increasing investigation into whether LLMs can augment or complement traditional software-quality techniques, particularly for defects that require contextual or semantic reasoning.

One major research direction applies LLMs to *debugging and fault localization*. Systems such as SoapFL and FlexFL use language-model reasoning to identify program regions likely to contain defects, illustrating the potential for LLMs to assist developers in narrowing the search space before repair [76, 107]. A related body of work integrates LLMs with static-analysis results in an effort to improve the interpretation of analyzer findings and reduce false-positive or false-negative behavior [47, 46, 104]. These hybrid approaches are particularly relevant because they combine deterministic or analysis-derived evidence with the broader contextual reasoning capabilities of a language model rather than requiring either mechanism to operate in isolation.

LLMs have also been increasingly investigated for *software testing* and *automatic program repair*. Research on LLM-based testing examines their ability to generate test cases, infer assertions, and incorporate program semantics into test construction [96, 63, 78]. In automated program repair, code-oriented models such as CodeBERT and Codex have been used to generate candidate patches and semantically meaningful code transformations [55, 42, 15]. Additional studies demonstrate that prompt design can materially affect LLM performance for software vulnerability detection and repair [10, 116]. Collectively, this literature demonstrates substantial potential for LLM-assisted software quality assurance while also identifying persistent challenges involving prompt sensitivity, incomplete program context, false-positive and false-negative predictions, computational cost, and dependence on domain-specific knowledge [122, 23, 21].

These limitations are particularly significant for parallel software. MPI defects frequently depend on relationships among operations executed by different processes rather than on an isolated syntactic pattern. Deadlocks, mismatched communications, collective inconsistencies, race conditions, and rank-dependent failures therefore require reasoning about communication semantics and distributed control flow that may be poorly represented by general-purpose software-engineering examples. The following subsections review four areas of LLM-based software-quality research most relevant to this dissertation: debugging and fault localization, LLM-augmented static analysis and defect detection, software testing, and automatic program repair. A final subsection examines prompt engineering and code-oriented language models as mechanisms for introducing the specialized knowledge required for domain-specific software analysis.

2.2.1 LLM-Based Debugging and Fault Localization

Fault localization is a central step in debugging because repair cannot be reliably performed until the region responsible for the observed failure has been identified. LLM-based approaches ex-

ploit learned representations of code and natural language to rank or identify potentially defective functions. SoapFL uses LLM-supported reasoning to localize buggy functions from larger code bases [76]. Its workflow relies on information from failed tests, which can provide strong failure evidence but also assumes the availability and adequacy of a test suite. FlexFL reduces this dependency by supporting fault localization without requiring failed unit-test output as a mandatory input, thereby improving applicability to projects for which tests are incomplete or unavailable [107].

Despite these advances, LLM-based fault localization remains sensitive to the scope of program context and the bug-related evidence available to the model. SoapFL addresses method-level localization over larger code scopes, while FlexFL specifically addresses the dependence of earlier LLM-based methods on bug-triggering test cases by allowing multiple forms of bug-related information. These limitations are particularly relevant to MPI, where a defect may not be localized to one function but may instead arise from an incorrect relationship among operations executed by different processes [76, 107].

2.2.2 LLM-Augmented Static Analysis and Defect Detection

Another research direction combines conventional static analysis with LLM reasoning. In these systems, static analyzers provide structured evidence about possible defects while the LLM interprets, filters, or supplements the reported findings. The objective is often to reduce false positives or to identify defects that are difficult to capture using a fixed rule set [47, 46, 104].

LLift integrates static-analysis results with LLM reasoning to identify use-before-initialization defects in the Linux kernel [47]. The work demonstrates that an LLM can contribute semantic reasoning beyond the underlying analyzer, including the discovery of previously unreported defects. However, the approach is specialized to a narrow defect category and requires analysis findings to

be incorporated into the LLM interaction. Its reported false-positive behavior also illustrates that adding an LLM does not, by itself, guarantee a production-quality detector. AdvScanner combines static analysis with LLM generation by extracting attack flows for reentrancy vulnerabilities and using them to guide generation of adversarial smart contracts [104]. Its scope is therefore a specific smart-contract vulnerability class rather than the general communication and synchronization semantics required for MPI analysis.

More generally, empirical work comparing LLMs with conventional analyzers shows both the promise and limitations of generative models for static defect detection. Empirical work also shows that LLM reasoning can complement conventional static analysis by filtering false-positive findings and, in some cases, identifying defects missed by the underlying analyzer [46]. These results support a hybrid perspective in which analyzers provide measurable program evidence while LLMs contribute context-sensitive interpretation rather than replacing deterministic analysis entirely.

2.2.3 LLM-Based Software Testing

Software testing is another area in which LLMs have been used to reduce the human effort required to construct test cases. Test generation differs from general program synthesis because a useful test must capture behavioral expectations, create appropriate inputs, and formulate assertions capable of distinguishing correct from incorrect behavior. As a result, effective LLM-based testing frequently depends on representations that expose semantic relationships between the code under test and the corresponding test logic [21, 96].

TeCo is a neural test-completion model that incorporates multiple forms of code semantics, including execution-derived information, when predicting the next statement in a test method [63]. CAT-LM extends this direction with a GPT-style language model trained on aligned source code and tests and uses a longer context window to expose additional code-under-test information during

generation [78]. Together, these systems show that semantics-aware neural and language-model approaches can improve automated test generation when program context and execution-relevant information are incorporated explicitly.

For parallel software, this limitation is especially consequential because correctness may depend on process count, nondeterministic schedules, communication ordering, and synchronization. A test suite that validates one execution configuration may therefore fail to exercise an MPI defect that appears under another configuration. LLM-generated testing can contribute to the evaluation process, but it is most useful when combined with domain-specific validation and execution evidence.

2.2.4 LLM-Based Automatic Program Repair

LLMs have also been incorporated into automatic program repair (APR), where the objective is to transform defective source code into a semantically valid implementation. Code-oriented language models can generate candidate patches using learned relationships among source-code patterns, natural-language descriptions, and examples of code modification. Codex-based repair studies demonstrate that large pretrained code models can produce plausible and semantically meaningful patches [42]. SequenceR applies sequence-to-sequence learning with a copy mechanism to end-to-end source-code repair [15]. Separately, pretrained code representations such as CodeBERT have been adapted to repair simple Java defects [55].

The ability to generate plausible patches does not eliminate the need for verification. LLM-generated repairs can alter program behavior beyond the target defect, and their quality depends strongly on prompt design and the model's knowledge of the target domain. Existing APR research therefore continues to report limitations associated with computational cost, hallucinated or overbroad edits, prompt sensitivity, and dependence on domain-specific information [122, 23].

Code-refinement studies likewise show that ChatGPT can propose useful program changes but that candidate modifications require external validation [26]. CodeBERT-based vulnerability detection further demonstrates the value of code-specialized representations for C and C++ security analysis [106].

2.2.5 Prompt Engineering and Code-Oriented Language Models

Prompt design provides a lightweight mechanism for introducing task-specific knowledge without retraining the underlying model. For software defects, carefully structured prompts can supply examples, constraints, defect descriptions, or reasoning guidance that alter how the model interprets a program. Cao *et al.* study prompt design for detecting and repairing defects in programs that use deep-learning libraries [10], while Zhang *et al.* demonstrate that prompt construction affects ChatGPT-based software vulnerability detection [116]. These findings are significant for domain-specific software because relevant knowledge may not be sufficiently represented in the model's default inference behavior even when the model has substantial general coding capability.

The software-engineering LLM ecosystem includes both commercial and open-weight systems. Commercial coding assistants and agentic models such as OpenAI Codex, Anthropic Claude Code, and Google Gemini Code Assist coexist with openly available code-oriented models such as Code Llama and Qwen2.5-Coder. These systems offer increasingly capable code generation, review, and transformation, yet strong general coding performance does not guarantee reliable defect detection. Models can accept superficially plausible code, miss errors requiring precise semantic or concurrency reasoning, or report defects that are not actually present. For MPI software, these limitations are amplified because correctness frequently depends on cross-rank communication relationships rather than on a local code pattern.

This body of work therefore motivates explicit domain grounding for parallel-program analysis.

General LLM-based debugging, static-analysis augmentation, testing, and APR demonstrate that language models can contribute meaningful software reasoning, but MPI defects remain comparatively underexplored. Deadlocks, message mismatches, collective inconsistencies, and rank-dependent execution require specialized communication knowledge and reasoning over relationships that do not have direct analogues in sequential software. Domain-informed examples, retrieval, and structured reasoning offer a mechanism for supplying this missing information without requiring parameter fine-tuning.

2.3 Execution-Grounded Code Optimization

Beyond defect detection and repair, a growing body of research treats language-model-based software engineering as an iterative process in which generated artifacts are evaluated by external tools and the resulting evidence is used to guide subsequent generations. This execution-grounded perspective is important because language models cannot reliably infer functional correctness or runtime performance from source-code plausibility alone.

Reflexion uses verbal feedback derived from task outcomes to improve subsequent agent behavior [84]. SWE-Agent and AutoCodeRover demonstrate that language-model agents can interact with software-engineering environments, inspect repositories, invoke tools, and refine code in response to observed results [109, 120]. These systems establish the value of iterative tool-assisted refinement in which the model is embedded within a larger software-development loop rather than treated as a one-shot generator.

Other systems make the optimization process itself a primary object of study. Language Agents as Optimizable Graphs explores the systematic optimization of agentic workflows [127], while AlphaEvolve demonstrates the use of language-model-guided evolutionary refinement for scientific

and algorithmic discovery [65]. TextGrad formalizes textual feedback as an analogue of gradient information and uses natural language to optimize prompts and other textual variables without modifying model weights [114]. Subsequent work extends this perspective toward broader generative-AI optimization and end-to-end harness optimization [114, 44].

These approaches demonstrate that external feedback and textual updates can improve agent behavior, but they address optimization settings that differ from distributed-memory performance engineering. Recent work by Wei *et al.* optimizes application-specific parallel-program mappers using execution feedback and an AutoGuide mechanism that converts raw execution output into actionable guidance [100]. The reported workflow, however, does not describe a persistent cross-task memory mechanism for retaining optimization lessons across benchmark tasks.

More broadly, systems such as TEXTGRAD, Meta-Harness, and ParaCodex demonstrate increasingly sophisticated feedback-driven optimization and, in some cases, substantial reuse of prior optimization trajectories [114, 44, 40]. What remains comparatively unexplored is the representation, validation, and transfer of persistent execution-grounded optimization knowledge specifically for distributed-memory MPI code generation. This gap motivates a framework in which both the persistent natural-language skill and external optimization memory can carry evidence-supported knowledge across parallel-programming tasks.

The literature therefore establishes execution feedback as an effective basis for iterative refinement while leaving open the question of how measured optimization experience should be represented, retained, and transferred across parallel-programming tasks. This gap motivates approaches in which the object being optimized is not only the current program, but also the persistent natural-language instructions and memory used to guide future generation.

2.4 Memory-Augmented Agents

Persistent memory has emerged as an important mechanism for extending language-model agents beyond the context of a single interaction. Memory systems can retain previous observations, decisions, summaries, or task outcomes and selectively reintroduce relevant information during future reasoning, supporting continuity across interactions without modifying the underlying model parameters [125, 71, 72].

Several systems illustrate different forms of persistent agent memory. MemoryBank stores and updates long-term conversational memories [125]; MemGPT manages information across constrained context windows through an operating-system-inspired memory architecture [71]; and Generative Agents stores and retrieves accumulated experiences to support persistent simulated behavior [72]. More recent systems such as MemInsight and Mem0 further investigate autonomous or structured memory management for LLM agents [82, 16].

Retrieval-augmented generation (RAG) provides a complementary mechanism for selectively incorporating external information. RAG retrieves relevant documents or records from an external knowledge source and places that information in the model context before generation [45]. In software engineering, RepoCoder applies iterative retrieval to repository-level code completion, demonstrating that access to relevant cross-file context can improve generated code [117]. These results show that external memory can expand the effective information available to the model and reduce dependence on information encoded solely in model parameters.

For software optimization, however, storing conversation history or source code alone is not equivalent to storing optimization knowledge. A reusable optimization memory must represent not only what code was generated, but also what behavior was observed, which changes improved or degraded execution, and under what structural or task conditions a strategy was effective. Compiler

diagnostics, static-analysis findings, correctness outcomes, profiling results, and reward changes provide evidence that can distinguish a useful optimization lesson from an unverified suggestion. This distinction motivates memory representations that associate natural-language guidance with the execution evidence supporting it.

Accordingly, prior memory-augmented agents establish the feasibility of persistent external knowledge, while retrieval-oriented software systems demonstrate the value of selective contextualization. The remaining research challenge for parallel-code optimization is to transform execution trajectories into compact, evidence-supported lessons that can be retrieved by structural, task, correctness, and performance characteristics rather than by conversational similarity alone.

2.5 Performance Analysis and Optimization in HPC

Performance engineering in HPC has long relied on direct measurement of program execution. Unlike conventional functional testing, performance analysis must identify where time is spent, how communication and synchronization affect scalability, and whether work is distributed effectively across processing elements. Profiling and tracing therefore provide empirical evidence that cannot be obtained reliably from source-code inspection alone.

HPCToolkit provides measurement and analysis capabilities for optimized parallel programs and supports attribution of performance costs to program structure [4]. Score-P provides a common performance-measurement infrastructure used by multiple HPC analysis environments and supports collection of execution information relevant to parallel performance [41]. Such systems enable developers to identify communication overhead, synchronization delay, load imbalance, and computational bottlenecks that may not be visible from program output.

Correctness-analysis tools complement these profilers. MUST detects incorrect MPI usage and

deadlock-related behavior during execution [29], while PARCOACH uses static analysis to identify classes of MPI communication errors, including nonblocking and persistent-communication problems [62]. LLM-based MPI defect analysis provides an additional source of semantic reasoning for communication and synchronization defects [73]. Together, these tools demonstrate the value of combining multiple forms of evidence when reasoning about parallel software: correct output alone does not imply absence of communication defects, and absence of correctness defects does not imply efficient execution.

Recent autonomous code-generation systems have begun to incorporate HPC performance evidence directly into LLM workflows. ParaCodex uses a Codex-based agentic process that integrates correctness checking, hotspot analysis, data-movement planning, profiling, and iterative refinement for OpenMP GPU-offload generation and translation [40]. This work demonstrates that execution-guided LLM refinement can produce performant parallel code when the model is embedded within a domain-specific tool chain. At the same time, the workflow is focused on GPU-offload translation and does not explicitly retain previously learned optimization strategies as cross-task memory. The paper also reports substantial token consumption for both ParaCodex and its automated baseline, illustrating the cost of non-interactive agentic optimization.

The HPC literature therefore provides two important foundations for this dissertation. First, compiler diagnostics, static analysis, correctness checking, runtime profiling, and load-balance measurement offer complementary signals for evaluating parallel software. Second, recent LLM-based systems demonstrate that these signals can be incorporated into iterative code generation and optimization. What remains less explored is how heterogeneous execution evidence can be transformed into persistent natural-language knowledge that improves future generation across tasks without updating model parameters. This research gap connects traditional HPC performance engineering with execution-grounded meta-learning and memory-augmented language-model systems.

2.6 Trustworthy Persistent Skill Evolution and Ontology-Grounded Validation

Persistent natural-language skills and external memory create an additional trust boundary for self-improving language-model systems. Once optimization knowledge is retained beyond the episode in which it was learned, an incorrect or reward-exploiting strategy can potentially influence multiple subsequent generations. This concern is distinct from ordinary generation errors because the problematic artifact is no longer only the generated program; it can become part of the persistent instructions that govern future model behavior.

Recent research establishes several components of this problem. Reward hacking has been observed during iterative self-improvement of code agents, showing that optimization can favor changes that improve a measured proxy without correspondingly improving the intended objective [121]. Research on self-evolving agents further demonstrates that persistent memory and self-improvement mechanisms can produce unintended behavioral degradation even without an externally inserted malicious skill [83]. At the same time, systems such as CODESKILL demonstrate that software-engineering trajectories can be distilled into reusable procedural skills that influence later tasks [50].

Security research establishes a complementary threat model in which persistent skills themselves become attack surfaces. SkillJack, for example, demonstrates that externally poisoned experience can be processed by an agent's self-evolution mechanism and converted into a durable reusable skill [111]. Such work establishes that harmful information can propagate through persistent skill-learning mechanisms, although the initiating cause is external adversarial influence rather than ordinary reward-guided optimization.

Taken together, these studies establish reward exploitation, unsafe self-evolution, reusable procedural skills, and persistent skill compromise as existing research areas. They do not, however,

directly establish the complete causal sequence in which a code agent autonomously discovers a reward- or evaluation-exploiting strategy during ordinary optimization, explicitly promotes that strategy into a persistent reusable natural-language skill, and then realizes or transfers the strategy through later generations. The remaining research gap is therefore narrower than reward hacking or persistent skill security individually; it concerns the trustworthiness of the *optimization knowledge admitted into persistent skill policy*.

Structured semantic knowledge provides one possible mechanism for governing this boundary. The Web Ontology Language (OWL 2) supports explicit representation of concepts and semantic relationships, while the Shapes Constraint Language (SHACL) provides machine-evaluable constraints over graph representations [94, 95]. An ontology-backed knowledge graph can therefore represent relationships among learned skills, tasks, objectives, evaluators, measurements, provenance, program behavior, and trustworthiness requirements. The unresolved question is whether such semantic representation, combined with independent executable evidence, can determine whether newly learned optimization knowledge should be allowed to become persistent. This question motivates the ontology-grounded skill-validation research proposed in Section 6.1.

The preceding sections motivate the dissertation's primary LLM-centered research program: domain-grounded defect detection and repair, execution-grounded skill optimization, persistent knowledge transfer, and the trustworthiness of learned optimization knowledge. The dissertation also includes a complementary study that examines the same broader principle of domain grounding in a safety-critical physical system. Rather than grounding an LLM in software-engineering knowledge and execution evidence, this study grounds reinforcement learning in vehicle dynamics, physical impact models, traffic interaction, and independently measurable safety outcomes. The following section reviews the literature supporting that complementary study.

2.7 Pothole-Aware Autonomous Driving and Vehicle Control

Pothole-aware autonomous driving lies at the intersection of road-surface perception, vehicle dynamics, motion planning, and safety-critical control. Although modern autonomous vehicles can increasingly perceive road-surface hazards, effective mitigation requires more than geometric obstacle detection. The response must also account for transient tire–road interaction, vertical vehicle dynamics, ride comfort, structural loading, available lane geometry, and the behavior of surrounding traffic. The literature relevant to this problem can therefore be organized into three complementary areas: vehicle control technologies for pothole-hazard driving, reinforcement learning for autonomous-driving and vehicle-control applications, and the role of potholes as recurrent pre-crash events.

2.7.1 Vehicle Control Technologies for Pothole-Hazard Driving

Research on pothole-aware vehicle control has increasingly moved from isolated hazard detection toward integrated perception, planning, and control systems that use preview information to reduce impact severity and ride discomfort while maintaining an executable vehicle trajectory [11, 39, 110]. Contemporary autonomous-driving systems can detect potholes using cameras, LiDAR, and multi-sensor perception and can estimate geometric characteristics such as pothole position, width, and depth [110, 3, 105]. These capabilities make pothole-aware planning technically feasible; however, many existing approaches formulate the problem primarily as geometric obstacle avoidance. Such formulations do not necessarily account for the transient loads generated during a wheel–pothole interaction or the resulting effects on suspension dynamics, structural safety, and ride comfort.

Two major approaches dominate the vehicle-control literature for rough or damaged road surfaces:

preview-based suspension control and longitudinal speed regulation. Preview-based suspension methods exploit look-ahead road information to adjust active or semi-active suspension behavior before the vehicle reaches a surface disturbance, thereby reducing body motion and wheel-load variation relative to purely reactive control [88]. Longitudinal speed regulation addresses the same problem from the motion-planning perspective by modifying traversal speed before the vehicle encounters the disturbance, since pothole-induced vertical excitation depends on both surface geometry and vehicle velocity [86]. More recent work combines preview suspension control with velocity planning to jointly optimize ride comfort, longitudinal smoothness, travel efficiency, and impact mitigation [102, 52, 34].

This progression demonstrates increasing recognition that pothole mitigation is a coupled planning-and-control problem rather than an exclusively suspension-level problem. Nevertheless, most prior approaches remain strongly oriented toward longitudinal mitigation. Reducing speed before impact can lower vertical excitation, but excessive braking may reduce traffic efficiency and create interaction risks with closely following vehicles. Furthermore, potholes are typically spatially localized hazards, which means that the vehicle may possess useful lateral redundancy within the current lane. Small lateral offsets can allow a vehicle to bypass a pothole, align the hazard between the wheels, or reduce direct tire contact without requiring a full lane change. This source of maneuvering flexibility has received less attention than longitudinal speed control.

Recent pothole-aware planning research has also begun to incorporate interaction safety. For example, road-induced vibration observed in a leading vehicle can be used to anticipate upcoming surface disturbances and support speed planning that accounts for the surrounding traffic environment [118]. Such work broadens the problem from isolated vehicle vibration mitigation to interactive motion planning under incomplete or delayed information.

Taken together, the literature establishes substantial progress in pothole detection, preview suspen-

sion control, and longitudinal speed planning, but it also reveals two important limitations. First, vehicle-dynamics quantities such as transient impact load and vertical response are often treated as low-level control objectives rather than explicit constraints within tactical motion planning. Second, lane-bounded lateral maneuvering remains comparatively underexplored despite its potential to avoid or straddle localized hazards. These limitations motivate motion-planning formulations that coordinate lateral positioning and longitudinal speed while explicitly accounting for physical impact limits, ride comfort, and traffic interaction.

2.7.2 Reinforcement Learning for Autonomous Driving and Vehicle Control

Reinforcement learning (RL) has become an important framework for autonomous driving because it can optimize decision policies through repeated interaction with nonlinear and stochastic environments. By embedding multiple objectives into the reward function, RL can represent trade-offs among safety, comfort, efficiency, and constraint satisfaction within a common optimization process. Surveys of RL-based autonomous driving describe applications across the autonomy stack, including high-level behavior planning, motion planning under kinematic and dynamic constraints, and low-level vehicle control [103, 5, 31, 98]. These studies collectively indicate that RL is particularly useful in complex or long-tailed scenarios in which fixed rules are difficult to design and maintain.

A related body of research applies RL to vehicle-control problems with strong dynamic coupling, uncertain model parameters, and competing physical objectives. Representative applications include emergency collision avoidance, car-following control, and speed planning over rough or degraded road surfaces [123, 97, 20]. These problems illustrate a central advantage of learning-based control: instead of optimizing each subsystem independently, an RL policy can learn a coordinated response from the combined consequences of its actions.

However, unrestricted RL exploration is problematic in safety-critical vehicle applications. Unsafe actions can violate physical operating limits, create collisions, or expose the vehicle to damaging loads. Consequently, recent work increasingly incorporates physics-guided and constraint-aware learning formulations that restrict the action space to physically meaningful controls or explicitly enforce safety boundaries [123]. Research in safe RL further investigates constrained optimization, risk-sensitive objectives, and formal safety filters for improving constraint satisfaction during training and deployment [38]. Hybrid RL–model-predictive-control (MPC) architectures provide another approach by combining the adaptation capability of RL with the explicit constraint handling and model structure of MPC [97].

For pothole-hazard driving, the primary challenge is therefore not simply selecting an RL algorithm, but constructing an environment and objective function that represent the relevant physical and traffic processes with sufficient fidelity. A useful learning environment must connect tactical actions to wheel–pothole impact mechanics, vertical vehicle response, available lateral maneuvering space, and interaction risk from surrounding traffic. The state and action representations must remain physically interpretable, while the reward must distinguish between efficient hazard passage and unsafe strategies such as excessive braking or structurally damaging impact.

The training process is also complicated by the heterogeneous geometry of potholes and the sparsity of severe failure events. Policies that are exposed immediately to the full range of possible pothole dimensions and traffic conditions may converge to overly conservative strategies or unstable local solutions. Multi-physics simulation and structured training strategies are therefore particularly relevant for this domain. Open simulation platforms and datasets provide a foundation for constructing such environments, while RL can supply the adaptive decision mechanism needed to coordinate lateral and longitudinal responses [87, 98]. This literature supports the use of maximum-entropy and curriculum-based RL formulations for pothole handling, provided that the learning process remains grounded in explicit physical constraints and measurable safety out-

comes.

2.7.3 Pothole Hazards as Frequent Pre-Crash Events

The significance of pothole-aware control extends beyond vehicle durability and occupant comfort. Naturalistic driving studies conducted by the National Highway Traffic Safety Administration (NHTSA) identify roadway-surface hazards, including potholes, as frequent pre-crash events [59, 60, 61]. These studies associate roadway anomalies with abrupt steering, uncontrolled or rapid braking, and sudden path changes. Such responses can destabilize the vehicle, reduce available reaction time, and increase the probability of both single-vehicle loss-of-control events and conflicts with surrounding traffic.

This observation changes the role of pothole mitigation in autonomous driving. A control action that protects the wheel or improves ride comfort is not necessarily the safest action at the traffic-system level. For example, strong braking may reduce the load generated by a pothole impact but can simultaneously increase rear-end collision risk when another vehicle is following closely. Likewise, an aggressive evasive steering maneuver can avoid direct wheel contact while introducing lateral instability or conflict with adjacent road users. Pothole handling must therefore be evaluated as a multi-objective safety problem rather than solely as a road-surface comfort problem.

Learning-based driver-assistance and autonomous-driving systems provide a potential mechanism for balancing these competing objectives. Instead of using a fixed response for every pothole encounter, an RL policy can condition its action on pothole geometry, vehicle speed, available preview distance, and the relative state of surrounding traffic. Reward functions can penalize excessive deceleration, unsafe following conditions, vehicle instability, and damaging impact loads while encouraging smooth and efficient passage. This makes it possible to learn context-dependent behavior in which braking, lateral repositioning, or a combination of the two is selected according

to the specific hazard and traffic configuration.

The NHTSA findings therefore provide an important safety motivation for pothole-aware autonomous planning. Road-surface defects should not be treated only as infrastructure-maintenance concerns or sources of mechanical damage; they can initiate evasive behaviors that precede more serious crashes. Accordingly, autonomous pothole-handling systems should be designed to reduce both the immediate physical consequences of the road defect and the secondary traffic risks created by the vehicle's response [61].

2.8 Synthesis of Research Gaps

The preceding literature identifies a set of related gaps that motivate the research questions investigated in this dissertation. Classical MPI-analysis techniques provide strong guarantees for selected defect classes and execution conditions, but no single approach provides comprehensive, low-overhead coverage of the heterogeneous communication and synchronization defects that arise in parallel programs. At the same time, general LLM-based debugging and repair methods demonstrate substantial software-reasoning capability but are not inherently grounded in MPI communication semantics. This gap motivates RQ1 and RQ2, which examine whether explicit MPI bug knowledge, representative examples, structured reasoning, and domain grounding improve defect detection and repair.

Retrieval provides a mechanism for supplying external knowledge during inference, but conventional retrieval similarity does not necessarily encode the relational communication semantics that determine MPI correctness. Accordingly, RQ3 examines whether retrieval-augmented generation can select and apply useful MPI knowledge and where conventional retrieval mechanisms fail in this domain.

Execution-guided software agents and textual-gradient methods demonstrate that external evidence can support iterative improvement. HPC profiling and correctness-analysis tools further establish that parallel software must be evaluated using heterogeneous evidence that includes correctness, static quality, runtime behavior, and process-level utilization. What remains less explored is whether these signals can jointly refine a persistent natural-language skill for multi-objective MPI code generation. RQ4 and RQ5 address this gap by evaluating execution-grounded skill refinement and its effect on generated-program correctness, code quality, runtime, and load imbalance.

Persistent-memory systems demonstrate that language-model agents can retain and retrieve information across interactions, but persistent conversational or source-code memory is not equivalent to evidence-supported optimization knowledge. Comparatively little work has examined whether execution-derived parallel-programming lessons can be encoded into persistent natural-language skills and external memory and then improve previously unseen MPI tasks. RQ6 therefore examines cross-task transfer, while RQ7 characterizes the limitations, failure modes, costs, and tradeoffs associated with these domain-grounded systems.

The future work extends this progression from the *effectiveness* of persistent skill optimization to the *trustworthiness* of the knowledge being persisted. Reward-hacking, self-evolution, and skill-security research show that improved measured performance does not by itself establish that a learned strategy is trustworthy. The proposed ontology-grounded validation research therefore investigates the emergence and persistence of untrustworthy skill knowledge, its propagation and transfer, the effectiveness of semantic pre-commit validation, and the tradeoff between trustworthiness enforcement and legitimate optimization utility.

Finally, the pothole-aware autonomous-driving study provides a complementary demonstration of the dissertation's broader grounding principle. In that study, the learned policy is grounded not in MPI semantics and software-analysis evidence, but in vehicle dynamics, tire–pothole impact

mechanics, traffic interaction, and measurable physical-safety constraints. It therefore broadens the methodological evidence for domain-grounded learning without serving as evidence for RQ1–RQ7, which remain specifically focused on LLM-based parallel software engineering.

Chapter 3: Improving MPI error detection and repair with large language models and bug references

This chapter presents our approach for improving the ability of large language models (LLMs) to detect and repair software defects in Message Passing Interface (MPI) programs through explicit domain grounding. We investigated whether supplementing an LLM's pretrained knowledge with representative MPI bug references and domain-expert reasoning could improve its ability to recognize complex parallel-programming defects that are difficult to identify using either general-purpose LLM prompting or conventional software-analysis techniques. Our approach combined Few-Shot Learning with annotated MPI defect examples, MPI-specific Chain-of-Thought reasoning, and Retrieval-Augmented Generation to evaluate different mechanisms for incorporating domain knowledge into the model's analysis process. We further examined whether improvements in defect detection could support automated repair of defective MPI programs and whether those improvements generalized across multiple general-purpose and code-specialized LLMs. This work establishes the first component of the dissertation's broader investigation into domain-grounded LLMs for parallel software engineering by demonstrating how explicitly encoded domain knowledge can improve model reliability without requiring parameter fine-tuning.

3.1 Introduction

High-performance computing (HPC) applications increasingly rely on parallel and distributed execution to address computational problems whose scale exceeds the practical capabilities of sequential systems. The Message Passing Interface (MPI) remains one of the principal programming models for distributed-memory HPC systems. Under the Single Program, Multiple Data (SPMD) model, multiple processes execute the same program while maintaining independent ad-

dress spaces and explicitly exchanging data through MPI communication primitives. This programming model provides substantial flexibility and scalability, but it also introduces software defects that do not have direct analogues in conventional sequential programs.

MPI program correctness depends on relationships among operations distributed across multiple processes. A communication operation that appears valid when considered locally may be incorrect when evaluated against the corresponding operation executed by another rank. Send and receive operations must agree on parameters such as message tags, communicators, data types, and message sizes; collective operations must be invoked consistently by participating processes; nonblocking communication requires correct request lifecycles; and dynamically created MPI resources must be released appropriately. Errors in these relationships can result in deadlocks, incorrect data transfers, resource leaks, nondeterministic failures, or behavior that changes with process count or execution order [43].

These characteristics make MPI defect detection particularly challenging. The behavior of a defective parallel program may depend on the scheduling and ordering of operations across multiple ranks, and failures may emerge only under particular execution conditions. Consequently, evaluating an MPI program requires reasoning not only about source-code syntax and local control flow, but also about communication semantics and the relationships among operations performed by different processes.

Large language models (LLMs) provide a potentially complementary approach to this problem. Modern LLMs trained on natural language and source code exhibit substantial capabilities in code comprehension, defect identification, explanation, and program transformation. Their ability to reason over source code offers the possibility of identifying defects that are difficult to express through fixed syntactic rules. However, this capability does not imply that a general-purpose LLM possesses sufficient knowledge to reliably distinguish correct from incorrect uses of a specialized

parallel programming interface.

The experiments presented in this chapter examine this limitation directly. The central hypothesis is that an LLM's MPI defect-detection capability can be substantially improved by explicitly grounding its reasoning in domain knowledge. Rather than modifying model parameters, the approach supplies the model with examples of known MPI defects, expert explanations of those defects, and structured reasoning procedures that direct attention toward communication and synchronization properties that are important for MPI correctness.

This chapter refers to these examples and explanations collectively as *bug references*. A bug reference contains defective MPI code together with information describing the nature and location of the defect and guidance concerning its correction. Bug references therefore expose information that may be poorly represented in the model's pretrained knowledge and convert expert MPI knowledge into contextual information that can be used during inference.

Three mechanisms for delivering or supplementing this knowledge are investigated. First, few-shot learning incorporates representative bug references directly into the prompt. Second, Chain-of-Thought (CoT) guidance structures the analysis around MPI-specific correctness checks, including communication matching, message sizes, synchronization, request lifecycles, and resource management. Third, Retrieval-Augmented Generation (RAG) retrieves semantically similar MPI programs from external collections containing different mixtures of correct and defective programs.

The study addresses four complementary questions. First, it investigates how accurately an LLM identifies MPI defects without domain-specific assistance. Second, it measures whether explicit bug references and structured MPI reasoning improve defect detection. Third, it evaluates whether the resulting domain grounding can support automatic repair of defective MPI programs. Finally, it examines whether the observed improvements are specific to ChatGPT or generalize to open and code-specialized LLMs.

The results provide an early foundation for the broader dissertation thesis developed in subsequent chapters. They demonstrate that the quality of an LLM's behavior can be changed substantially without parameter fine-tuning when relevant domain knowledge is expressed as natural-language and source-code context. In particular, the experiments show that supplying the model with explicit descriptions of defective behavior is substantially more effective than relying on its pretrained knowledge alone. This observation motivates the later development of persistent and automatically optimized domain knowledge in METAFORGE.

3.2 Methodology

3.2.1 Experimental Overview

The experimental methodology was designed to progressively introduce domain knowledge and measure its contribution to MPI defect detection and repair. The initial experiments establish the performance of an LLM when it receives only an MPI program and a request to determine whether that program contains defects. Subsequent configurations add explicit examples, structured reasoning, and retrieved context. The highest-performing detection configuration is then used as the basis for automatic program repair.

OpenAI's GPT-3.5 Turbo model served as the primary model during the initial study and is referred to throughout this chapter as ChatGPT, consistent with the terminology used in the published study. Because LLM generation is nondeterministic, each defect-detection configuration was evaluated through five independent trials. A new conversation was instantiated for each request so that results from one program did not influence subsequent classifications.

The experiments used the OpenAI API together with ChromaDB and LangChain for the retrieval experiments. To assess whether the proposed approach depended on a single commercial model,

the experimental protocol was subsequently repeated with Llama 2, Code Llama, and Qwen2.5-Coder. These models were executed locally using Ollama.

3.2.2 MPI Program Dataset

The experimental programs were drawn from the MPI Bugs Initiative dataset [43], which contains both correct and defective MPI programs covering a range of communication and resource-management behaviors. The evaluation dataset contained 241 programs. The ground-truth distribution represented 174 defective programs and 67 programs without the target defects.

An important preprocessing step was applied before presenting these programs to the LLMs. Programs in the original MPI Bugs Initiative dataset may contain comments that explicitly describe the injected defect. Such comments would provide the model with information that would not ordinarily be available when analyzing an unknown program and could therefore artificially inflate detection performance. All source-code comments were consequently removed before evaluation.

This preprocessing is important to the interpretation of the experiments. The LLM must infer whether the program contains a defect from the MPI operations and their relationships rather than from an annotation explicitly identifying the defect. At the same time, the ground-truth labels provided by the benchmark allow each response to be classified objectively.

The defect classes represented in the study include defects associated with point-to-point and collective communication, invalid or mismatched communication parameters, synchronization behavior, request lifecycles, concurrency, and resource management. These classes require different forms of reasoning. A resource leak may involve recognizing that a dynamically created MPI object was never released, whereas a communication deadlock may require relating operations occurring on different ranks and identifying a mismatch that prevents an expected message from

being received.

3.2.3 Evaluation Metrics

Defect detection is treated as a binary classification problem, and the confusion matrix terms are defined relative to whether a real defect exists in software code and the analysis reports a defect. A *true positive* (TP) occurs when an actual defect is reported, whereas a *true negative* (TN) occurs when a defect-free program is correctly classified as such. A *false positive* (FP) denotes a valid program incorrectly identified as defective, and a *false negative* (FN) represents an actual defect that the model fails to identify. Table 3.1 depicts these terms, definitions, and the interpretations used on our work.

Table 3.1: Confusion Matrix of Core Definitions in Software Defect Detection

Term	**Definition**	**Interpretation**
True Positive (TP)	The analysis reports a defect and the defect is truly present in the code	Correctly detected software defect
True Negative (TN)	The analysis reports no defect and the code is truly defect-free	Correctly ignored non-defective code
False Positive (FP)	The analysis reports a defect when no defect exists in the code	Spurious warning or false alarm
False Negative (FN)	The analysis reports no defect when a defect is actually present	Missed software defect

The evaluation reports accuracy, precision, recall, and F1 score. Of particular importance is recall because false negatives correspond to defects that escape detection. For a software-analysis system, a high rate of false negatives is problematic because the absence of a warning may incorrectly suggest that defective software is safe. Precision remains important because excessive false posi-

tives increase the amount of developer effort required to inspect warnings that do not correspond to actual defects.

The use of repeated trials also permits analysis of output stability. Variance in TP, TN, FP, and FN counts is used to assess whether a prompting strategy produces more consistent behavior across independent model executions.

3.2.4 Zero-Shot Baseline

The first experiment establishes the LLM's inherent MPI defect-detection capability. No defect examples, MPI-specific rules, or external context are provided. Each program is supplied using a minimal query asking whether the MPI program contains bugs.

Zero-Shot Prompt:

Does the following MPI program *[CODE]* contain bugs?

Each of the 241 programs was evaluated five times. This configuration provides the reference point against which the subsequent domain-grounding strategies are measured.

3.2.5 Domain-Informed Bug References

The principal intervention investigated in this study is the explicit introduction of MPI bug knowledge. A collection of representative defect examples was constructed to expose the LLM to common failure mechanisms before asking it to classify an unseen program.

The bug references cover resource leaks, incorrect MPI operation ordering, invalid communication

parameters, point-to-point communication defects, collective communication defects, synchronization defects, and deadlock-producing communication mismatches. Each example associates source code with an explanation identifying the defect and its location and describing how the defect should be corrected.

This representation is intentionally different from simply providing additional source code. The explanatory component labels the relevant behavior and makes the relationship between the MPI operations and the defect explicit. The resulting context therefore supplies both an example and a compact representation of domain expertise.

3.2.6 Few-Shot Bug Referencing

The first use of the bug repository incorporates examples directly into the prompt through Few-Shot Learning. Nine representative examples are supplied before the target program. Each example contains a defective program and an accompanying explanation.

Conceptually, the prompt takes the following form:

Few-Shot Bug-Reference Prompt:

Here are examples of MPI programs and explanations of their defects.

[EXAMPLE1] [EXPLANATION1]

[EXAMPLE2] [EXPLANATION2]

...

[EXAMPLE9] [EXPLANATION9]

Does the following MPI program *[CODE]* contain bugs? Respond with your verdict.

This configuration tests whether representative defective programs and their explanations are sufficient to improve the model's classification of previously unseen MPI code without changing the underlying model.

3.2.7 MPI-Guided Chain-of-Thought Reasoning

Few-Shot examples provide knowledge of representative defects but do not guarantee that the model systematically examines the relationships responsible for those defects. The next experimental condition therefore combines the bug references with domain-informed CoT guidance [101].

Rather than relying on an unconstrained request to "reason step-by-step," the reasoning procedure encodes MPI-specific checks. Nine reasoning prompts were developed to guide analysis of point-to-point communication, collective operations, send/receive message sizes, resource management, message races, communication tags, missing start operations, incomplete scatter/gather patterns, and missing wait operations.

A condensed representation of the reasoning procedure is shown below.

Bug References + MPI-Guided Reasoning:

Analyze the MPI program step-by-step. In particular, determine whether:

1. point-to-point and collective communication operations are correctly matched;

2. send and receive message sizes are compatible;

3. message races or communication-tag mismatches are present;

4. dynamically allocated MPI resources are released; and

5. required start and wait operations are present and correctly ordered.

For each applicable condition, analyze the program and provide a final defect verdict.

Each of the 241 programs was evaluated using this combined Few-Shot+CoT strategy across five independent trials. The distinction between this condition and conventional Few-Shot prompting is important to the dissertation: the examples provide *domain knowledge*, whereas the MPI-specific reasoning instructions provide a reusable *domain reasoning procedure*.

3.2.8 Retrieval-Augmented MPI Context

The study also investigates whether dynamically retrieved MPI examples can improve upon fixed bug references. RAG was implemented using ChromaDB and a corpus of correct and defective MPI source code. Five retrieval databases were created to vary the proportion of correct and defective programs presented to the model.

Table 3.2 summarizes these configurations. The nomenclature denotes the percentage of the retrieval corpus composed of known correct programs.

For every target program, semantically related source-code examples were retrieved and inserted into the model context together with the engineered detection prompt. This process produced a separate retrieval-augmented query for each of the 241 target programs. The experiment therefore

Table 3.2: Nomenclature and description of retrieval databases used in RAG studies.

Nomenclature	Description
RAG_100%	Corpus containing only known good MPI source code
RAG_75%	Corpus containing 75% defect free MPI source code and 25% MPI source code known to contain at least one defect
RAG_50%	Corpus containing 50% defect-free MPI source code and 50% MPI source code known to contain at least one defect
RAG_25%	Corpus containing 75% MPI source code with at least one known defect and 25% known defect-free MPI source code
RAG_0%	Corpus containing only MPI source code with each document containing at least one known defect

evaluates not only whether retrieval improves MPI analysis, but also whether the composition of the retrieved knowledge affects the result.

3.2.9 Automatic MPI Program Repair

The highest-performing defect-detection configuration, Few-Shot bug references combined with MPI-specific CoT guidance, was subsequently adapted for program repair. The repair benchmark was formed from defective programs that ChatGPT correctly identified in all five Few-Shot+CoT detection trials. Requiring consistent detection reduced the likelihood of evaluating repair on programs for which the model did not reliably recognize the original defect.

This procedure produced a repair dataset containing 164 MPI programs. The experiment is therefore best interpreted as measuring repair capability *conditioned on reliable prior defect detection*; it is not an unconditional repair rate over the complete 241-program detection dataset.

The repair prompt preserved the bug-reference and reasoning context and instructed the model to return a corrected MPI C program:

MPI Repair Prompt:

[Few-Shot Bug References] + [MPI-Guided Reasoning] + [CODE]

You are an expert in Message Passing Interface (MPI) parallel programming. The MPI program above contains errors. Repair the program and return only the repaired MPI C source code.

The generated programs were evaluated on an MPICH cluster consisting of four compute nodes, each containing 28 CPU cores and 512 GB of memory and running Red Hat Enterprise Linux.

Because manually validating 164 generated programs would introduce both substantial effort and opportunities for inconsistent judgment, the repair evaluation was automated. Each generated program was first compiled. Executable programs were then evaluated for deadlock behavior and resource leakage. A 900-second alarm was injected into each execution so that programs that failed to terminate within 15 minutes could be classified as deadlocked. Resource leaks were evaluated using *Valgrind* with `-leak-check=yes` under *mpirun*. Leaks attributable to the MPI implementation itself were excluded so that the measurement reflected resources leaked by the generated application.

A repair was considered successful only when the generated program satisfied the required compilation and defect-removal criteria. This execution-based validation is important because a plausible textual repair is not sufficient evidence that the underlying MPI defect has been corrected.

3.2.10 Cross-Model Generalization

To determine whether the benefits of bug referencing were specific to ChatGPT, the detection protocol was replicated using Llama 2, Code Llama, and Qwen2.5-Coder. Llama 2 provides an

open general-purpose contrast to ChatGPT, while Code Llama and Qwen2.5-Coder provide models developed specifically for source-code tasks [89, 81, 36].

More specifically, we selected Llama 2, Code Llama, and Qwen2.5-Coder to evaluate the generality of our approach across both general-purpose and code-specialized models from distinct model families. Llama 2 provides a general-purpose contrast to ChatGPT, whereas Code Llama and Qwen2.5-Coder provide code-specialized contrasts designed for programming and source-code reasoning tasks [89, 81, 36]. This selection allows us to test whether the benefits of domain-grounded prompting persist across different model families and model specializations rather than depending on the behavior of a single LLM.

The models were executed using Ollama on Ubuntu 22.04 with two NVIDIA 4070 Super GPUs. Default model configurations were retained. The same progression of Zero-Shot, Few-Shot, Few-Shot+CoT, and retrieval-augmented conditions was applied so that the effect of domain grounding could be examined independently of the specific model used.

3.3 Evaluation

3.3.1 Baseline MPI Defect Detection

The Zero-Shot experiment reveals a substantial limitation in out-of-the-box MPI reasoning. Across five trials, ChatGPT achieved an average accuracy of only 44.39%, with a recall of 34.82% and an F1 score of 47.46%. More importantly, the model missed an average of 113.4 defective programs in each 241-program trial.

This high false-negative rate indicates that general code-generation capability does not translate directly into reliable MPI defect analysis. The model frequently accepted locally plausible code

without identifying errors that depended on MPI-specific call ordering, resource management, or cross-process communication relationships.

The result motivates the central claim of this chapter: specialized parallel-programming knowledge must be made sufficiently explicit in the model context before the model can be expected to apply that knowledge consistently.

3.3.2 Effect of Bug References and Structured Reasoning

Table 3.3 summarizes the principal ChatGPT defect-detection results. Few-Shot bug references alone increased accuracy from 44.39% to 56.34% and nearly doubled recall, from 34.82% to 70.91%. The larger improvement occurred when the bug references were combined with MPI-specific CoT guidance.

Table 3.3: Classification performance across ChatGPT prompting strategies. RAG conditions vary the composition of the retrieval corpus. Bold indicates the best value for each metric across the evaluated configurations.

Prompting Tech	**TP**	**TN**	**FP**	**FN**	**Acc(%)**	**Prec(%)**	**Rec(%)**	**F1(%)**
Zero-Shot	60.6	**46.4**	**20.6**	113.4	44.39	74.60	34.82	47.46
Few-Shot	121.8	14.0	55.2	50.0	56.34	68.82	70.91	69.83
Few-Shot+CoT	**170.2**	17.4	49.6	**3.8**	**77.84**	77.43	**97.81**	**86.43**
Few-Shot+CoT +RAG_100%	115.8	36.0	30.6	58.6	62.98	**79.07**	66.39	72.11
Few-Shot+CoT +RAG_75%	103.6	37.4	29.6	70.4	58.50	77.77	59.54	67.39
Few-Shot+CoT +RAG_50%	101.2	36.4	30.6	72.8	57.09	76.77	58.16	66.17
Few-Shot+CoT +RAG_25%	105.6	36.6	30.4	68.4	59.00	77.70	60.68	68.05
Few-Shot+CoT +RAG_0%	101	36.8	30.2	73	57.17	77.08	58.04	66.13

Few-Shot+CoT increased accuracy by 33.45 percentage points relative to Zero-Shot and increased recall from 34.82% to 97.81%. The average number of false negatives decreased from 113.4 to only 3.8 programs per trial. The F1 score increased correspondingly from 47.46% to 86.43%.

Figure 3.1 visualizes the progression from Zero-Shot through the domain-grounded configurations.

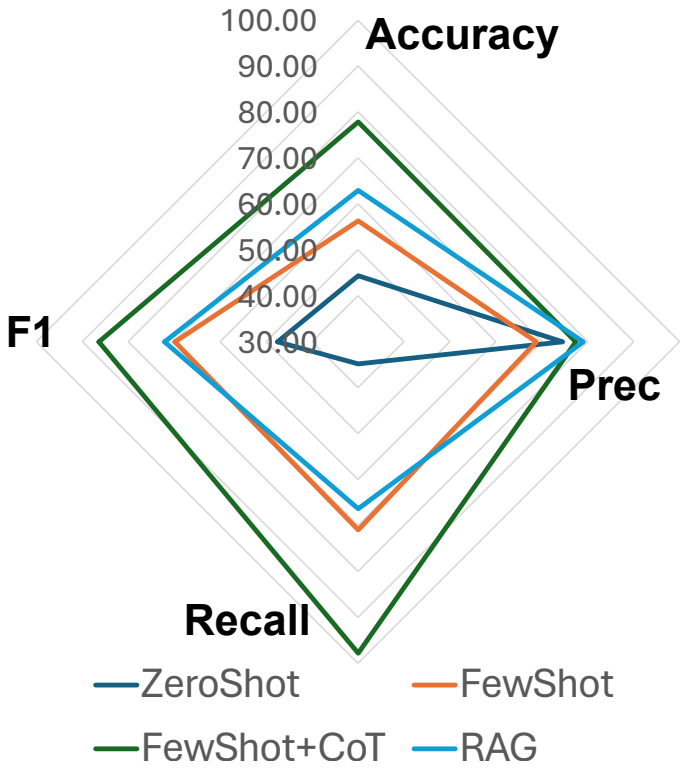


Figure 3.1: Comparison of Zero-Shot, Few-Shot, Few-Shot+CoT, and retrieval-augmented prompting for ChatGPT. The strongest detection performance is obtained when explicit MPI bug references are combined with structured MPI reasoning.

The improvement is not uniform across all components of the confusion matrix. The substantial reduction in missed defects is accompanied by an increase in false positives relative to Zero-Shot. This reflects a more conservative classification strategy in which the model is substantially more likely to identify an actual defect but is also more likely to question correct code. In the context of defect detection, this tradeoff may be preferable to the Zero-Shot operating point because false

positives can be subjected to secondary analysis, whereas an undetected defect receives no further attention.

Statistical analysis supports the conclusion that this difference is not attributable merely to normal model variability. Across the five ChatGPT trials, true positives increased from an aggregate 303 under Zero-Shot prompting to 851 under Few-Shot+CoT, while false negatives decreased from 567 to 19. McNemar's test produced $\chi^2 \approx 233.2$ with $p < 0.001$, demonstrating a statistically significant change in classification behavior.

3.3.3 Stability Across Independent Trials

In addition to improving average performance, the domain-grounded strategy produced substantially more stable classifications across repeated executions. Table 3.4 reports the variance of confusion-matrix counts across the five ChatGPT trials.

Table 3.4: Variance (σ^2) of confusion-matrix counts across five ChatGPT trials. Lower variance indicates greater stability; bold marks the lowest variance for each count.

Prompting Tech	**TP** σ^2	**TN** σ^2	**FP** σ^2	**FN** σ^2
Zero-Shot	17.84	9.04	9.04	17.84
Few-Shot	3.76	4.8	8.96	6.4
Few-Shot+CoT	**3.76**	**2.24**	**2.24**	**3.76**
Few-Shot+CoT+RAG_100%	62.56	4.4	3.44	60.64
Few-Shot+CoT+RAG_75%	33.04	3.44	3.44	33.04
Few-Shot+CoT+RAG_50%	12.56	5.84	5.84	12.56
Few-Shot+CoT+RAG_25%	52.64	20.3	20.3	65.8
Few-Shot+CoT+RAG_0%	62.0	45.76	45.76	61.9

Our results in Table 3.4 show that Few-Shot+CoT exhibits low variance across TP, TN, FP, and FN counts. This result is important because reliability in an LLM-based analysis system depends not only on obtaining a strong mean result but also on reducing sensitivity to repeated sampling.

Explicit domain knowledge and structured reasoning appear to constrain the model toward a more consistent analysis process.

3.3.4 Retrieval-Augmented Generation

The RAG experiments produce a different result. Retrieval improves upon the Zero-Shot baseline, but none of the five retrieval configurations matches the Few-Shot+CoT strategy. The strongest RAG configuration, RAG_100%, achieves 62.98% accuracy and 72.11% F1, compared with 77.84% accuracy and 86.43% F1 for Few-Shot+CoT.

Figure 3.2 illustrates the performance differences among the retrieval-corpus compositions.

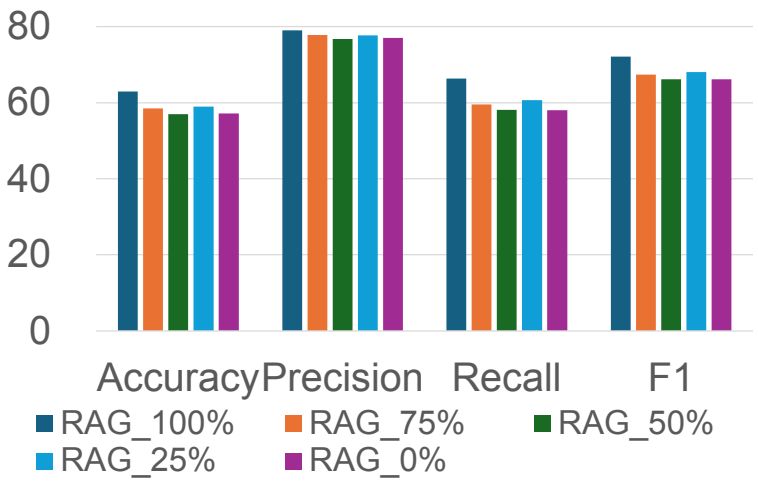


Figure 3.2: ChatGPT classification performance across retrieval corpora containing different proportions of correct and defective MPI programs.

The limited improvement cannot be explained simply by a failure to retrieve lexically related code. As shown in Table 3.5, the ChromaDB distances occupy a narrow range from 0.119176 to 0.129195. The retrieval mechanism therefore consistently identified examples that were close to the target under the embedding representation.

Table 3.5: Mean ChromaDB embedding-space retrieval distance for each retrieval-corpus composition. Lower values indicate closer proximity under the embedding representation.

Retrieval Database	**Retrieval Score**
RAG_100%	0.119315
RAG_75%	0.119244
RAG_50%	0.119176
RAG_25%	0.123624
RAG_0%	0.129195

These results expose an important distinction between lexical or embedding similarity and semantic similarity for parallel programs. Two MPI programs can contain nearly identical communication routines while differing in a single message tag, communicator, operation ordering, or rank-dependent branch that determines whether execution is correct. Conversely, programs implementing the same communication behavior may appear textually dissimilar.

The composition of the retrieval corpus also matters because a defective source-code example does not necessarily contain any textual indication that it is defective. When such code is retrieved without an accompanying defect annotation, its presence may reinforce an incorrect pattern rather than warn the model against it. This helps explain why retrieving additional source code is less effective than explicitly labeled bug references.

These RAG results therefore strengthen, rather than weaken, this chapter's broader domain-grounding argument. More context does not necessarily produce better reasoning. The *representation* of domain knowledge matters. Explicit defect examples and explanations encode the semantics of why an MPI pattern is incorrect, whereas generic source-code embeddings primarily encode textual similarity. This result motivates future MPI-aware retrieval representations that incorporate communication structure, rank relationships, call ordering, tags, communicators, and other domain-specific features.

3.3.5 Comparison with Conventional MPI Defect Detectors

The Few-Shot+CoT approach was also compared with four established non-LLM MPI analysis tools: CIVL, ISP, MPI-SV, and PARCOACH. These systems employ different combinations of symbolic execution, model checking, static analysis, and dynamic analysis.

Table 3.6 compares their performance on the same dataset with the domain-grounded ChatGPT configuration.

Table 3.6: MPI defect-detection performance of conventional analysis tools on the evaluation dataset.

Tool	**TP**	**TN**	**FP**	**FN**	**Acc(%)**	**Prec(%)**	**Rec(%)**	**F1(%)**
CIVL[53]	61	18	61	101	32.78	37.65	50.00	42.96
ISP[92]	116	17	94	14	55.19	89.23	55.24	68.24
MPISV[14]	68	15	64	94	34.44	41.98	51.52	46.26
PARCOACH[62]	140	10	79	12	62.24	92.11	63.93	75.47

The most consequential difference is the reduction in missed defects. Our bug-referenced Few-Shot+CoT approach produces a miss rate of approximately 2.2%, whereas the conventional tools exhibit substantially higher miss rates on this deliberately difficult MPI dataset. Its recall of 97.81% is also substantially higher than that of the strongest conventional comparator.

These results should not be interpreted as evidence that LLMs replace deterministic MPI analysis tools. Static analyzers, symbolic methods, and model checkers provide properties and guarantees that a generative model does not. Rather, the results demonstrate that domain-grounded LLM reasoning can identify classes of defects that are difficult for fixed analysis techniques and may therefore serve as a complementary analysis mechanism.

3.3.6 Automatic Program Repair

The repair experiments evaluate whether the same domain knowledge that improves defect recognition can support program transformation. Of the 164 programs in the ChatGPT repair benchmark, 152 generated repairs compiled successfully, yielding a compilation rate of 92.68%. The 12 compilation failures demonstrate that repair can introduce new defects even when the original problem is correctly identified. Five failures contained duplicate variable declarations, four contained basic syntax errors, and three passed incorrect argument types to MPI routines.

Resource-leak repair was evaluated on the 13 programs in the repair dataset containing known application-level resource leaks. After excluding leaks originating within the MPI implementation itself, 11 of the 13 target programs had their resource leaks removed. Deadlock removal was more difficult: the detailed repair evaluation reports successful removal in 76.83% of the applicable cases.

Using the study's aggregate success definition, 125 of the 164 generated programs satisfied the repair criteria, corresponding to an overall successful repair rate of 76.22%.

Figure 3.3 summarizes successful and unsuccessful outcomes across the repair metrics.

The distribution of failures is particularly informative. Parameter-matching defects account for many unsuccessful repairs. These errors can manifest as deadlocks when, for example, one process sends a message using one tag while another process waits for the corresponding message using a different tag. Repairing such a defect requires more than knowledge of C syntax or individual MPI function signatures. The model must infer the intended communication relationship between processes and determine which of the conflicting values reflects that intent.

This distinction helps explain why resource-management defects are generally easier for an LLM

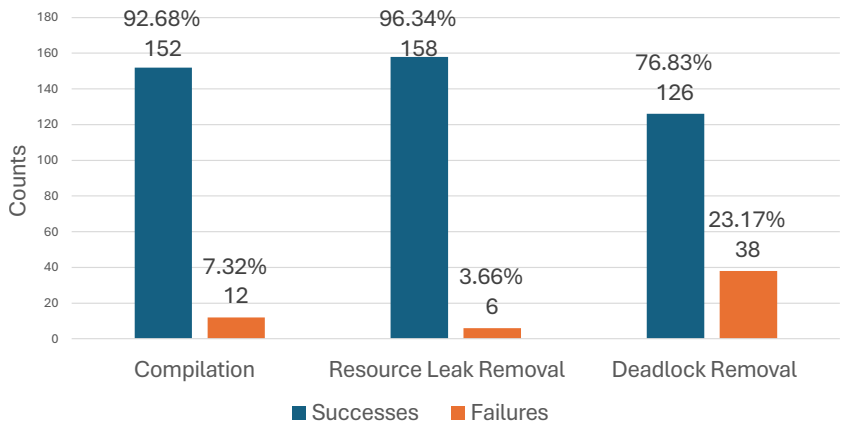


Figure 3.3: ChatGPT repair outcomes for successful compilation, resource-leak removal, and deadlock removal using the domain-grounded Few-Shot+CoT repair strategy.

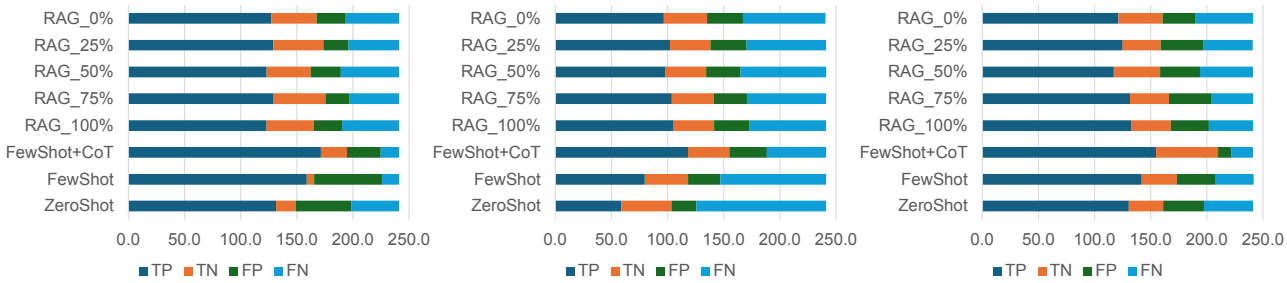
to repair than some deadlock-producing communication defects. Resource lifecycles often have recognizable local patterns: an MPI object is created and must later be freed. In contrast, communication defects can require relational reasoning across ranks, operations, and control-flow paths. The repair experiment therefore identifies *program intent* as an important missing component for future domain-grounded repair systems.

3.3.7 Generalization Across Large Language Models

The external-validity experiments demonstrate that the benefits of bug referencing are not confined to ChatGPT. The same overall pattern appears when the methodology is applied to Llama 2, Code Llama, and Qwen2.5-Coder.

Llama 2 increased from 61.83% Zero-Shot accuracy to 80.75% with Few-Shot+CoT. Code Llama increased from 42.90% to 64.32%. Qwen2.5-Coder produced the strongest absolute result, increasing from approximately 66% Zero-Shot accuracy to 87.05% under the domain-grounded configuration.

Figure 3.4 compares the underlying TP, TN, FP, and FN behavior across the three additional models.



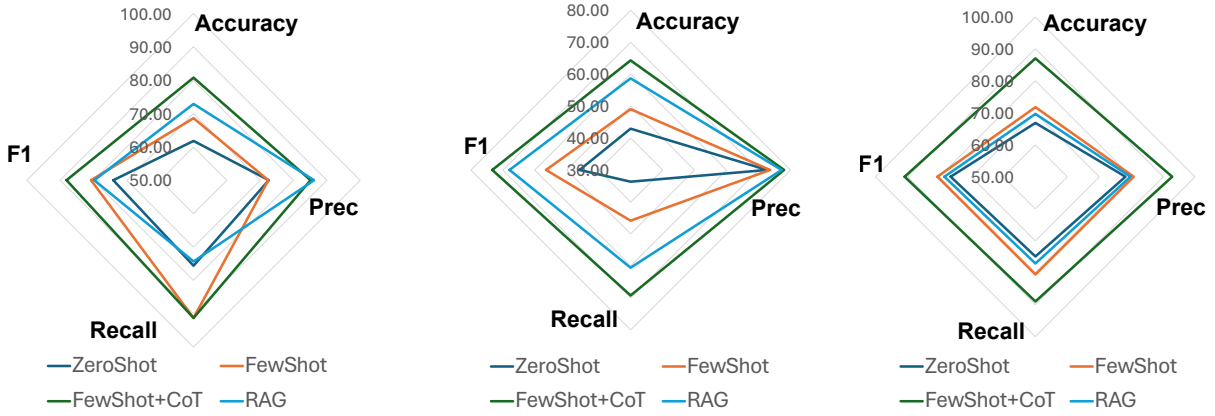
(a) Llama 2 (b) Code Llama (c) Qwen2.5-Coder

Figure 3.4: Confusion-matrix outcomes across prompting strategies for Llama 2, Code Llama, and Qwen2.5-Coder. Few-Shot bug references combined with MPI-specific CoT reasoning produce the largest increase in true positives and the strongest reduction in false negatives across all three models.

The corresponding classification metrics are shown in Figure 3.5. Although the magnitude of improvement varies by model, Few-Shot+CoT provides the strongest overall detection profile in each case.

McNemar's tests provide additional evidence that the changes observed between Zero-Shot and bug-referenced Few-Shot+CoT are statistically meaningful. Table 3.7 consolidates the results across all four models evaluated in the study.

The repeated effect across substantially different models is important to the dissertation argument. The contribution is not a new model and does not depend on modifying model weights. Instead, the intervention changes the information and reasoning structure available to the model at inference time. The fact that general-purpose and code-specialized models respond similarly suggests that explicit domain knowledge can improve MPI reasoning independently of the underlying LLM architecture.



(a) Llama 2 (b) Code Llama (c) Qwen2.5-Coder

Figure 3.5: Accuracy, precision, recall, and F1 across prompting strategies for the three additional LLMs.

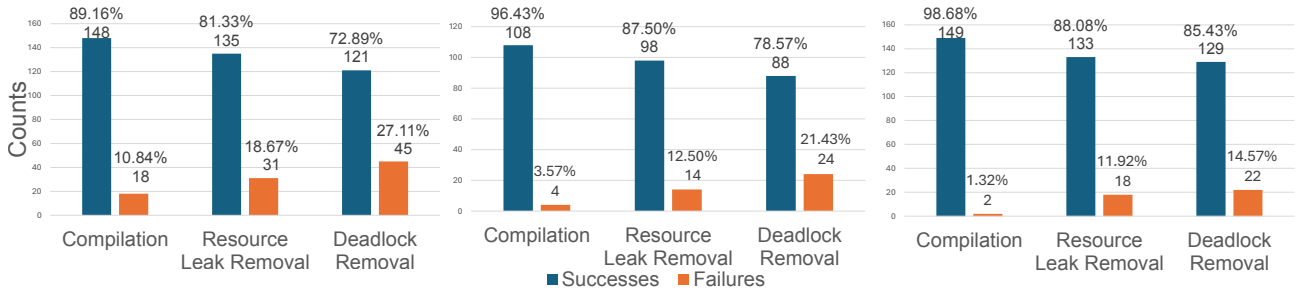
Table 3.7: McNemar tests comparing Zero-Shot detection with the Few-Shot+CoT bug-referencing approach across the four evaluated LLMs.

Model	χ^2	p
ChatGPT (GPT-3.5 Turbo)	≈ 233.2	< 0.001
Llama 2	≈ 64.1	< 0.001
Code Llama	≈ 164.1	< 0.001
Qwen2.5-Coder	≈ 181.2	< 0.001

3.3.8 Cross-Model Repair Behavior

The repair experiments were also repeated with Llama 2, Code Llama, and Qwen2.5-Coder using model-specific repair datasets constructed according to the same consistent-detection criterion used for ChatGPT. Similar repair patterns emerged across models, providing additional evidence that the usefulness of the bug-reference representation extends beyond defect classification.

Figure 3.6 compares repair success and failure behavior for the three additional models.



(a) Llama 2 (b) Code Llama (c) Qwen2.5-Coder

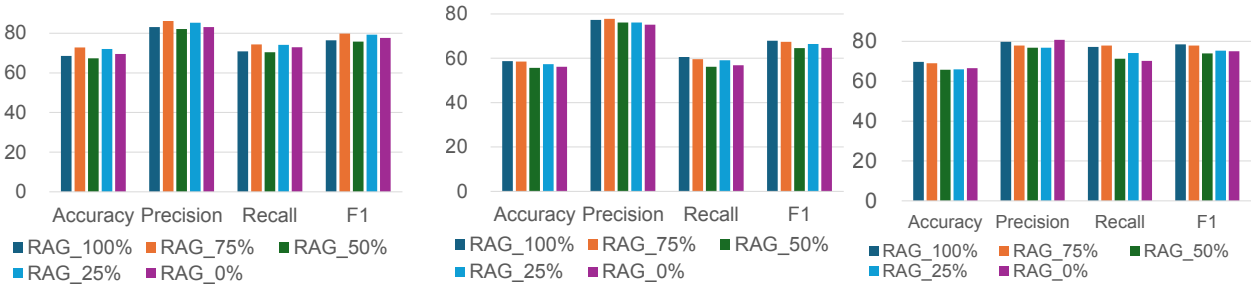
Figure 3.6: Repair success and failure rates across Llama 2, Code Llama, and Qwen2.5-Coder.

Qwen2.5-Coder produced the strongest repair behavior among these models, including the highest successful-compilation rate and strong performance in resource-leak and deadlock removal. This result is consistent with its stronger overall code-oriented behavior in the defect-detection experiments. Nevertheless, the recurrence of similar repair patterns across the models indicates that the domain-grounding strategy is not tied exclusively to the capabilities of one LLM.

3.3.9 Cross-Model RAG Behavior

The RAG limitation observed with ChatGPT also generalizes across the three additional models. Adding retrieved MPI source code generally failed to improve upon Few-Shot+CoT and frequently degraded performance. Figure 3.7 illustrates the result across retrieval-corpus compositions.

The consistency of this result across models suggests that the limitation resides primarily in the representation and retrieval strategy rather than in a single LLM. Conventional text embeddings identify lexical or token-level similarity but do not explicitly encode the process relationships that define MPI communication semantics. This finding provides a direct motivation for later work on domain-aware representations in which retrieval is based on properties such as MPI operations,



(a) Llama 2 (b) Code Llama (c) Qwen2.5-Coder

Figure 3.7: Accuracy, Precision, Recall, and F1 Score by Retrieval Corpus Composition using Llama 2, Code Llama, and Qwen2.5-Coder, showing repeated trends for each retrieval corpus by LLM.

communicators, message parameters, synchronization patterns, request lifecycles, and cross-rank dependencies.

3.3.10 Discussion

Taken together, the experiments demonstrate that domain grounding can substantially change the reliability of LLM reasoning about parallel software. The Zero-Shot baseline shows that pretrained model knowledge alone is insufficient for dependable MPI defect detection. Providing labeled bug references produces a substantial improvement, demonstrating that explicit examples of defective behavior compensate for weaknesses in the model's latent knowledge. Adding MPI-specific reasoning guidance produces the largest improvement, indicating that domain knowledge is most useful when it is paired with a procedure that directs the model toward the relationships that determine correctness.

The distinction between the Few-Shot+CoT and RAG results is particularly important. Both approaches increase the amount of MPI-related context available to the model, but only the former

consistently produces strong detection performance. The difference is that bug references are *semantically labeled*: they explain what is wrong and why. Retrieved source code may be similar at the token level while providing no indication of whether the represented communication pattern is valid. These experiments therefore show that context quantity and context relevance are not sufficient measures of grounding quality. The knowledge must represent the properties that matter to the target reasoning task.

The repair results expose a complementary limitation. Once the defect is reliably recognized, LLMs can correct a substantial proportion of defective MPI programs. However, repair becomes more difficult when correctness depends on inferring program intent. Communication mismatches are representative of this problem: recognizing that two tag values disagree establishes that a defect exists, but selecting the correct value requires understanding which communication behavior the programmer intended. Future repair systems should therefore integrate task descriptions, specifications, assertions, or other intent-bearing information with the source code and defect knowledge.

These findings also establish a conceptual bridge to the subsequent METAFORGE work in this dissertation. The present chapter demonstrates that externally supplied natural-language domain knowledge can improve an LLM's software-engineering behavior without modifying model parameters. However, the bug references and reasoning instructions used here are constructed manually by domain experts. METAFORGE extends this idea by investigating whether reusable natural-language skills can instead be refined automatically from compiler diagnostics, static-analysis findings, correctness tests, runtime measurements, and load-imbalance profiles. In this progression, the work presented in this chapter establishes the value of explicit domain grounding, while the later work investigates how such grounding can be learned, retained, and transferred automatically.

The results should nevertheless be interpreted within several limitations. LLM outputs remain nondeterministic, which motivated the use of five independent detection trials. The possibility

of benchmark contamination cannot be completely excluded because the training corpus for the commercial model is unavailable. The experiments also focus on a specific collection of MPI C programs and therefore do not establish that the same absolute performance will be obtained for substantially larger production applications or other parallel programming paradigms. Finally, model versions change over time, creating challenges for exact long-term reproduction.

Despite these limitations, the central finding is consistent across the experimental conditions and across multiple LLMs: explicit MPI defect knowledge combined with structured MPI reasoning substantially improves the detection of difficult parallel-programming defects. The work therefore demonstrates that domain grounding is a viable mechanism for improving LLM reliability in specialized software-engineering domains and provides the first major empirical foundation for the dissertation's broader investigation of domain-grounded LLMs for reliable and efficient parallel software engineering.

.

Chapter 4: A Meta-Learning Framework for Harness Optimization in Automated Code Generation for Parallel Computing

This chapter presents METAFORGE, an execution-grounded meta-learning framework for improving automated Message Passing Interface (MPI) C code generation by optimizing the persistent natural-language skill specification that guides a large language model (LLM). Rather than treating each generated program as an isolated artifact, METAFORGE repeatedly evaluates generated code using correctness tests, compiler diagnostics, static-analysis findings, runtime measurements, and rank-level load-imbalance profiles and uses this evidence to refine the instructions that condition subsequent generations. The framework further maintains an external memory that distills reward-supported optimization experience and makes that experience available to later iterations and tasks. The study therefore evaluated whether reusable, language-encoded optimization knowledge could improve the reliability and efficiency of parallel-code generation without modifying the parameters of the underlying language model.

4.1 Introduction

Large language models have become increasingly capable code-generation systems, yet producing a syntactically plausible program is substantially different from producing software that is simultaneously correct, efficient, and appropriate for a particular execution environment. Natural-language requirements may omit assumptions, leave performance objectives implicit, or permit several reasonable interpretations. These ambiguities can materially affect generated software, particularly when a task requires the model to translate high-level intent into implementation decisions rather than merely complete a local code fragment [115, 108, 22].

This difficulty is amplified in high-performance parallel computing. MPI programs distribute both computation and data across multiple processes, and their correctness depends on relationships among communication operations, collectives, synchronization points, data decomposition, and memory layout. A program can compile and produce the expected output while still exhibiting substantial communication overhead, poor decomposition, root-process bottlenecks, or rank-level imbalance. Conversely, an apparently attractive optimization can introduce deadlock, invalidate communication ordering, or alter numerical results. These characteristics make parallel-code generation an execution-grounded optimization problem in which program behavior must be measured rather than inferred solely from source-code plausibility [49, 14, 29, 35, 19, 48, 113, 93].

A second limitation of one-shot code generation is that useful optimization experience is typically lost when the task ends. An LLM may receive compiler feedback, static-analysis diagnostics, or runtime measurements during one interaction, but those observations do not automatically become a persistent, inspectable policy for future generations. METAFORGE addresses this problem by shifting the target of optimization from the generated program alone to a persistent *skill specification*: a structured natural-language artifact that describes how the code-generation model should approach MPI programming and optimization.

For each candidate, the framework compiles and tests the generated MPI C program, applies static analysis, executes the program, and records critical-path runtime and rank-level load imbalance. These heterogeneous signals are combined into a scalar reward used to compare candidates. METAFORGE then converts the measured deficiencies and reward trend into a *textual gradient*, a natural-language statement describing how the skill should change. A second refinement call applies that gradient while preserving instructions that are unrelated to the observed deficiency. The process is therefore grounded in externally measured program behavior rather than relying on the LLM to judge the quality of its own output.

The framework also separates broad and specific forms of accumulated knowledge. The evolving skill specification captures reusable generation and optimization behavior that can be carried from one task to another. In parallel, a persistent external memory stores detailed iteration evidence and distills reward-improving episodes into compact lessons indexed by MPI constructs, parallel patterns, task characteristics, and correctness or performance symptoms. Later iterations retrieve only the most relevant lessons. Because both the skill and memory remain external to the model, their contents can be inspected, transferred, and evaluated without parameter fine-tuning.

The central hypothesis evaluated in this chapter is that execution-grounded refinement of persistent coding skills can improve generated MPI software across multiple objectives—correctness, static-analysis quality, execution time, and load balance—while also retaining optimization knowledge that is useful beyond the task on which it was learned. The evaluation therefore compares METAFORGE with zero-shot generation using an empty skill and with a TEXTGRAD-based iterative-refinement baseline [114]. It further examines the contribution of retrieval memory, transfer to held-out tasks, zero-update skill transfer, cross-model behavior, and inference cost.

4.2 Methodology

4.2.1 Framework Overview

METAFORGE consists of two coupled optimization processes. The *within-run* process repeatedly generates, evaluates, and refines the skill specification for a fixed programming task. The *cross-run* process converts the resulting trajectory into persistent optimization knowledge that can be retrieved during later runs. The framework maintains a current skill $\mathcal{S}$, a best-observed skill $\mathcal{S}^*$, an episode store $\mathcal{E}$ containing iteration-level evidence, a distilled memory store $\mathcal{M}$, and an inverted retrieval index $\mathcal{I}$. This separation allows detailed execution traces to remain available for audit and

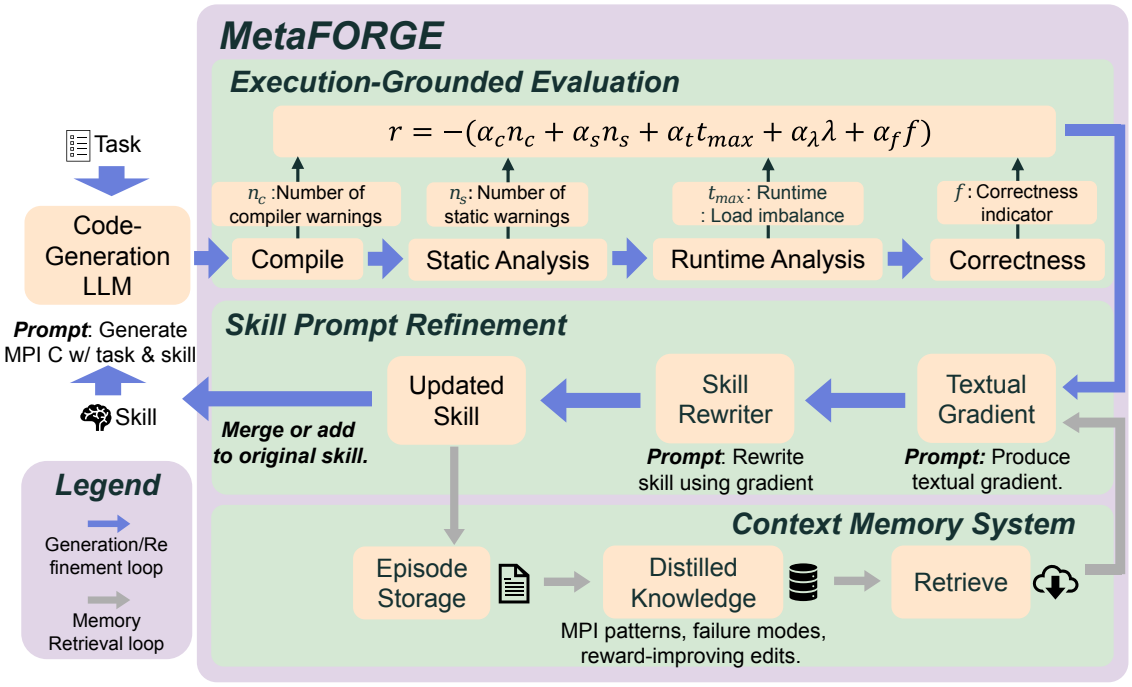


Figure 4.1: **MetaFORGE: An automated meta-learning skill-refinement architecture.** MetaFORGE optimizes LLM coding skills through execution-grounded feedback. Generated MPI C code is compiled, statically analyzed, executed, and scored using Eq. (4.1). Diagnostics and reward trends are converted into a textual gradient for skill-prompt rewriting, while iteration artifacts are stored, distilled, indexed, and retrieved as long-term memory, enabling future updates to reuse prior reward-improving strategies.

distillation while exposing only compact, reward-supported guidance to later model calls.

Figure 4.1 summarizes the architecture. A generated MPI program is evaluated by external tools before the next skill update is constructed. Consequently, the information that modifies the persistent skill comes from observed compiler, analyzer, correctness, runtime, and utilization behavior rather than from model confidence.

4.2.2 Notation and Framework State

Tables 4.1 and 4.2 summarize the notation used throughout the framework and its memory subsystem. The first table describes the state and measurements used by the optimization loop, whereas the second defines the structured tuple and memory terms used for cross-run retrieval.

Table 4.1: MetaFORGE notation.

Symbol	Description
$\mathcal{S}$	Skill specification
$\mathcal{T}$	Fixed task specification
C_i	MPI C program at iteration i
n_c	Compiler-warning count
n_s	Static-analysis-warning count
$\boldsymbol{t}$	Per-rank wall-clock times in $\mathbb{R}^K$
$t_{\max}$	Critical-path runtime, $\max_k t_k$
λ	Load imbalance, $(\max_k t_k - \min_k t_k)/\bar{t}$
r	Scalar reward; closer to zero is better
r^*	Best observed reward
∇_{text}	Textual gradient for skill rewriting
p	Non-improvement counter
P	No-improvement limit
ρ	Rewriter backend

4.2.3 Execution-Grounded Evaluation

Let $\mathcal{T}$ denote a fixed task specification, $\mathcal{S}$ the current skill, and K the number of MPI processes. For a generated program, let $\boldsymbol{t} = (t_0, \ldots, t_{K-1})$ denote the measured per-rank wall-clock times. Candidate quality is summarized using the reward

$$r = -\left(\alpha_c n_c + \alpha_s n_s + \alpha_t t_{\max} + \alpha_\lambda \lambda + \alpha_f f\right), \tag{4.1}$$

Table 4.2: Context-retrieval notation.

Symbol	Meaning
$\mathbf{q}$	Tuple bundle: $\mathbf{q}.mpi, \mathbf{q}.pat, \mathbf{q}.task, \mathbf{q}.perf, \mathbf{q}.corr$
q, m	Query tuple and stored memory entry
$\mathcal{E}, \mathcal{M}$	Episode and memory stores
$\mathcal{I}$	Inverted index from tags to memory IDs
$J(A, B)$	Jaccard similarity; $J(\emptyset, \emptyset) = 0$
Δr_m	Reward improvement for memory m
$\mathrm{age}(m)$	Age of memory m in days
Π	MPI-pattern keyword dictionary
Φ_s	Predicate for performance symptom s
$\mathcal{W}$	MPI task vocabulary
$\mathcal{V}$	Action-verb lexicon
$S_{\mathrm{perf}}, S_{\mathrm{corr}}$	Performance and correctness symptom sets
K_s	Diagnostic keywords for symptom s

where n_c is the number of compiler warnings, n_s is the number of static-analysis warnings, $t_{\max} = \max_k t_k$ is the critical-path runtime, and

$$\lambda = \frac{t_{\max} - t_{\min}}{\bar{t}}$$

measures normalized rank-level load imbalance. The binary variable f records task-specific correctness failure. A sufficiently large α_f prevents an incorrect program from being selected merely because it executes quickly or produces fewer warnings. The remaining positive weights control the relative contribution of code quality, runtime, and utilization. Because the reward is non-

positive, values closer to zero represent better candidates.

The scalar reward is used for selection and trend comparison rather than as a claim that program quality can be reduced to one universal number. Each component is retained separately so that the refinement mechanism can determine why a candidate improved or regressed. This distinction is important in parallel computing: reducing load imbalance does not necessarily reduce end-to-end runtime, and a faster program may still contain static quality problems or fail correctness tests.

4.2.4 Meta-Learning Control Flow

Algorithm 1 defines the complete METAFORGE refinement loop. Starting from an initial skill $\mathcal{S}_0$, each iteration generates an MPI C candidate and evaluates it with the common execution pipeline. The best reward and corresponding skill are retained. The current task, program, metrics, diagnostics, and previous textual gradient are transformed into a structured tuple bundle and stored as an episode. Unless the no-improvement threshold has been reached, the tuple bundle is used to retrieve relevant cross-run memories, and the retrieved context is incorporated into the skill-rewrite context. After the trajectory terminates, the recorded episodes are compressed into memory for later use.

The patience variable p records consecutive iterations that do not improve the best reward r^*. The threshold P therefore bounds unproductive refinement and prevents an additional rewrite after the stopping condition has been satisfied. The episode identifier list $\mathbf{ids}$ serves a different purpose: it preserves the trajectory needed to construct a cross-run memory after the refinement loop ends. Thus, stopping and memory construction are handled independently.

Algorithm 1 MetaFORGE Skill Meta-Learning Harness

Input: Task $\mathcal{T}$, initial skill $\mathcal{S}_0$, limits (N, P), process count K, retrieval budget k, stores $(\mathcal{E}, \mathcal{M}, \mathcal{I})$, weights $\boldsymbol{\alpha}$, and rewriter ρ
Output: Optimized skill $\mathcal{S}^*$

Initialize $\mathcal{S} \leftarrow \mathcal{S}_0$, $\mathcal{S}^* \leftarrow \mathcal{S}_0$, $r^* \leftarrow -\infty$, $p \leftarrow 0$, $\nabla_{\text{prev}} \leftarrow \varepsilon$, and $\mathbf{ids} \leftarrow [\,]$.
for $i = 1$ **to** N **do**
  $C_i \leftarrow \textsc{Generate}(\mathcal{S}, \mathcal{T})$ ▷ Prompt 4.2
  $(\mathbf{m}_i, \ell_i, r_i) \leftarrow \textsc{Evaluate}(C_i, K, \boldsymbol{\alpha})$, which compiles, tests, analyzes, executes, and scores C_i.
  If $r_i > r^*$, set $(\mathcal{S}^*, r^*, p) \leftarrow (\mathcal{S}, r_i, 0)$; otherwise set $p \leftarrow p + 1$.
  $\mathbf{q}_i \leftarrow \textsc{ExtractTuples}(\mathcal{T}, C_i, \mathbf{m}_i, \nabla_{\text{prev}}, \ell_i)$. *(Algorithm 3)*
  $\text{id}_i \leftarrow \mathcal{E}.\text{store}(\mathcal{S}, \mathcal{T}, C_i, \mathbf{q}_i, \mathbf{m}_i, r_i, r^*)$; append id_i to $\mathbf{ids}$.
  break if $p \geq P$
  $\boldsymbol{\xi}_i \leftarrow \textsc{FormatContextBlock}\big(\textsc{RetrieveContext}(\mathbf{q}_i, k)\big)$. *(Algorithm 6)*
  $\mathcal{X}_i \leftarrow [\mathcal{S}, \mathcal{T}, C_i, \mathbf{m}_i, \ell_i, r_i, r^*, \boldsymbol{\xi}_i]$.
  $(\mathcal{S}', \nabla_i) \leftarrow \textsc{RewriteViaGradient}(\mathcal{X}_i, \rho)$ ▷ Alg. 2
  $\mathcal{E}.\text{updateGradient}(\text{id}_i, \nabla_i)$; $\mathcal{S} \leftarrow \mathcal{S}'$; $\nabla_{\text{prev}} \leftarrow \nabla_i$.
end for
$\textsc{CompressandIndex}(\mathbf{ids})$ ▷ Section 4.2.6 and Algorithm 4
return $\mathcal{S}^*$

4.2.5 Reward-Guided Textual-Gradient Refinement

The numerical reward establishes whether a candidate improved, but it does not explain what should be changed in the natural-language skill. METAFORGE therefore separates diagnosis from modification, as shown in Algorithm 2. The first model call produces a textual gradient from the current skill, task, generated program, metric summary, diagnostics, reward comparison, and retrieved memory. The second call applies that textual gradient to the persistent skill. Retrieved memory therefore influences skill rewriting through the evidence-grounded textual gradient rather than being supplied independently to the rewrite prompt. If the diagnostic stage produces no actionable gradient, the skill is retained unchanged.

This two-stage structure makes the proposed update explicit before the persistent skill is modified. It also produces a reusable textual artifact: the gradient can be attached to the current episode and later distilled into cross-run memory. Retrieved lessons are advisory rather than authoritative: they are incorporated into the diagnostic context used to construct the textual gradient, while the current execution evidence and reward trend determine whether prior guidance is applicable to the present

Algorithm 2 Skill Rewriting via Textual Gradient: REWRITEVIAGRADIENT

Input: Refinement context $\mathcal{X} = [\mathcal{S}, \mathcal{T}, C, \mathbf{m}, \ell, r_i, r^*, \boldsymbol{\xi}]$ and rewriter ρ
Output: Updated skill $\mathcal{S}'$ and textual gradient ∇_{text}

Unpack $\mathcal{X}$ into the current skill $\mathcal{S}$, task $\mathcal{T}$, generated code C, metric summary $\mathbf{m}$, diagnostics ℓ, current and best rewards (r_i, r^*), and retrieved context $\boldsymbol{\xi}$.
Step 1: Compute textual gradient
$\nabla_{\text{text}} \leftarrow \text{TEXTGRADIENT}(\mathcal{X})$ ▷ Prompt 4.3
return $(\mathcal{S}, \nabla_{\text{text}})$ **if** $\nabla_{\text{text}} = \varepsilon$
Step 2: Apply textual gradient
$\mathcal{S}' \leftarrow \text{REWRITESKILL}(\mathcal{S}, \nabla_{\text{text}}, \boldsymbol{\xi}; \rho)$ ▷ Prompt 4.4
return $(\mathcal{S}', \nabla_{\text{text}})$

candidate.

4.2.6 Retrieval-Augmented Optimization Memory

The external memory system maintains two levels of persistence. Iteration-level episodes retain the full evidence required to reconstruct an optimization step, including the skill, task, generated source, measurements, diagnostics, reward, textual gradient, and structured tuple bundle. Cross-run memories are deliberately more compact. They retain distilled actionable lessons together with the tags and reward evidence that support those lessons.

Tuple extraction summarizes five complementary aspects of an iteration: MPI constructs, reusable parallel patterns, task characteristics, performance symptoms, and correctness symptoms. This representation is used as a deterministic retrieval key rather than embedding the entire source program or execution log. Algorithm 3 formalizes the extraction procedure.

After a run completes, COMPRESSANDINDEX converts its trajectory into a persistent memory record. Reward-improving episodes are preferred as evidence; if none exists, the complete trajectory is used so that memory construction remains defined. Structural and symptom tags are aggregated, and textual gradients from selected episodes are placed first in the distillation sequence. The resulting memory records the distilled guidance, reward improvement, and supporting episode

Algorithm 3 Tuple Extraction

Input: Source c, task $\mathcal{T}$, metrics $\mathbf{m}$, previous textual gradient g_{prev}, diagnostics ℓ
Output: Tuple bundle $\mathbf{q}$

$\mathbf{q}.mpi \leftarrow \{x \mid x$ matches `MPI_\w+` in $c \cup g_{\text{prev}}\}$
$\mathbf{q}.pat \leftarrow \{p \mid \exists kw \in \Pi(p) : kw \in \text{lower}(c \cup g)\}$
$\mathbf{q}.task \leftarrow \{w \in \mathcal{W} \mid w \in \text{lower}(\mathcal{T})\}$
$\mathbf{q}.perf \leftarrow \{s \in S_{\text{perf}} \mid \Phi_s(\mathbf{m}) = \top\}$
Identify additional performance symptoms in S_{perf} that are not already in $\mathbf{q}.perf$ and whose keywords appear in $\text{lower}(g)$.
Add the identified symptoms to $\mathbf{q}.perf$.
$\mathbf{q}.corr \leftarrow \{s \in S_{\text{corr}} \mid \exists kw \in K_s : kw \in \text{lower}(\ell \cup g_{\text{prev}})\}$
$\mathbf{q} \leftarrow \{\mathbf{q}.mpi, \mathbf{q}.pat, \mathbf{q}.task, \mathbf{q}.perf, \mathbf{q}.corr\}$
return $\mathbf{q}$

metadata.

Algorithm 4 Compress and Index

Input: Episode IDs $\mathbf{ids} = [\text{id}_1, \ldots, \text{id}_n]$
Output: Memory ID μ; return $\perp$ if empty

Exit and return $\perp$ if $\mathbf{ids}$ is empty.
$E \leftarrow [\mathcal{E}.\text{load}(\text{id}) \mid \text{id} \in \mathbf{ids}]$
$E^+ \leftarrow \{e \in E \mid e.\text{improved} = \top\}$
$E^{\text{sel}} \leftarrow \begin{cases} E^+, & E^+ \neq \emptyset, \\ E, & E^+ = \emptyset \end{cases}$
For each tuple category $c \in \{\text{mpi}, \text{pat}, \text{task}, \text{perf}, \text{corr}\}$, set $T_c \leftarrow \bigcup_{e \in E^{\text{sel}}} e.\mathbf{t}_c$
$G \leftarrow [e.g \mid e \in E^{\text{sel}}] +\!\!+ [e.g \mid e \in E \setminus E^{\text{sel}}]$
$(h, A, \ell) \leftarrow \text{DistillLesson}(G, E, E^{\text{sel}})$ ▷ Alg. 5
$\Delta r \leftarrow \max_{e \in E} e.r - E[0].r$
$\mu \leftarrow$ "mem_" $+ \text{timestamp}()$
$\mathcal{M}.\text{write}(\mu, \{\mu, T, h, A, \ell, \Delta r, |E^+|, |E|\})$
For each tuple category c and value $v \in T_c$, update $\mathcal{I}[c][v] \leftarrow \mathcal{I}[c][v] \cup \{\mu\}$
$\mathcal{I}.\text{flush}()$
return μ

Algorithm 5 converts the ordered textual gradients into a compact lesson. Action-oriented statements are deduplicated, the strongest reward-supported episode supplies the headline, and the knowledge-distillation prompt converts the retained actions and best metrics into concise recommendations.

At retrieval time, the inverted index first identifies memories that share at least one tuple value with the current query. Candidate memories are then ranked using structural similarity, performance

Algorithm 5 Lesson Distillation

Input: Gradients $G = [g_1, g_2, \ldots]$ with selected episodes first; all episodes E; selected episodes E^{sel}
Output: Headline h, actions A, lesson ℓ

seen $\leftarrow \emptyset$; $A \leftarrow [\,]$
$L \leftarrow \{\text{strip}(\lambda) \mid g \in G,\ \lambda \in \text{lines}(g)\}$
$A \leftarrow \text{UniqueFirst}_{A_{\max}}(\{\lambda \in L \mid |\lambda| \geq L_{\min} \wedge \exists v \in \mathcal{V} : v \in \text{lower}(\lambda)\})$
$e^* \leftarrow \arg\max_{e \in E^{\text{sel}}} e.r$
$h \leftarrow \text{firstSubstantiveLine}(e^*.g)$
if $h = \varepsilon$ **then**
    $h \leftarrow \begin{cases} A[0], & A \neq [\,], \\ \textit{(no lesson)}, & \text{otherwise} \end{cases}$
end if
$e_b \leftarrow \arg\max_{e \in E} e.r$
hdr $\leftarrow \langle$best metrics of $e_b\rangle$
$\ell \leftarrow$ SUMMARIZERECOMMENDATIONS(*hdr*, A) ▷ Prompt 4.5
return $h,\ A[0{:}A_{\text{ret}}],\ \ell$

symptoms, task terms, reward-supported confidence, and recency. Algorithm 6 performs candidate collection and ranking, while Table 4.3 reports the weighting used by WEIGHTEDSCORE.

Algorithm 6 Weighted Context Retrieval

Input: Query tags q, retrieval budget k
Output: Ranked memories $\mathbf{R}$

$\mathcal{C} \leftarrow \bigcup_{c,\, v \in q_c} \mathcal{I}[c][v]$ ▷ tag-overlap candidates
$\mathbf{R} \leftarrow [\,]$
$\mathbf{R} \leftarrow \{(m, \sigma) \mid \mu \in \mathcal{C},\ m = \mathcal{M}.\text{load}(\mu),\ \sigma = \text{WeightedScore}(q, m),\ \sigma > 0\}$
return $\text{TopK}(\mathbf{R}, k)$ sorted by descending σ

The weighting places the greatest emphasis on MPI constructs and parallel patterns because these terms most directly characterize implementation structure. Performance symptoms and task terms provide additional context, while reward improvement favors lessons supported by successful trajectories. A small recency component prevents old guidance from dominating otherwise similar memories.

Table 4.3: Context-retrieval scoring weights. Structural similarity receives the greatest weight, while reward improvement and recency provide secondary ranking signals.

Component	Weight	Score term
MPI construct	0.25	$J(q_{\mathrm{mpi}}, m_{\mathrm{mpi}})$
Parallel pattern	0.25	$J(q_{\mathrm{pat}}, m_{\mathrm{pat}})$
Performance symptom	0.20	$J(q_{\mathrm{perf}}, m_{\mathrm{perf}})$
Task keyword	0.15	$J(q_{\mathrm{task}}, m_{\mathrm{task}})$
Reward confidence	0.10	$\mathrm{clamp}(\Delta r_m/2, 0, 1)$
Recency	0.05	$e^{-\mathrm{age}(m)/30}$

4.2.7 Cross-Task Knowledge Persistence

Meta-learning is represented through two distinct persistent artifacts. The best skill learned during one optimization sequence can initialize a later task, transferring broadly applicable coding and optimization instructions. The memory store provides a second channel that retrieves more specific execution-derived strategies associated with similar MPI constructs, parallel patterns, or observed symptoms. Separating these mechanisms makes it possible to determine experimentally whether improvement is retained in the skill itself, in external memory, or in both.

4.2.8 Prompt Specifications

The framework uses four principal prompt templates. Retaining them in the proposal makes the refinement procedure explicit and reproducible rather than treating the LLM interaction as an unspecified implementation detail.

4.2.8.1 Code-Generation Prompt

The generation prompt keeps the task specification separate from the persistent skill so that the task states *what* should be produced and the skill describes *how* the model should approach implementation.

In retrieval-enabled runs, the formatted cross-task memory context is appended programmatically to the textual-gradient request and is therefore not shown as a fixed placeholder in the template below.

Code Generation Prompt

```
# Skill Specification
{Skill Text}
# Task
{Task Text}
Generate a complete MPI C program implementing the task per the skill specification.
```

Figure 4.2: MetaFORGE code-generation prompt. The task specification is kept separate from the persistent skill specification so that the task defines the required program while the skill supplies reusable generation guidance.

4.2.8.2 Textual-Gradient Prompt

The textual-gradient prompt exposes the current skill, task, measured correctness and performance evidence, the reward objective, diagnostics, and generated code. Correctness restoration is explicitly prioritized whenever the candidate fails task-specific validation; otherwise, the prompt directs attention toward the largest weighted reward terms.

{Previous Skill Specification}
{Task Specification}
{Metric Summary: current and best values for correctness status, compiler warnings, static-analysis warnings, runtime, load imbalance, and reward}

$$\begin{aligned} \max \quad & r \\ \text{s.t.} \quad & r = -(\alpha_c n_c + \alpha_s n_s + \alpha_t t_{\max} \\ & \qquad + \alpha_\lambda \lambda + \alpha_f f). \end{aligned}$$

where $f = 1$ if correctness testing fails and $f = 0$ otherwise.

{Compiler Warnings Text}
{Static Analysis Warnings}
{Generated Code}
Produce the textual gradient for the skill specification above. Prioritize changes that restore correctness when $f = 1$; otherwise, prioritize changes that reduce the largest weighted reward terms.

Figure 4.3: MetaFORGE textual-gradient prompt. The prompt combines the current skill, task, execution-derived metrics, diagnostics, reward objective, and generated code to produce evidence-grounded guidance for the next skill update.

4.2.8.3 Skill-Rewrite Prompt

The rewrite prompt applies the previously generated textual gradient to the skill while requesting no unrelated output. Skill versioning provides a deterministic record of successive revisions.

{Previous Skill Definition}
{Computed Textual Gradient}
Rewrite the skill specification applying the gradient's guidance. Increment the 'version' field in the formatter by 1. Output ONLY the updated skill Markdown.

Figure 4.4: MetaFORGE skill-rewrite prompt. The prompt applies the computed textual gradient to the persistent skill while preserving the distinction between diagnosis and skill modification.

4.2.8.4 *Knowledge-Distillation Prompt*

The final prompt compresses action statements extracted from a trajectory. It preserves MPI-specific terminology, removes vague or redundant guidance, and resolves contradictions in favor of actions associated with improving iterations.

```
The following action lines were extracted from MPI optimization gradients across {n} candidate lines.
Best result achieved in this run:
{Best Candidate reward, compiler stats, static results, and runtime}
Action Lines:
{Action Lines}
Produce a concise summary of actionable recommendations.
Rules:
- Preserve specific MPI function names, flags, and data types.
- Merge near-duplicates into a single, precise statement.
- Resolve contradictions in favor of actions from improving iterations.
- Remove vague or non-actionable lines.
- Output only the recommendations, one per line, with no bullets, numbering, or extra commentary.
```

Figure 4.5: MetaFORGE knowledge-distillation prompt. Reward-supported action statements are compressed into concise, reusable recommendations for persistent cross-task memory.

4.3 Evaluation

4.3.1 Experimental Design

The primary evaluation used five MPI C programming tasks: Calculate π, Matrix Multiplication, Mandelbrot Sets, Heat Transfer using partial differential equations (PDEs), and Distributed principal component analysis (PCA). Each task was instantiated at small, medium, and large workload sizes. Five independent trials were conducted for every task–workload configuration. The primary code-generation and refinement model was `gpt-5.1-codex-max`, accessed through the OpenAI Python application programming interface (API) (v2.36.0). The model was selected for its

support of agentic coding and long-running software-engineering workflows [67, 68]. Its use is also consistent with prior evidence concerning Codex-style program synthesis and iterative code refinement [13, 66].

Each trial used an independent API conversation so that conversational state was not shared across repetitions. This design produces independent stochastic generation and refinement trajectories under the same task, workload, model, iteration budget, stopping rule, and evaluation configuration.

4.3.1.1 Compilation and Execution Environment

Generated programs were compiled with Clang 18.1.3 using the following command:

```
mpicc -Wall -Wextra -Wpedantic -O2 -std=c11
```

Static analysis used Clang-Tidy 18.1.3 with the following check groups:

```
clang-diagnostic-*, clang-analyzer-*,
clang-analyzer-optin.mpi.MPI-Checker, mpi-*,
bugprone-*, performance-*.
```

Runtime evaluation was performed on a four-node OpenMPI 4.1 Ubuntu Linux cluster. Each node contained an Intel i7 processor and 16 GB of memory. Runtime trials were executed with exclusive access to the cluster. The compilation, correctness, static-analysis, runtime, and load-imbalance pipeline was implemented independently of the refinement backend so that competing methods were evaluated by the same external procedure.

4.3.1.2 *Baselines and Experimental Controls*

Two baselines were used. The first was zero-shot Codex generation with an empty skill specification; this condition provides the reference program for reported percentage improvements. The second was the unmodified TEXTGRAD procedure for iterative skill refinement [114].

The METAFORGE and TEXTGRAD conditions began with the same initial skill, received the same task specification, used `gpt-5.1-codex-max` for candidate generation, queried the same cross-task memory, and shared the same maximum iteration budget, patience-based stopping criterion, and program-evaluation pipeline. The patience threshold was eight consecutive non-improving iterations. Both methods also received the same task, source code, compiler diagnostics, static-analysis findings, and top-k retrieval context.

The refinement procedures retained their native differences. With `-inject-reward-to-gradient` enabled, METAFORGE received the numeric current/preceding metric comparison and explicit weighted-reward decomposition. TEXTGRAD's loss-evaluation path consumed qualitative evidence rather than these numeric artifacts. A METAFORGE iteration used one model call for code generation followed by a textual-critique call and a skill-rewrite call, for three LLM calls per iteration. The unmodified TEXTGRAD sequence used code generation, forward loss evaluation, backward textual-gradient generation, and its optimizer, for four calls per iteration. METAFORGE's refinement calls used `gpt-5.1-codex-max`, whereas the unmodified TEXTGRAD backward engine used its configured `gpt-4o` model. The comparison is therefore an end-to-end comparison of the two refinement systems as implemented, rather than a model-call-matched ablation of only their update rules.

4.3.1.3 Metrics

Correctness was established using task-specific output checks. The remaining measures were compiler-warning count, static-analysis-warning count, critical-path wall-clock runtime, rank-level load imbalance, and token consumption. The primary experiments used $\alpha_c = 1.0$, $\alpha_s = 1.0$, $\alpha_t = 2.0$, $\alpha_\lambda = 1.5$, and $\alpha_f = 10000$. Static-analysis, runtime, and load-balance results are reported as percentage improvements relative to the corresponding zero-shot no-skill program for the same task and workload. Positive values therefore indicate improvement, whereas negative values indicate regression.

4.3.2 Overall Results

Table 4.4 presents the principal comparison across the five tasks and three workload sizes. At the task-average level, METAFORGE produces larger improvements than TEXTGRAD in all 15 comparisons formed by five tasks and three outcome dimensions: static-analysis quality, runtime, and load balance. At the individual workload level, METAFORGE produces the larger improvement in 44 of the 45 metric comparisons; the exception is load-balance improvement for the large matrix-multiplication workload.

The results also reveal task-dependent optimization behavior. Matrix multiplication, heat transfer, and distributed PCA exhibit the largest runtime improvements, whereas calculation of π produces the strongest static-analysis and load-balance gains. Mandelbrot generation yields a smaller runtime improvement, but METAFORGE remains positive in runtime and balance while TEXTGRAD is approximately neutral. Together, these tasks span regular reductions, dense linear algebra, irregular image generation, stencil computation, and distributed machine-learning computation.

Token consumption is more nuanced. On a per-iteration basis, METAFORGE uses fewer tokens

Table 4.4: Aggregate METAFORGE and TEXTGRAD results across five trials per workload. Static-analysis, runtime, and load-imbalance improvements are computed relative to zero-shot Codex; higher is better. Workload-level percentages are computed from pooled raw measurements and may therefore differ from the arithmetic means of the trial-level percentages reported in the detailed tables. Tokens denote mean per-iteration consumption; lower is better. Task-average rows summarize the three workload-level aggregate results. Avg. best iter. reports the mean iteration at which the best reward was obtained and is descriptive rather than a quality metric. Bold marks the better result.

Task	Workload	MetaFORGE					TextGrad [114]				
		Static improv.	Runtime improv.	Balance improv.	Avg. best iter.	Tokens	Static improv.	Runtime improv.	Balance improv.	Avg. best iter.	Tokens
π calculation	small	**100.00%**	**20.77%**	**97.49%**	10.2	**8548**	10.00%	10.93%	23.23%	**4.4**	11146
	medium	**100.00%**	**15.60%**	**93.00%**	9.6	**8391**	6.67%	10.42%	72.38%	**6.4**	12154
	large	**100.00%**	**12.98%**	**98.75%**	6.0	**7756**	10.00%	0.55%	10.54%	**3.8**	13926
	average	**100.00%**	**16.45%**	**96.41%**	8.60	**8232**	8.89%	7.30%	35.38%	**4.87**	12409
Matrix multiplication	small	**90.96%**	**93.49%**	**60.37%**	11.2	**15997**	-0.06%	55.60%	-22.73%	**6.6**	18127
	medium	**29.91%**	**71.95%**	**63.41%**	8.4	**15250**	-2.46%	36.85%	48.55%	**5.2**	17809
	large	**50.60%**	**98.52%**	17.34%	**6.8**	**15675**	-36.90%	73.94%	**24.89%**	8.8	18157
	average	**57.16%**	**87.99%**	**47.04%**	8.80	**15641**	-13.14%	55.46%	16.91%	**6.87**	18032
Mandelbrot set	small	**65.00%**	**2.15%**	**24.77%**	9.2	**15753**	4.58%	-1.26%	1.98%	**6.0**	16933
	medium	**73.47%**	**14.31%**	**30.20%**	6.4	16954	10.48%	-0.01%	-1.90%	**4.8**	**16317**
	large	**96.92%**	**11.49%**	**20.90%**	16.0	19597	3.85%	0.70%	0.36%	**2.6**	**16561**
	average	**78.47%**	**9.32%**	**25.29%**	10.53	17435	6.30%	-0.19%	0.15%	**4.47**	**16604**
Heat-transfer PDE	small	**27.94%**	**78.18%**	**41.23%**	14.4	21779	12.14%	23.92%	1.65%	**2.8**	**20143**
	medium	**87.78%**	**96.80%**	**51.40%**	15.8	21186	29.58%	1.53%	50.39%	**8.6**	**19544**
	large	**73.52%**	**72.39%**	**61.61%**	14.4	21085	-5.85%	19.08%	30.45%	**9.6**	**18437**
	average	**63.08%**	**82.46%**	**51.41%**	14.87	21351	11.96%	14.84%	27.50%	**7.00**	**19375**
Distributed PCA	small	**100.00%**	**68.91%**	**12.20%**	15.6	**17397**	44.63%	16.72%	1.55%	**9.0**	24175
	medium	**100.00%**	**99.45%**	**44.67%**	15.2	**18585**	16.79%	73.69%	12.68%	**5.2**	23150
	large	**83.33%**	**90.63%**	**49.00%**	10.0	**17914**	3.89%	21.15%	6.23%	**2.8**	25569
	average	**94.44%**	**86.33%**	**35.29%**	13.60	**17966**	21.77%	37.19%	6.82%	**5.67**	24299

for calculation of π, matrix multiplication, and distributed PCA, but more for heat transfer and Mandelbrot generation. Because the two methods have different native model-call sequences and can terminate after different numbers of iterations, per-iteration cost is not equivalent to end-to-end cost. Complete-run token results are reported separately below.

Figure 4.6 summarizes the relationship between optimization improvement and per-iteration token consumption across the five tasks.

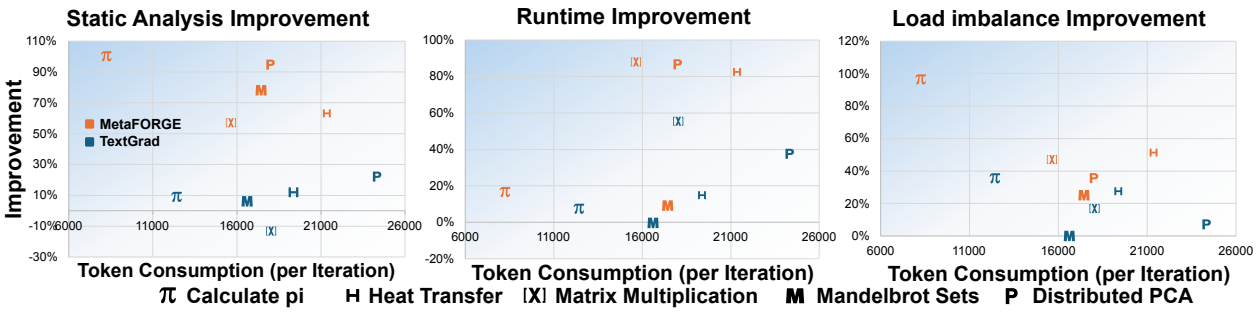


Figure 4.6: Average optimization improvement versus per-iteration token consumption for MetaFORGE and TextGrad across five MPI programming tasks. MetaFORGE achieves larger task-average improvements in static-analysis quality, runtime, and load-balance metrics across all five tasks, while per-iteration token consumption is task dependent: lower for Calculate π, Matrix Multiplication, and Distributed PCA, and higher for Mandelbrot Sets and Heat Transfer PDE.

4.3.3 Task-Level and Trial-Level Results

The following subsections retain the complete trial-level evidence for each benchmark. In every table, static-analysis, runtime, and balance columns are percentage improvements over the matching zero-shot baseline. The final rows report the arithmetic mean and sample standard deviation across the five trials; negative improvements are retained as measured rather than treated as missing data.

4.3.3.1 Calculate π

The π benchmark directs the model to approximate π using a summation-of-integrals method. The small, medium, and large workloads use 134,217,728, 268,435,456, and 536,870,912 intervals, respectively. Across the three workloads, METAFORGE achieves an average 100.00% static-analysis improvement, 16.45% runtime improvement, and 96.41% load-balance improvement, compared with 8.89%, 7.30%, and 35.38% for TEXTGRAD. Its average per-iteration token consumption is 8,232.34 versus 12,409.15 for TEXTGRAD.

Figure 4.7 visualizes the relationship between optimization improvement and per-iteration token consumption for the Calculate π benchmark.

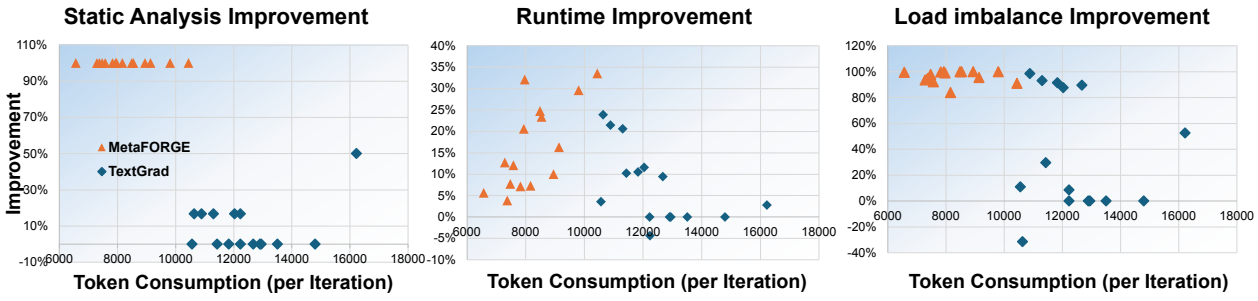


Figure 4.7: MPI π calculation results comparing MetaFORGE and TextGrad [114]. MetaFORGE achieves higher code-quality, runtime, and load-balance improvements with lower token consumption across small, medium, and large workloads over 15 trials. (5 trials per size)

Tables 4.5, 4.6, and 4.7 provide the trial-level evidence underlying these aggregate results. A particularly consistent pattern is the complete removal of static-analysis warnings by METAFORGE: the mean static-analysis improvement is 100.00% with zero standard deviation for the small, medium, and large workloads. Runtime improvement remains positive as workload size increases, with mean improvements of 20.77%, 15.60%, and 12.98%, respectively, while mean load-balance improvement remains above 93% for all three sizes. The large workload is especially informative because METAFORGE retains a 98.75% mean balance improvement while the TEXTGRAD results remain far more variable and provide little mean runtime improvement. The trial tables also show a favorable per-iteration cost trend: METAFORGE's mean token consumption decreases from approximately 8,549 tokens on the small workload to 7,757 on the large workload, whereas TEXTGRAD's mean per-iteration consumption increases with workload size. Collectively, the three tables indicate that the learned skill remains stable as the numerical-integration workload grows rather than achieving its aggregate advantage from a single workload size or isolated trial.

Table 4.5: Per-trial π-calculation results for the small workload.

Trial	Method	Iter.	Exec. iter.	Best iter.	Best reward	Static improv. (%)	Runtime improv. (%)	Balance improv. (%)	Total tokens	Tokens/ iteration
1	METAFORGE	12	11	6	-1.072	100.00	7.70	97.91	89,657	7,471.42
	TEXTGRAD	12	12	6	-6.082	16.67	23.88	-31.46	127,622	10,635.17
2	METAFORGE	16	15	10	-1.068	100.00	20.56	99.83	126,852	7,928.25
	TEXTGRAD	9	9	3	-6.083	0.00	3.55	11.03	95,055	10,561.67
3	METAFORGE	20	18	14	-1.070	100.00	9.98	99.89	178,765	8,938.25
	TEXTGRAD	8	8	2	-6.078	16.67	21.42	98.45	87,075	10,884.38
4	METAFORGE	20	18	18	-1.071	100.00	33.55	90.87	208,787	10,439.35
	TEXTGRAD	11	11	5	-6.075	0.00	10.19	29.54	125,670	11,424.55
5	METAFORGE	9	9	3	-1.074	100.00	32.08	98.94	71,694	7,966.00
	TEXTGRAD	12	12	6	-6.079	16.67	-4.40	8.60	146,711	12,225.92
Mean ± SD	METAFORGE	15.4 ± 4.9	14.2 ± 4.1	10.2 ± 6.0	-1.071 ± 0.002	100.00 ± 0.00	20.77 ± 12.03	97.49 ± 3.79	$135,151 \pm 58,073$	$8,548.65 \pm 1,184.41$
	TEXTGRAD	10.4 ± 1.8	10.4 ± 1.8	4.4 ± 1.8	-6.079 ± 0.003	10.00 ± 9.13	10.93 ± 11.92	23.23 ± 47.57	$116,427 \pm 24,729$	$11,146.33 \pm 691.85$

Table 4.6: Per-trial π-calculation results for the medium workload.

Trial	Method	Iter.	Exec. iter.	Best iter.	Best reward	Static improv. (%)	Runtime improv. (%)	Balance improv. (%)	Total tokens	Tokens/ iteration
1	METAFORGE	10	9	4	-1.145	100.00	12.75	93.55	72,877	7,287.70
	TEXTGRAD	12	11	6	-6.159	0.00	10.53	91.50	141,907	11,825.58
2	METAFORGE	10	9	4	-1.141	100.00	12.10	92.19	75,734	7,573.40
	TEXTGRAD	14	14	8	-6.160	16.67	20.58	93.19	158,231	11,302.21
3	METAFORGE	20	20	20	-1.139	100.00	16.30	95.37	173,519	9,132.58
	TEXTGRAD	17	17	11	-6.174	16.67	11.55	87.58	204,478	12,028.12
4	METAFORGE	12	12	6	-1.135	100.00	7.26	83.96	97,943	8,161.92
	TEXTGRAD	12	12	6	-6.175	0.00	9.43	89.61	152,069	12,672.42
5	METAFORGE	20	18	14	-1.135	100.00	29.58	99.94	196,072	9,803.60
	TEXTGRAD	7	6	1	-6.345	0.00	0.00	0.00	90,611	12,944.43
Mean ± SD	METAFORGE	14.4 ± 5.2	13.6 ± 5.1	9.6 ± 7.1	-1.139 ± 0.004	100.00 ± 0.00	15.60 ± 8.45	93.00 ± 5.84	$123,229 \pm 57,588$	$8,391.84 \pm 1,058.72$
	TEXTGRAD	12.4 ± 3.6	12.0 ± 4.1	6.4 ± 3.6	-6.202 ± 0.080	6.67 ± 9.13	10.42 ± 7.32	72.38 ± 40.51	$149,459 \pm 40,715$	$12,154.55 \pm 660.07$

4.3.3.2 Matrix Multiplication

Matrix multiplication was selected because it is a foundational operation in machine-learning tensors, image processing, and encryption [24]. It is also a demanding MPI benchmark because a high-quality implementation must preserve numerical correctness while selecting an appropriate decomposition and balancing communication and local work. The evaluated matrix dimensions were intentionally non-square and not evenly divisible across the four-node environment, preventing a trivial uniform partition.

At the task-average level, METAFORGE improves static-analysis quality by 57.16%, while

Table 4.7: Per-trial π-calculation results for the large workload.

Trial	Method	Iter.	Exec. iter.	Best iter.	Best reward	Static improv. (%)	Runtime improv. (%)	Balance improv. (%)	Total tokens	Tokens/ iteration
1	METAFORGE	10	9	4	-1.281	100.00	5.66	99.49	65,642	6,564.20
	TEXTGRAD	7	7	1	-7.346	0.00	0.00	0.00	90,282	12,897.43
2	METAFORGE	13	11	7	-1.279	100.00	23.41	99.99	102,491	8,540.92
	TEXTGRAD	7	7	1	-6.477	0.00	0.00	0.00	85,579	12,225.57
3	METAFORGE	12	11	6	-1.330	100.00	7.16	99.70	93,851	7,820.92
	TEXTGRAD	7	7	1	-6.755	0.00	0.00	0.00	94,509	13,501.29
4	METAFORGE	11	10	5	-1.287	100.00	3.89	94.72	81,061	7,369.18
	TEXTGRAD	7	5	1	-7.377	0.00	0.00	0.00	103,551	14,793.00
5	METAFORGE	14	14	8	-1.263	100.00	24.78	99.86	118,823	8,487.36
	TEXTGRAD	20	19	15	-4.435	50.00	2.75	52.69	324,311	16,215.55
Mean ± SD	METAFORGE	12.0 ± 1.6	11.0 ± 1.9	6.0 ± 1.6	-1.288 ± 0.025	100.00 ± 0.00	12.98 ± 10.22	98.75 ± 2.26	$92,374 \pm 20,290$	$7,756.51 \pm 825.41$
	TEXTGRAD	9.6 ± 5.8	9.0 ± 5.7	3.8 ± 6.3	-6.478 ± 1.206	10.00 ± 22.36	0.55 ± 1.23	10.54 ± 23.56	$139,646 \pm 103,443$	$13,926.57 \pm 1,590.87$

TEXTGRAD exhibits a 13.14% degradation. Runtime improves by 87.99% versus 55.46%, and load balance improves by 47.04% versus 16.91%. These improvements are accompanied by lower per-iteration token consumption: 15,641.22 for METAFORGE compared with 18,031.71 for TEXTGRAD.

Figure 4.8 summarizes the corresponding optimization and token-consumption behavior for Matrix Multiplication.

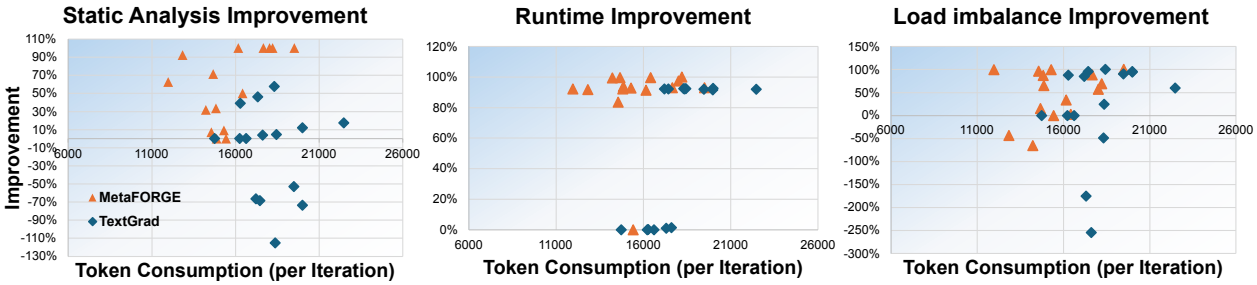


Figure 4.8: MPI matrix multiplication results comparing MetaFORGE and TextGrad across 15 trials. MetaFORGE achieves higher code-quality, runtime, and load-balance improvements.

Tables 4.8, 4.9, and 4.10 show that the matrix-multiplication gains arise from different optimization profiles across workload sizes. On the small workload, METAFORGE achieves mean improvements of 90.96% in static-analysis quality and 93.49% in runtime, while also improving load

Table 4.8: Per-trial matrix-multiplication results for the small workload.

Trial	Method	Iter.	Exec. iter.	Best iter.	Best reward	Static improv. (%)	Runtime improv. (%)	Balance improv. (%)	Total tokens	Tokens/ iteration
1	METAFORGE	12	4	6	-18.581	62.50	92.24	99.92	143,595	11,966.25
	TEXTGRAD	9	8	1	-20.634	0.00	0.00	0.00	132,627	14,736.33
2	METAFORGE	18	17	12	-1.000	100.00	97.45	57.28	324,435	18,024.17
	TEXTGRAD	10	10	2	-132.581	3.70	1.34	-254.74	176,071	17,607.10
3	METAFORGE	20	20	17	-9.007	100.00	92.95	88.44	353,321	17,666.05
	TEXTGRAD	21	19	14	-32.262	12.00	91.61	94.21	399,772	19,988.60
4	METAFORGE	20	20	19	-9.002	100.00	92.99	99.99	389,984	19,499.20
	TEXTGRAD	18	17	9	-42.036	-73.68	92.59	95.55	339,845	19,990.88
5	METAFORGE	8	8	2	-11.722	92.31	91.84	-43.81	102,665	12,833.12
	TEXTGRAD	15	14	7	-20.468	57.69	92.44	-48.66	274,746	18,316.40
Mean ± SD	METAFORGE	15.6 ± 5.4	13.8 ± 7.4	11.2 ± 7.2	-9.862 ± 6.314	90.96 ± 16.26	93.49 ± 2.26	60.37 ± 60.80	$262,800 \pm 130,405$	$15,997.76 \pm 3,369.63$
	TEXTGRAD	14.6 ± 5.1	13.6 ± 4.6	6.6 ± 5.3	-49.596 ± 47.253	-0.06 ± 47.21	55.60 ± 50.15	-22.73 ± 143.77	$264,612 \pm 111,007$	$18,127.86 \pm 2,164.62$

balance by 60.37%. The medium workload is more difficult from a static-analysis perspective, reducing the mean static improvement to 29.91%, but runtime and balance improvements remain substantial at 71.95% and 63.41%, respectively. On the large workload, runtime improvement rises to 98.52% and static-analysis improvement recovers to 50.60%. The large-workload load-balance result is the one workload-level case in the overall evaluation where TEXTGRAD achieves the larger mean balance improvement; however, METAFORGE still provides the stronger static-analysis and runtime results for that workload. Across all three sizes, METAFORGE also uses fewer tokens per iteration than TEXTGRAD. The trial-level results therefore show that its advantage is not limited to one matrix size, although the relative contribution of code quality, runtime, and balance changes as the decomposition problem becomes larger.

4.3.3.3 Mandelbrot Sets

Mandelbrot-set generation provides an irregular workload. Pixels are independent, but the number of iterations required to classify a point varies across the complex plane; points near the set boundary are generally more expensive than points that diverge rapidly. Consequently, simple static row- or block-based decompositions can create substantial rank-level imbalance. The task additionally

Table 4.9: Per-trial matrix-multiplication results for the medium workload.

Trial	Method	Iter.	Exec. iter.	Best iter.	Best reward	Static improv. (%)	Runtime improv. (%)	Balance improv. (%)	Total tokens	Tokens/ iteration
1	METAFORGE	15	14	9	-32.672	33.33	92.21	87.00	222,329	14,821.93
	TEXTGRAD	9	8	1	-228.006	0.00	0.00	0.00	149,584	16,620.44
2	METAFORGE	9	9	3	-54.924	7.14	83.50	96.25	130,999	14,555.44
	TEXTGRAD	11	10	3	-49.596	-68.42	92.15	95.64	191,867	17,442.45
3	METAFORGE	15	15	9	-26.961	9.09	92.76	99.89	229,347	15,289.80
	TEXTGRAD	9	8	1	-52.536	0.00	0.00	0.00	146,097	16,233.00
4	METAFORGE	7	7	1	-39.490	0.00	0.00	0.00	107,983	15,426.14
	TEXTGRAD	20	18	19	-40.718	17.24	92.00	59.65	449,399	22,469.95
5	METAFORGE	20	19	20	-19.478	100.00	91.30	33.93	323,138	16,156.90
	TEXTGRAD	10	6	2	-217.332	38.89	0.10	87.48	162,808	16,280.80
Mean ± SD	METAFORGE	13.2 ± 5.2	12.8 ± 4.8	8.4 ± 7.4	-34.705 ± 13.488	29.91 ± 41.14	71.95 ± 40.40	63.41 ± 44.31	$202,759 \pm 86,187$	$15,250.04 \pm 616.59$
	TEXTGRAD	11.8 ± 4.7	10.0 ± 4.7	5.2 ± 7.8	-117.638 ± 96.053	-2.46 ± 40.19	36.85 ± 50.41	48.55 ± 46.29	$219,951 \pm 129,524$	$17,809.33 \pm 2,650.03$

Table 4.10: Per-trial matrix-multiplication results for the large workload.

Trial	Method	Iter.	Exec. iter.	Best iter.	Best reward	Static improv. (%)	Runtime improv. (%)	Balance improv. (%)	Total tokens	Tokens/ iteration
1	METAFORGE	9	9	3	-8.713	71.43	99.68	15.40	131,976	14,664.00
	TEXTGRAD	16	14	8	-52.336	45.95	0.97	-175.08	276,971	17,310.69
2	METAFORGE	20	20	20	-1.021	100.00	100.00	69.32	364,294	18,214.70
	TEXTGRAD	18	18	10	-67.685	-66.67	92.13	85.04	309,841	17,213.39
3	METAFORGE	9	8	3	-20.216	50.00	99.54	2.37	147,783	16,420.33
	TEXTGRAD	16	16	8	-59.502	-115.38	92.22	24.48	293,795	18,362.19
4	METAFORGE	9	9	3	-39.804	0.00	93.86	65.23	133,718	14,857.56
	TEXTGRAD	16	16	8	-52.789	4.55	92.33	100.00	294,897	18,431.06
5	METAFORGE	11	10	5	-16.347	31.58	99.50	-65.64	156,450	14,222.73
	TEXTGRAD	18	16	10	-57.577	-52.94	92.06	90.01	350,503	19,472.39
Mean ± SD	METAFORGE	11.6 ± 4.8	11.2 ± 5.0	6.8 ± 7.4	-17.220 ± 14.617	50.60 ± 38.07	98.52 ± 2.61	17.34 ± 55.02	$186,844 \pm 99,713$	$15,675.86 \pm 1,643.55$
	TEXTGRAD	16.8 ± 1.1	16.0 ± 1.4	8.8 ± 1.1	-57.978 ± 6.234	-36.90 ± 63.02	73.94 ± 40.79	24.89 ± 115.64	$305,201 \pm 27,870$	$18,157.94 \pm 929.26$

requires the distributed pixel results to be reassembled into a correct high-resolution color image.

METAFORGE improves static-analysis quality by 78.47%, compared with 6.30% for TEXTGRAD; runtime by 9.32%, compared with a 0.19% regression; and load balance by 25.29%, compared with 0.15%. METAFORGE uses 17,435.36 tokens per iteration on average, modestly more than TEXTGRAD's 16,604.08, illustrating a case in which higher inference cost is accompanied by substantially stronger optimization gains.

Figure 4.9 summarizes the optimization and token-consumption behavior for Mandelbrot Sets.

Tables 4.11, 4.12, and 4.13 illustrate how METAFORGE responds to an increasingly irregular

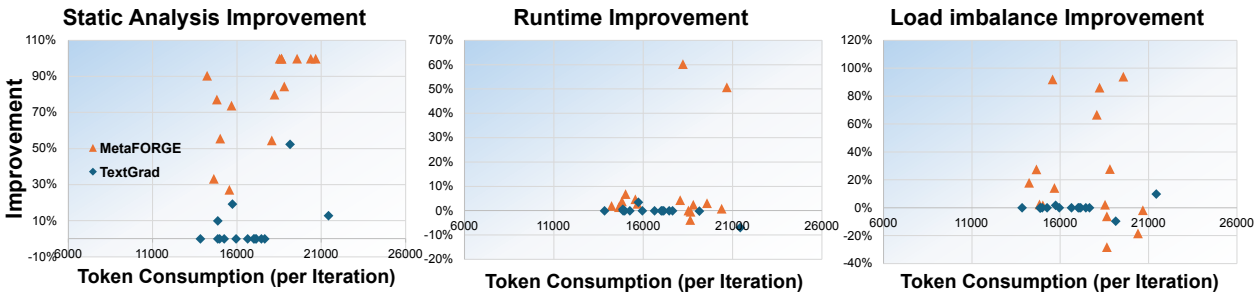


Figure 4.9: Mandelbrot-set generation results comparing METAFORGE and TEXTGRAD across 15 trials. METAFORGE achieves stronger static-analysis warning reduction, runtime improvement, and load-balance improvement, with modestly higher aggregate per-iteration token consumption.

workload. Mean static-analysis improvement increases from 65.00% for the small workload to 73.47% for the medium workload and 96.92% for the large workload, indicating that refinement becomes particularly effective at eliminating analyzer findings in the most demanding configuration. Runtime improvement is smaller than for the dense linear-algebra and stencil tasks, but remains positive for all three sizes: 2.15%, 14.31%, and 11.49%. Mean load-balance improvement is likewise positive throughout, ranging from 20.90% to 30.20%, whereas TEXTGRAD remains near zero or regresses on several of these execution measures. The large workload requires more refinement effort: METAFORGE reaches its best reward at a mean iteration of 16.0 and consumes more tokens per iteration than TEXTGRAD. This cost increase is accompanied by the strongest static-analysis improvement and materially better runtime and balance behavior, showing that the additional refinement effort is associated with measurable optimization rather than merely longer trajectories.

Table 4.11: Per-trial Mandelbrot-set results for the small workload.

Trial	Method	Iter.	Exec. iter.	Best iter.	Best reward	Static improv. (%)	Runtime improv. (%)	Balance improv. (%)	Total tokens	Tokens/ iteration
1	METAFORGE	13	12	7	-25.181	73.91	2.84	14.32	203,819	15,678.38
	TEXTGRAD	8	1	3	-62.261	0.00	0.00	0.00	110,732	13,841.50
2	METAFORGE	17	16	11	-22.589	90.48	2.00	17.94	242,083	14,240.18
	TEXTGRAD	6	4	1	-41.816	0.00	0.00	0.00	102,437	17,072.83
3	METAFORGE	11	10	5	-28.757	33.33	1.51	27.61	161,096	14,645.09
	TEXTGRAD	18	14	12	-71.217	12.90	-6.84	10.00	385,691	21,427.28
4	METAFORGE	20	19	19	-20.572	100.00	-0.41	-28.09	372,906	18,645.30
	TEXTGRAD	19	4	13	-57.602	10.00	0.56	-0.10	282,484	14,867.58
5	METAFORGE	10	10	4	-39.211	27.27	4.81	92.06	155,598	15,559.80
	TEXTGRAD	6	3	1	-42.133	0.00	0.00	0.00	104,734	17,455.67
Mean ± SD	METAFORGE	14.2 ± 4.2	13.4 ± 4.0	9.2 ± 6.1	-27.262 ± 7.348	65.00 ± 33.09	2.15 ± 1.91	24.77 ± 43.26	$227,100 \pm 88,734$	$15,753.75 \pm 1,726.59$
	TEXTGRAD	11.4 ± 6.5	5.2 ± 5.1	6.0 ± 6.0	-55.006 ± 12.864	4.58 ± 6.36	-1.26 ± 3.13	1.98 ± 4.48	$197,216 \pm 130,201$	$16,932.97 \pm 2,928.93$

Table 4.12: Per-trial Mandelbrot-set results for the medium workload.

Trial	Method	Iter.	Exec. iter.	Best iter.	Best reward	Static improv. (%)	Runtime improv. (%)	Balance improv. (%)	Total tokens	Tokens/ iteration
1	METAFORGE	12	11	4	-11.328	80.00	60.24	86.18	218,732	18,227.67
	TEXTGRAD	20	16	18	-31.637	52.38	-0.07	-9.49	382,966	19,148.30
2	METAFORGE	18	18	10	-18.957	55.56	6.85	1.72	270,328	15,018.22
	TEXTGRAD	8	4	1	-21.224	0.00	0.00	0.00	133,101	16,637.62
3	METAFORGE	13	12	5	-16.544	77.27	3.88	2.50	192,676	14,821.23
	TEXTGRAD	8	5	1	-32.459	0.00	0.00	0.00	127,596	15,949.50
4	METAFORGE	15	13	7	-12.035	100.00	-3.74	-6.05	279,616	18,641.07
	TEXTGRAD	10	4	3	-34.587	0.00	0.00	0.00	149,736	14,973.60
5	METAFORGE	14	9	6	-26.242	54.55	4.34	66.67	252,908	18,064.86
	TEXTGRAD	8	2	1	-33.633	0.00	0.00	0.00	119,035	14,879.38
Mean ± SD	METAFORGE	14.4 ± 2.3	12.6 ± 3.4	6.4 ± 2.3	-17.021 ± 6.047	73.47 ± 18.97	14.31 ± 25.98	30.20 ± 42.88	$242,852 \pm 36,425$	$16,954.61 \pm 1,870.72$
	TEXTGRAD	10.8 ± 5.2	6.2 ± 5.6	4.8 ± 7.4	-30.708 ± 5.419	10.48 ± 23.43	-0.01 ± 0.03	-1.90 ± 4.24	$182,487 \pm 112,631$	$16,317.68 \pm 1,741.09$

4.3.3.4 Heat Transfer Using Partial Differential Equations

The heat-transfer benchmark stresses numerical correctness, domain decomposition, halo exchange, synchronization, and communication efficiency. The generated program solves a heat-transfer partial differential equation using a Poisson/Laplace formulation with Jacobi relaxation and produces a visualization of the resulting heat field. Because stencil computation advances at the pace of the slowest rank, inefficient partitioning or synchronization directly affects runtime.

Across workloads, METAFORGE achieves 63.08% static-analysis improvement, 82.46% runtime improvement, and 51.41% load-balance improvement. The corresponding TEXTGRAD values are

Table 4.13: Per-trial Mandelbrot-set results for the large workload.

Trial	Method	Iter.	Exec. iter.	Best iter.	Best reward	Static improv. (%)	Runtime improv. (%)	Balance improv. (%)	Total tokens	Tokens/ iteration
1	METAFORGE	20	18	15	-21.537	100.00	0.83	-18.18	407,872	20,393.60
	TEXTGRAD	10	6	3	-44.490	0.00	0.00	0.00	170,111	17,011.10
2	METAFORGE	20	19	20	-21.091	100.00	3.10	94.00	391,483	19,574.15
	TEXTGRAD	14	5	6	-44.473	19.23	3.52	1.78	220,271	15,733.64
3	METAFORGE	20	19	17	-11.024	100.00	50.77	-1.58	413,642	20,682.10
	TEXTGRAD	10	0	2	-10010.000	0.00	0.00	0.00	152,592	15,259.20
4	METAFORGE	17	16	9	-25.501	84.62	2.56	27.94	319,676	18,804.47
	TEXTGRAD	8	4	1	-40.385	0.00	0.00	0.00	141,216	17,652.00
5	METAFORGE	20	19	19	-21.440	100.00	0.18	2.30	370,686	18,534.30
	TEXTGRAD	8	3	1	-43.604	0.00	0.00	0.00	154,368	17,152.00
Mean ± SD	METAFORGE	19.4 ± 1.3	18.2 ± 1.3	16.0 ± 4.4	-20.119 ± 5.394	96.92 ± 6.88	11.49 ± 21.99	20.90 ± 44.08	$380,672 \pm 37,960$	$19,597.72 \pm 944.72$
	TEXTGRAD	10.0 ± 2.4	3.6 ± 2.3	2.6 ± 2.1	$-2,036.590 \pm 4,457.272$	3.85 ± 8.60	0.70 ± 1.57	0.36 ± 0.79	$167,712 \pm 31,132$	$16,561.59 \pm 1,015.06$

11.96%, 14.84%, and 27.50%. METAFORGE uses 21,350.48 tokens per iteration compared with 19,375.34 for TEXTGRAD; the additional per-iteration cost is accompanied by substantially larger code-quality and execution improvements.

Figure 4.10 summarizes the optimization and token-consumption behavior for the Heat Transfer PDE benchmark.

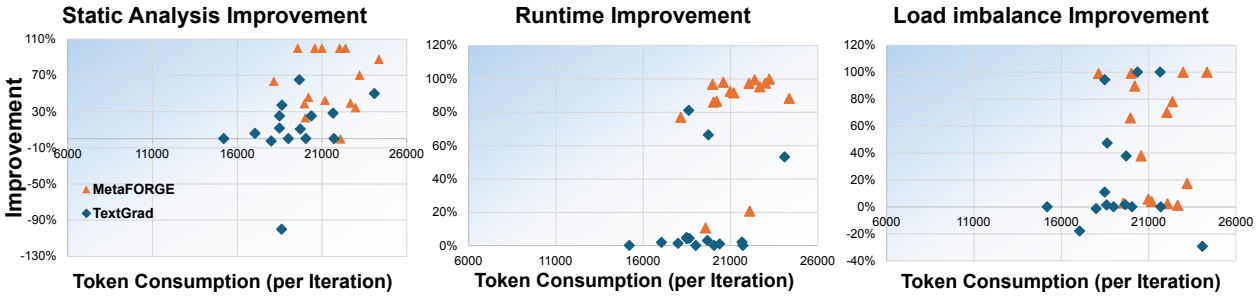


Figure 4.10: Heat-transfer PDE results comparing MetaFORGE and TextGrad. MetaFORGE achieves stronger code-quality, runtime, and load-balance improvements, with modestly higher token consumption over 15 trials.

Tables 4.14, 4.15, and 4.16 show that the heat-transfer benchmark benefits from METAFORGE across all three workload scales despite the communication and synchronization demands of iterative halo exchange. For the small workload, mean static-analysis, runtime, and load-balance im-

provements are 27.94%, 78.18%, and 41.23%, respectively. The medium workload produces the strongest combination of code-quality and performance gains, with 87.78% mean static-analysis improvement and 96.80% runtime improvement, while load balance also improves substantially. For the large workload, METAFORGE maintains 73.52% static-analysis improvement, 72.39% runtime improvement, and 61.61% load-balance improvement. The detailed tables also reveal that these gains generally require longer refinement than TEXTGRAD: the mean best iteration for METAFORGE is approximately 14–16 across the three sizes. Per-iteration token use is also higher on this task, but the additional inference cost corresponds to much larger improvements in the measured program-quality and execution metrics. Thus, the trial-level evidence supports the interpretation that execution-grounded refinement remains effective as the stencil workload and communication burden increase.

Table 4.14: Per-trial heat-transfer PDE results for the small workload.

Trial	Method	Iter.	Exec. iter.	Best iter.	Best reward	Static improv. (%)	Runtime improv. (%)	Balance improv. (%)	Total tokens	Tokens/ iteration
1	METAFORGE	20	17	17	-24.759	42.50	91.63	3.86	423,434	21,171.70
	TEXTGRAD	14	7	7	-35.266	10.71	66.36	37.65	275,961	19,711.50
2	METAFORGE	14	4	6	-35.469	0.00	20.69	2.24	309,170	22,083.57
	TEXTGRAD	9	2	1	-42.510	0.00	0.00	0.00	136,708	15,189.78
3	METAFORGE	20	12	18	-22.268	39.39	95.15	1.13	453,188	22,659.40
	TEXTGRAD	12	8	4	-45.201	50.00	53.24	-29.38	288,915	24,076.25
4	METAFORGE	20	16	16	-20.562	34.48	97.45	99.87	459,460	22,973.00
	TEXTGRAD	8	6	1	-44.026	0.00	0.00	0.00	173,585	21,698.12
5	METAFORGE	20	15	15	-24.722	23.33	85.99	99.06	400,181	20,009.05
	TEXTGRAD	8	1	1	-45.356	0.00	0.00	0.00	160,339	20,042.38
Mean ± SD	METAFORGE	18.8 ± 2.7	12.8 ± 5.3	14.4 ± 4.8	-25.556 ± 5.817	27.94 ± 17.23	78.18 ± 32.43	41.23 ± 53.17	$409,087 \pm 60,727$	$21,779.34 \pm 1,203.64$
	TEXTGRAD	10.2 ± 2.7	4.8 ± 3.1	2.8 ± 2.7	-42.472 ± 4.186	12.14 ± 21.67	23.92 ± 33.08	1.65 ± 23.81	$207,102 \pm 70,179$	$20,143.61 \pm 3,264.18$

4.3.3.5 Distributed Principal Component Analysis

Distributed principal component analysis (PCA) was selected because it is a widely used machine-learning method for dimensionality reduction, feature extraction, noise reduction, and exploratory analysis. Its distributed implementation combines data decomposition, local statistics, global aggregation of means and covariance contributions, and iterative eigensolver computation. These

Table 4.15: Per-trial heat-transfer PDE results for the medium workload.

Trial	Method	Iter.	Exec. iter.	Best iter.	Best reward	Static improv. (%)	Runtime improv. (%)	Balance improv. (%)	Total tokens	Tokens/ iteration
1	METAFORGE	20	16	12	-26.912	38.89	96.75	65.80	399,218	19,960.90
	TEXTGRAD	15	4	8	-113.356	5.88	1.86	-17.99	255,631	17,042.07
2	METAFORGE	20	18	19	-1.788	100.00	92.52	5.62	419,707	20,985.35
	TEXTGRAD	17	10	9	-108.301	65.12	2.97	1.59	334,238	19,661.06
3	METAFORGE	20	17	17	-1.409	100.00	99.50	77.85	447,410	22,370.50
	TEXTGRAD	19	11	12	-105.967	28.21	2.10	99.98	411,419	21,653.63
4	METAFORGE	20	16	13	-3.951	100.00	97.92	37.75	411,451	20,572.55
	TEXTGRAD	8	3	1	-110.317	0.00	0.00	0.00	152,006	19,000.75
5	METAFORGE	20	15	18	-3.056	100.00	97.33	70.00	440,831	22,041.55
	TEXTGRAD	20	6	13	-108.138	25.00	1.06	99.98	407,325	20,366.25
Mean ± SD	METAFORGE	20.0 ± 0.0	16.4 ± 1.1	15.8 ± 3.1	-7.423 ± 10.942	87.78 ± 27.33	96.80 ± 2.61	51.40 ± 29.73	$423,723 \pm 20,131$	$21,186.17 \pm 1,006.54$
	TEXTGRAD	15.8 ± 4.8	6.8 ± 3.6	8.6 ± 4.7	-109.216 ± 2.779	24.84 ± 25.54	1.60 ± 1.12	36.71 ± 58.27	$312,124 \pm 109,819$	$19,544.75 \pm 1,709.91$

Table 4.16: Per-trial heat-transfer PDE results for the large workload.

Trial	Method	Iter.	Exec. iter.	Best iter.	Best reward	Static improv. (%)	Runtime improv. (%)	Balance improv. (%)	Total tokens	Tokens/ iteration
1	METAFORGE	20	13	20	-12.003	70.27	99.82	17.23	463,985	23,199.25
	TEXTGRAD	13	9	5	-293.298	36.84	4.26	47.24	242,217	18,632.08
2	METAFORGE	16	9	8	-57.625	45.95	86.56	89.53	323,152	20,197.00
	TEXTGRAD	20	9	18	-304.377	11.63	4.06	10.84	369,697	18,484.85
3	METAFORGE	19	14	11	-54.421	63.64	76.82	98.95	344,715	18,142.89
	TEXTGRAD	19	7	11	-296.371	25.00	4.78	94.26	351,141	18,481.11
4	METAFORGE	20	12	20	-36.601	100.00	10.49	2.59	391,036	19,551.80
	TEXTGRAD	17	7	9	-107.932	-100.00	81.00	1.29	316,198	18,599.88
5	METAFORGE	20	14	13	-11.357	87.76	88.25	99.72	486,774	24,338.70
	TEXTGRAD	13	4	5	-315.978	-2.70	1.30	-1.36	233,876	17,990.46
Mean ± SD	METAFORGE	19.0 ± 1.7	12.4 ± 2.1	14.4 ± 5.4	-34.401 ± 22.236	73.52 ± 21.05	72.39 ± 35.56	61.61 ± 47.64	$401,932 \pm 71,846$	$21,085.93 \pm 2,590.94$
	TEXTGRAD	16.4 ± 3.3	7.2 ± 2.0	9.6 ± 5.4	-263.591 ± 87.457	-5.85 ± 54.67	19.08 ± 34.64	30.45 ± 40.62	$302,626 \pm 62,073$	$18,437.68 \pm 258.95$

stages exercise dense numerical computation, collective communication, synchronization, memory layout, and rank-level balance.

At the task-average level, METAFORGE achieves a 94.44% static-analysis improvement compared with 21.77% for TEXTGRAD, an 86.33% runtime improvement compared with 37.19%, and a 35.29% load-balance improvement compared with 6.82%. It also uses fewer tokens per iteration, 17,965.56 versus 24,298.55.

Figure 4.11 summarizes the optimization and token-consumption behavior for Distributed PCA.

Tables 4.17, 4.18, and 4.19 provide the detailed evidence for the distributed PCA results. Static-

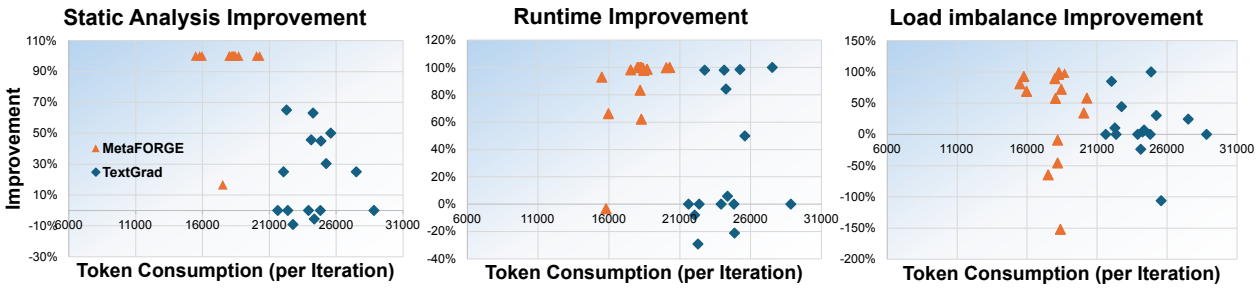


Figure 4.11: Distributed PCA results comparing MetaFORGE and TextGrad. MetaFORGE achieves higher code-quality, runtime, and load-balance improvements with lower per-iteration token consumption over 15 trials.

analysis behavior is highly consistent for METAFORGE: every small- and medium-workload trial achieves 100.00% improvement, and the large workload still averages 83.33%. Runtime optimization is also strong across scale. The small workload averages 68.91% runtime improvement, the medium workload approaches complete runtime improvement across its five trials, and the large workload maintains a 90.63% mean improvement. Load-balance results are more variable than static-analysis or runtime results, including negative trial-level values in some runs, but the mean remains positive for all three workload sizes. This variability is expected to be important when interpreting PCA because collective aggregation and eigensolver synchronization can magnify rank-level timing differences even when total runtime improves. Finally, METAFORGE consumes substantially fewer tokens per iteration than TEXTGRAD for all three PCA workloads. These detailed results show that the aggregate PCA advantage reflects persistent improvements across workload sizes, while also revealing that load balance is the least stable of the three optimization objectives.

4.3.4 Run-Level Token Consumption

Per-iteration token consumption does not account for differences in trajectory length. Table 4.20 therefore reports end-to-end prompt, processing, and total tokens for each completed optimization

Table 4.17: Per-trial distributed PCA results for the small workload.

Trial	Method	Iter.	Best iter.	Best reward	Static improv. (%)	Runtime improv. (%)	Balance improv. (%)	Total tokens	Tokens/ iteration
1	METAFORGE	19	11	-2.665	100.00	66.21	68.63	303,246	15,960.32
	TEXTGRAD	9	1	-22.526	0.00	0.00	0.00	215,161	23,906.78
2	METAFORGE	20	13	-2.486	100.00	-3.32	92.40	315,590	15,779.50
	TEXTGRAD	20	13	-14.850	45.00	-21.23	99.76	497,195	24,859.75
3	METAFORGE	20	16	-1.341	100.00	83.24	-45.98	363,650	18,182.50
	TEXTGRAD	20	12	-10.742	65.00	-29.22	10.05	445,359	22,267.95
4	METAFORGE	20	18	-1.523	100.00	98.82	97.97	373,570	18,678.50
	TEXTGRAD	20	15	-9.795	50.00	49.85	-106.12	511,755	25,587.75
5	METAFORGE	20	20	-1.366	100.00	99.59	-152.02	367,740	18,387.00
	TEXTGRAD	12	4	-8.209	63.16	84.21	4.08	291,072	24,256.00
Mean ± SD	METAFORGE	19.8 ± 0.4	15.6 ± 3.6	-1.876 ± 0.645	100.00 ± 0.00	68.91 ± 42.62	12.20 ± 108.77	$344,759 \pm 32,746$	$17,397.56 \pm 1,407.10$
	TEXTGRAD	16.2 ± 5.3	9.0 ± 6.1	-13.225 ± 5.750	44.63 ± 26.36	16.72 ± 48.69	1.55 ± 73.01	$392,108 \pm 132,016$	$24,175.65 \pm 1,242.99$

Table 4.18: Per-trial distributed PCA results for the medium workload.

Trial	Method	Iter.	Best iter.	Best reward	Static improv. (%)	Runtime improv. (%)	Balance improv. (%)	Total tokens	Tokens/ iteration
1	METAFORGE	20	16	-1.006	100.00	99.95	89.39	359,794	17,989.70
	TEXTGRAD	11	3	-20.848	25.00	-7.88	84.89	242,432	22,039.27
2	METAFORGE	20	20	-1.034	100.00	100.00	57.63	360,697	18,034.85
	TEXTGRAD	13	5	-21.388	-9.09	98.16	44.28	295,752	22,750.15
3	METAFORGE	20	16	-1.024	100.00	99.99	-9.17	363,838	18,191.90
	TEXTGRAD	9	1	-20.858	0.00	0.00	0.00	194,512	21,612.44
4	METAFORGE	20	16	-1.140	100.00	99.95	57.83	405,276	20,263.80
	TEXTGRAD	20	15	-21.509	30.43	98.55	30.30	504,622	25,231.10
5	METAFORGE	16	8	-1.114	100.00	97.84	72.38	295,122	18,445.12
	TEXTGRAD	10	2	-18.551	45.83	98.06	-23.84	241,191	24,119.10
Mean ± SD	METAFORGE	19.2 ± 1.8	15.2 ± 4.4	-1.064 ± 0.059	100.00 ± 0.00	99.55 ± 0.95	53.61 ± 37.44	$356,945 \pm 39,458$	$18,585.08 \pm 955.18$
	TEXTGRAD	12.6 ± 4.4	5.2 ± 5.7	-20.631 ± 1.201	18.44 ± 22.56	57.38 ± 56.05	27.13 ± 41.76	$295,702 \pm 122,164$	$23,150.41 \pm 1,502.31$

run. These totals incorporate early stopping and the native model-call sequence of each method. As a result, they should be interpreted as complete-system costs rather than as a model-call-matched efficiency ablation. Across the 15 workload-level comparisons, METAFORGE consumes fewer total tokens in six.

4.3.5 Retrieval-Memory Ablation

To isolate the contribution of external retrieval memory, METAFORGE was evaluated on calculation of π both with and without context retrieval. Table 4.21 shows that both configurations

Table 4.19: Per-trial distributed PCA results for the large workload.

Trial	Method	Iter.	Best iter.	Best reward	Static improv. (%)	Runtime improv. (%)	Balance improv. (%)	Total tokens	Tokens/ iteration
1	METAFORGE	14	6	-6.050	16.67	98.18	-65.00	245,178	17,512.71
	TEXTGRAD	12	4	-39.047	-5.56	5.76	6.72	292,295	24,357.92
2	METAFORGE	19	11	-1.024	100.00	92.94	80.94	294,289	15,488.89
	TEXTGRAD	9	1	-68.156	0.00	0.00	0.00	201,220	22,357.78
3	METAFORGE	20	13	-1.004	100.00	100.00	95.99	365,050	18,252.50
	TEXTGRAD	9	1	-70.885	0.00	0.00	0.00	223,229	24,803.22
4	METAFORGE	20	14	-1.034	100.00	99.93	33.92	400,875	20,043.75
	TEXTGRAD	9	1	-2167.000	0.00	0.00	0.00	259,410	28,823.33
5	METAFORGE	14	6	-1.011	100.00	62.08	99.15	255,812	18,272.29
	TEXTGRAD	15	7	-13.075	25.00	99.99	24.44	412,586	27,505.73
Mean ± SD	METAFORGE	17.4 ± 3.1	10.0 ± 3.8	-2.024 ± 2.251	83.33 ± 37.27	90.63 ± 16.21	49.00 ± 68.86	$312,241 \pm 68,280$	$17,914.03 \pm 1,644.76$
	TEXTGRAD	10.8 ± 2.7	2.8 ± 2.7	-471.633 ± 948.033	3.89 ± 12.04	21.15 ± 44.15	6.23 ± 10.59	$277,748 \pm 83,003$	$25,569.60 \pm 2,583.93$

eliminate the static-analysis warnings in all three workloads, yielding 100% static-analysis improvement. The principal difference is therefore performance rather than static quality.

With retrieval enabled, runtime improves by 20.77%, 15.60%, and 12.98% for the small, medium, and large workloads. Without retrieval, the corresponding improvements are 8.22%, 9.38%, and 8.22%. Load-balance performance remains strong in both conditions. Retrieval also reduces per-iteration token consumption for the medium and large workloads, whereas the no-context condition is slightly less expensive for the small workload. These results indicate that stored cross-run knowledge primarily assists performance-oriented refinement on this benchmark.

Figure 4.12 visualizes the effect of retrieval memory on static-analysis, runtime, and load-imbalance improvement relative to per-iteration token consumption.

4.3.6 Held-Out Cross-Task Transfer

A leave-one-task-out experiment was used to determine whether METAFORGE learns optimization knowledge that remains useful when the target task was excluded from skill and memory construction. The transfer pool contained calculation of π, Matrix Multiplication, and Mandelbrot

Table 4.20: Run-level optimization cost across five independent trials per workload. Values are mean ± sample standard deviation. Prompt and processing tokens sum to total tokens. Lower token counts indicate lower measured token consumption.

Task	Workload	Method	Iterations per run	Prompt tokens	Processing tokens	Total tokens	Tokens per iteration
Calculate π	small	METAFORGE	15.4 ± 4.9	$68,165 \pm 32,201$	$66,986 \pm 26,256$	$135,151 \pm 58,073$	$8,548.65 \pm 1,184.41$
		TEXTGRAD	10.4 ± 1.8	$89,999 \pm 18,527$	$26,428 \pm 6,563$	$116,427 \pm 24,729$	$11,146.33 \pm 691.85$
	medium	METAFORGE	14.4 ± 5.2	$61,614 \pm 34,457$	$61,615 \pm 23,171$	$123,229 \pm 57,588$	$8,391.84 \pm 1,058.72$
		TEXTGRAD	12.4 ± 3.6	$115,813 \pm 32,393$	$33,647 \pm 9,239$	$149,459 \pm 40,715$	$12,154.55 \pm 660.07$
	large	METAFORGE	12.0 ± 1.6	$44,926 \pm 10,514$	$47,447 \pm 10,223$	$92,374 \pm 20,290$	$7,756.51 \pm 825.41$
		TEXTGRAD	9.6 ± 5.8	$108,045 \pm 78,573$	$31,601 \pm 24,903$	$139,646 \pm 103,443$	$13,926.57 \pm 1,590.87$
Matrix multiplication	small	METAFORGE	15.6 ± 5.4	$98,021 \pm 47,199$	$164,779 \pm 83,286$	$262,800 \pm 130,405$	$15,997.76 \pm 3,369.63$
		TEXTGRAD	14.6 ± 5.1	$177,462 \pm 78,398$	$87,150 \pm 33,408$	$264,612 \pm 111,007$	$18,127.86 \pm 2,164.62$
	medium	METAFORGE	13.2 ± 5.2	$83,015 \pm 44,263$	$119,744 \pm 42,115$	$202,759 \pm 86,187$	$15,250.04 \pm 616.59$
		TEXTGRAD	11.8 ± 4.7	$142,465 \pm 73,524$	$77,486 \pm 56,264$	$219,951 \pm 129,524$	$17,809.33 \pm 2,650.03$
	large	METAFORGE	11.6 ± 4.8	$72,736 \pm 48,289$	$114,108 \pm 51,734$	$186,844 \pm 99,713$	$15,675.86 \pm 1,643.55$
		TEXTGRAD	16.8 ± 1.1	$208,605 \pm 22,330$	$96,596 \pm 10,820$	$305,201 \pm 27,870$	$18,157.94 \pm 929.26$
Mandelbrot Sets	small	METAFORGE	14.2 ± 4.2	$94,302 \pm 43,642$	$132,798 \pm 45,934$	$227,100 \pm 88,734$	$15,753.75 \pm 1,726.59$
		TEXTGRAD	11.4 ± 6.5	$125,950 \pm 78,644$	$71,266 \pm 53,146$	$197,216 \pm 130,201$	$16,932.97 \pm 2,928.93$
	medium	METAFORGE	14.4 ± 2.3	$93,944 \pm 19,429$	$148,908 \pm 22,976$	$242,852 \pm 36,425$	$16,954.61 \pm 1,870.72$
		TEXTGRAD	10.8 ± 5.2	$127,125 \pm 78,074$	$55,362 \pm 34,682$	$182,487 \pm 112,631$	$16,317.68 \pm 1,741.09$
	large	METAFORGE	19.4 ± 1.3	$157,616 \pm 20,455$	$223,055 \pm 27,376$	$380,672 \pm 37,960$	$19,597.72 \pm 944.72$
		TEXTGRAD	10.0 ± 2.4	$106,788 \pm 28,969$	$60,188 \pm 10,317$	$166,976 \pm 31,567$	$16,898.41 \pm 1,447.33$
Heat-transfer PDE	small	METAFORGE	18.8 ± 2.7	$150,460 \pm 33,844$	$258,627 \pm 35,237$	$409,087 \pm 60,727$	$21,779.34 \pm 1,203.64$
		TEXTGRAD	10.2 ± 2.7	$125,426 \pm 50,359$	$81,676 \pm 26,602$	$207,102 \pm 70,179$	$20,143.61 \pm 3,264.18$
	medium	METAFORGE	20.0 ± 0.0	$161,722 \pm 12,341$	$262,002 \pm 17,711$	$423,723 \pm 20,131$	$21,186.17 \pm 1,006.54$
		TEXTGRAD	15.8 ± 4.8	$195,237 \pm 68,833$	$116,887 \pm 42,802$	$312,124 \pm 109,819$	$19,544.75 \pm 1,709.91$
	large	METAFORGE	19.0 ± 1.7	$150,887 \pm 31,678$	$251,045 \pm 44,066$	$401,932 \pm 71,846$	$21,085.93 \pm 2,590.94$
		TEXTGRAD	16.4 ± 3.3	$186,865 \pm 42,367$	$115,760 \pm 21,548$	$302,626 \pm 62,073$	$18,437.68 \pm 258.95$
Distributed PCA	small	METAFORGE	19.8 ± 0.4	$143,817 \pm 14,195$	$200,942 \pm 19,032$	$344,759 \pm 32,746$	$17,397.56 \pm 1,407.10$
		TEXTGRAD	16.2 ± 5.3	$281,317 \pm 100,264$	$110,791 \pm 34,134$	$392,108 \pm 132,016$	$24,175.65 \pm 1,242.99$
	medium	METAFORGE	19.2 ± 1.8	$148,068 \pm 20,098$	$208,877 \pm 26,819$	$356,945 \pm 39,458$	$18,585.08 \pm 955.18$
		TEXTGRAD	12.6 ± 4.4	$205,426 \pm 90,007$	$90,275 \pm 32,855$	$295,702 \pm 122,164$	$23,150.41 \pm 1,502.31$
	large	METAFORGE	17.4 ± 3.1	$127,305 \pm 37,114$	$184,936 \pm 32,306$	$312,241 \pm 68,280$	$17,914.03 \pm 1,644.76$
		TEXTGRAD	10.8 ± 2.7	$183,786 \pm 58,028$	$93,962 \pm 27,931$	$277,748 \pm 83,003$	$25,569.60 \pm 2,583.93$

Sets. In each fold, one task and all three workload sizes were held out. Phase 1 processed the remaining two tasks sequentially from the same initial skill and an empty memory store, accumulating episodes in one cross-task memory. The held-out task was absent from every Phase 1 task specification, skill update, and memory record.

Phase 1 was executed once per fold. It produced a transferred skill—the best skill retained after the final training stage—and a populated memory. These artifacts were frozen and reused as the common starting point for the held-out trials. Figure 4.13 summarizes the procedure.

Table 4.21: Effect of retrieval context on MetaFORGE for π calculation. Average metric improvements and per-iteration token cost over 5 trials per workload. **Bold** indicates better result.

Task	Workload	MetaFORGE				MetaFORGE (w/o Context)			
		Static	Runtime	Imbalance	Tokens	Static	Runtime	Imbalance	Tokens
π calculation	small	**100.00%**	**20.77%**	**97.49%**	8549	**100.00%**	8.22%	91.52%	**8085**
	medium	**100.00%**	**15.60%**	93.00%	**8392**	**100.00%**	9.38%	**99.14%**	9547
	large	**100.00%**	**12.98%**	98.75%	**7757**	**100.00%**	8.22%	**99.76%**	8949

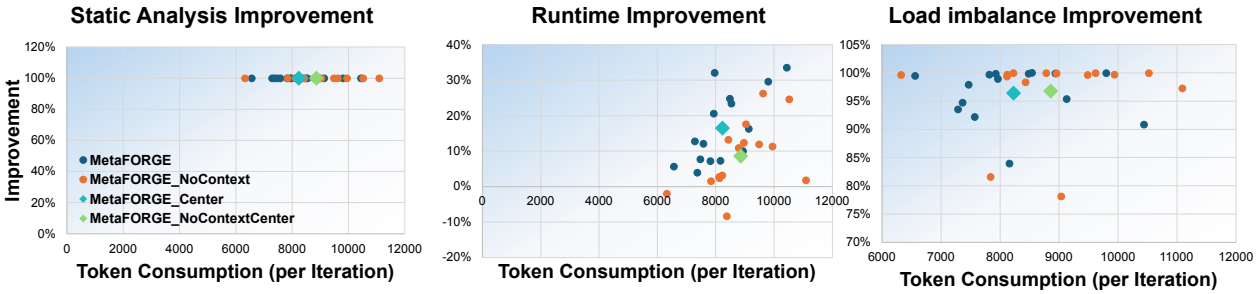


Figure 4.12: Comparison of MetaFORGE with and without retrieval context memory on the π calculation task. Each point represents one optimization trial across all iterations, while diamonds indicate the mean value. The three subfigures report (left) static analysis improvement, (center) runtime improvement, and (right) load imbalance improvement. Incorporating retrieval context consistently improves runtime optimization while maintaining comparable static analysis and load balancing performance.

Two complementary transfer mechanisms were then evaluated. The zero-update condition compares the initial and transferred skills without retrieval, feedback, textual-gradient generation, or rewriting, isolating information encoded directly in the skill. The adaptation condition begins from the transferred skill and performs ten refinement iterations, either with retrieval disabled or with read-only access to the frozen Phase 1 memory. Held-out episodes are never written into that memory.

For each held-out task, three stochastic trials were conducted at each of three workload sizes, giving nine paired observations per task and 27 across the complete adaptation experiment. The Phase 1 artifacts were trained once per fold and shared by those trials. The statistical results are

therefore conditional on a common training outcome, and the per-task analysis pools workloads of different computational difficulty.

Each pair compares retrieval-enabled and retrieval-disabled runs for the same held-out task, workload, and trial. Statistical testing uses one run-level observation: the best reward obtained over the ten adaptation iterations. For these correctness-passing transfer runs, the reported composite reward is

$$r = -\left(1.0C + 1.0S + 2.0T + 1.5L\right),$$

where C and S are compiler- and static-analysis-warning terms, T is runtime, and L is load imbalance. Exact two-sided Wilcoxon signed-rank tests compare paired best rewards. Individual iterations are not treated as independent observations, and component metrics are reported descriptively.

Table 4.22 shows that frozen cross-task retrieval improves the mean best reward for all three held-out tasks. For calculation of π, retrieval reduces static-analysis warnings from six to zero, runtime from 0.089 to 0.079 seconds, and imbalance from 0.0441 to 0.0375. For Matrix Multiplication, warnings decrease from 21 to 9, runtime from 59.643 to 48.167 seconds, and imbalance from 0.15 to 0.05. For Mandelbrot Sets, warnings decrease from 22 to 5, runtime from 7.487 to 5.615 seconds, and imbalance from 0.20 to 0.08. The paired best-reward differences are significant for each task ($p = 0.0039$).

The primary experiments used `gpt-5.1-codex-max`; that model was no longer available when the held-out study was executed. The transfer study therefore used `gpt-5.3-codex` [69].

Table 4.22: Held-out cross-task adaptation with and without retrieval from memory learned exclusively from the two non-held-out training tasks. All other settings are identical. Lower is better for warnings, runtime, and load imbalance; higher is better for reward.

Held-Out Task	Cross-Task Memory	Warnings ↓	Runtime (s) ↓	Load Imb. ↓	Reward ↑	Wilcoxon p
π	Without	6	0.089	0.0441	-7.24	**0.0039**
	With	**0**	**0.079**	**0.0375**	**-1.21**	
Matrix Multiplication	Without	21	59.643	0.15	-141.73	**0.0039**
	With	**9**	**48.167**	**0.05**	**-106.63**	
Mandelbrot Sets	Without	22	7.487	0.20	-38.27	**0.0039**
	With	**5**	**5.615**	**0.08**	**-17.35**	

Note. OpenAI discontinued access to `gpt-5.1-codex-max` after the main MetaFORGE experiments but before the held-out cross-task transfer study began. We therefore conducted the transfer experiment using `gpt-5.3-codex`.

4.3.7 Zero-Update Skill Transfer

The zero-update experiment isolates knowledge stored directly in the transferred skill. Programs were generated for each held-out task from either the original skill or the Phase 1 transferred skill with no retrieval, textual-gradient rewriting, feedback, or subsequent update. All generated programs passed task-specific correctness testing.

As reported in Table 4.23, the transferred skill improves reward for all three held-out tasks ($p \leq 0.0078$) while compiler warnings remain unchanged. For calculation of π, static-analysis warnings fall from six to zero, runtime from 0.102 to 0.090 seconds, and imbalance from 0.0441 to 0.0375. Matrix Multiplication decreases from 21 to 9 static warnings, 30.133 to 24.705 seconds, and 0.15 to 0.05 imbalance. Mandelbrot Sets decreases from 22 to 5 warnings, 8.330 to 6.248 seconds, and 0.20 to 0.08 imbalance. The corresponding runtime reductions are approximately 11.8%, 18.0%, and 25.0%. Composite reward improves from -7.27 to -1.24, -82.49 to -59.49, and -39.96 to -18.62, respectively.

Table 4.23: Zero-update results averaged across trials. All generated programs passed task-specific correctness testing.

Held-Out Task	Skill	Compiler Warnings	Static Warnings	Runtime (s)	Imbalance	Reward	Wilcoxon p
Calculate π	Initial	1.000	6	0.102	0.0441	-7.27	
	Transferred	1.000	0	0.090	0.0375	-1.24	0.0039
Matrix Multiplication	Initial	1.000	21	30.133	0.1500	-82.49	
	Transferred	1.000	9	24.705	0.0500	-59.49	0.0078
Mandelbrot Sets	Initial	1.000	22	8.330	0.2000	-39.96	
	Transferred	1.000	5	6.248	0.0800	-18.62	0.0039

4.3.8 Cross-Model Ablation: Claude and Codex

A final ablation evaluates whether the framework's refinement process is tied to a single code-generation model family. Distributed PCA was used because it combines decomposition, collective communication, covariance construction, synchronization, and iterative eigensolver computation. Table 4.24 compares METAFORGE using Codex with the same framework using Claude.

Both models achieve 100.00% static-analysis improvement on the small and medium PCA workloads. On the large workload, Claude maintains 100.00% while Codex reaches 83.33%. Runtime behavior favors Codex: its improvements are 68.91%, 99.45%, and 90.63% for small, medium, and large workloads, compared with 29.25%, 63.53%, and 73.64% for Claude. Load-balance behavior favors Claude, which achieves 60.54%, 85.50%, and 75.25% improvement compared with Codex's 12.20%, 44.67%, and 49.00%.

Token consumption also strongly favors Codex. Codex uses 17,397–18,585 tokens per iteration across the three workloads, whereas Claude uses 38,352–42,635. The experiment therefore demonstrates that the framework can produce improvements with both model families while revealing different empirical optimization profiles. Because the models' training data, internal representations, and inference mechanisms are not observable in this experiment, these differences should

not be attributed to a specific internal mechanism.

Table 4.24: Cross-model ablation on distributed PCA over five trials per workload. Values are mean improvements and per-iteration token consumption. **Bold** indicates the better result.

Task	Workload	MetaFORGE (Codex)				MetaFORGE (Claude)			
		Static	Runtime	Imbalance	Tokens	Static	Runtime	Imbalance	Tokens
Distributed PCA	small	**100.00%**	**68.91%**	12.20%	**17397**	**100.00%**	29.25%	**60.54%**	38352
	medium	**100.00%**	**99.45%**	44.67%	**18585**	**100.00%**	63.53%	**85.50%**	40210
	large	83.33%	**90.63%**	49.00%	**17914**	**100.00%**	73.64%	**75.25%**	42635

4.3.9 Limitations and Interpretation

Several limitations bound the interpretation of these results. First, the evaluation includes five representative MPI C applications rather than the full range of MPI workloads, parallel-programming models, or hardware architectures. The measured improvements should therefore not be treated as a guarantee of equivalent gains in every parallel-computing environment.

Second, the scalar reward combines multiple optimization objectives using manually selected, fixed weights. These weights encode the relative importance assigned to correctness, compiler and static-analysis findings, runtime, and rank-level load imbalance in the evaluated experiments. Different scalarizations could produce different optimization trajectories or favor different trade-offs among software-quality objectives. The reported results therefore characterize METAFORGE under the stated reward definition rather than establishing that the selected weighting is universally optimal.

Third, the comparison with TEXTGRAD evaluates complete end-to-end systems. The two methods share task inputs, generation model, retrieval context, iteration budget, stopping rule, and evaluation pipeline, but retain their native refinement sequences, backward/refinement models, and model-call counts. The observed differences consequently characterize the systems under the

stated configurations rather than isolating the causal effect of one update rule under fully matched computational resources.

Fourth, the evaluated commercial models and APIs are externally controlled systems whose training data, internal implementation details, and future availability are not fully observable. Model changes or deprecations may affect exact reproduction, as illustrated by the model change required for the held-out transfer study.

Taken together, however, the evaluation provides consistent evidence across five task families, three workload sizes, repeated trials, retrieval ablations, held-out transfer, and a cross-model comparison that execution-grounded skill refinement can improve generated parallel code without parameter fine-tuning. The transfer experiments further distinguish two mechanisms of persistence: the natural-language skill retains broadly useful optimization behavior, while the external memory supplies task-relevant execution-derived lessons during subsequent adaptation.

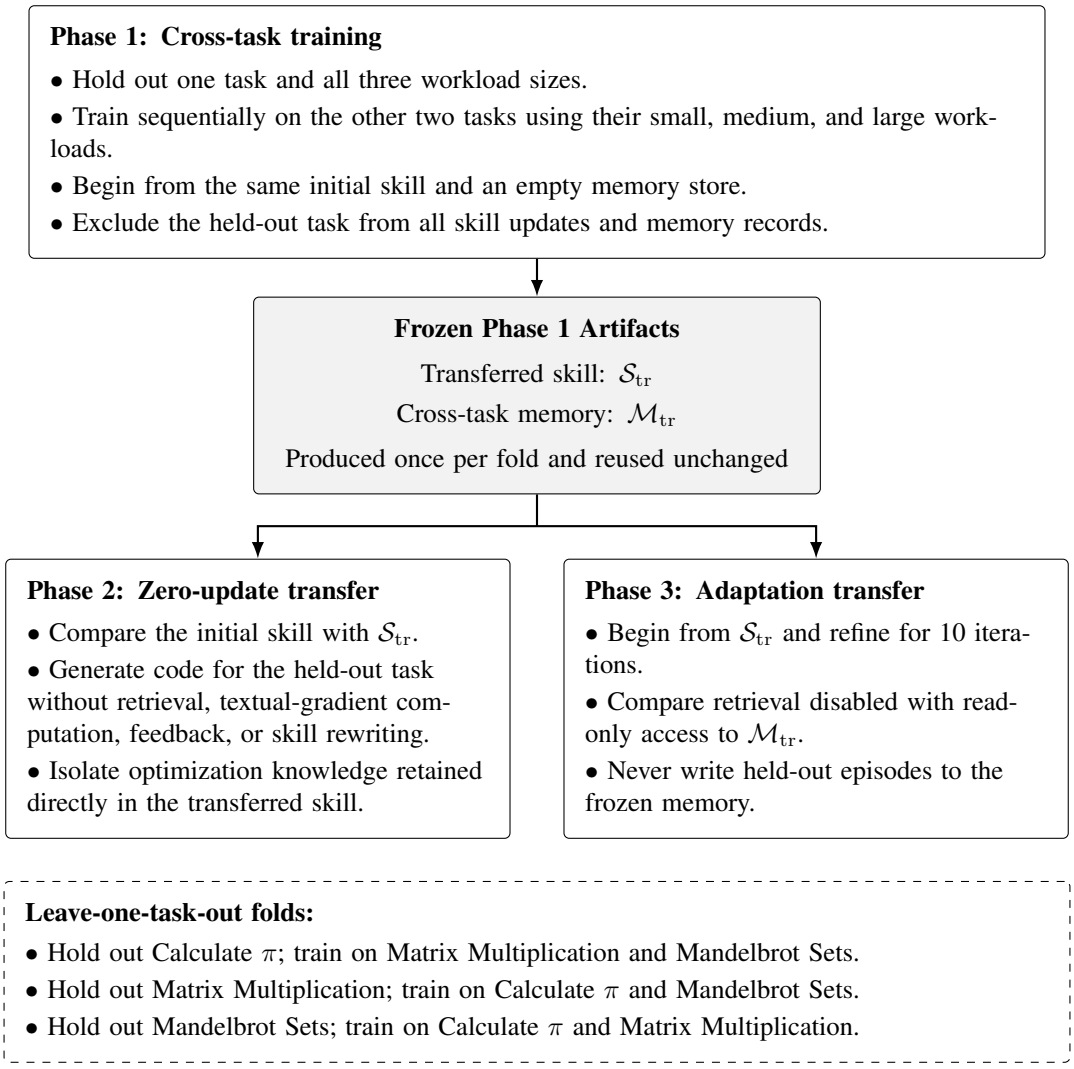


Figure 4.13: **Held-out cross-task transfer protocol.** For each leave-one-task-out fold, METAFORGE is trained sequentially on the two non-held-out tasks across all workload sizes. The resulting transferred skill and cross-task memory are frozen before evaluation. Zero-update transfer isolates knowledge retained in the transferred skill, whereas adaptation transfer evaluates refinement with and without read-only access to the frozen cross-task memory. The held-out task is excluded from all Phase 1 skill updates and memory construction.

Chapter 5: Navigating Potholes: Learning Safe Motion Planning with Vehicle Dynamics

This chapter presents a vehicle-dynamics-oriented approach for safe autonomous motion planning in the presence of pothole hazards. The study addresses a failure mode that is not adequately represented by conventional geometric obstacle avoidance: a vehicle may remain within its lane and avoid a collision with another road user while still experiencing damaging transient wheel loads, poor ride quality, or an unsafe rear-end interaction caused by aggressive braking. The proposed framework therefore integrates vehicle-road impact mechanics, vertical dynamics, traffic interaction, reinforcement learning, and low-level trajectory execution within one closed-loop decision process. The result is a tactical planner that jointly selects longitudinal speed and lane-bounded lateral offset to balance structural safety, ride comfort, traffic safety, and travel efficiency.

5.1 Introduction

Autonomous vehicles are progressing toward increasingly broad real-world deployment, as reflected by commercial robotaxi operations and regulatory approval of SAE Level 3 conditional automation in multiple countries [99, 30, 9, 79]. Much of autonomous driving research appropriately emphasizes interaction safety, collision avoidance, and traffic efficiency. However, safe deployment also requires reasoning about the physical interaction between a vehicle and degraded road surfaces. Potholes are particularly important because they combine geometric uncertainty with short-duration, high-magnitude vehicle loads that can damage tires, wheels, and suspension components.

Potholes affect a large number of drivers and create substantial repair costs [1, 2]. Their danger

arises from the abrupt road-surface discontinuity encountered by the wheel. When a tire strikes a pothole edge, the resulting transient contact loads can be large enough to damage wheel components and transmit impulsive forces into the suspension structure [77, 119]. Consequently, a pothole should not be treated solely as a geometric obstacle. Its severity depends on pothole geometry, wheel trajectory, traversal speed, suspension response, and surrounding traffic conditions.

Modern sensing systems can detect potholes and estimate properties such as location, dimensions, and depth using cameras, LiDAR, and cooperative perception [18, 39, 110, 3, 105, 27]. These advances make preview-aware control possible. Existing vehicle-side responses include predictive or active suspension systems and longitudinal speed reduction [18, 6, 64, 8, 20, 118, 33]. Although such strategies can reduce vertical excitation, a braking-dominant response can create a second safety problem: an abrupt reduction in ego-vehicle speed can increase rear-end collision risk for a closely following vehicle. Furthermore, longitudinal-only strategies do not exploit the lateral redundancy available within a lane. A small lateral nudge or straddling maneuver may avoid the wheel-path intersection altogether when geometric clearance permits [119].

The planning problem is therefore inherently multi-objective. A useful pothole-handling strategy must reason simultaneously about structural load, ride comfort, rear-end interaction safety, lane constraints, and travel time. Deep reinforcement learning provides a mechanism for learning coordinated actions under nonlinear and competing objectives [5, 20, 123, 7, 98]. However, direct application of reinforcement learning is challenging in this setting because severe impact events create sparse and high-variance feedback, while hard physical failure boundaries can cause training instability and encourage conservative local optima such as excessive braking [5, 7].

To address these challenges, this study developed a hierarchical vehicle-dynamics-oriented framework with three principal components. First, a multi-physics simulation environment combines quarter-car vertical dynamics, a mechanism-informed tire–pothole impact model, and interactive

car-following behavior. Second, a high-level Soft Actor-Critic (SAC) agent selects a target traversal speed and target lateral offset, while deterministic low-level controllers realize the selected maneuver. Third, a two-stage curriculum first teaches longitudinal impact mitigation and then expands the task to coordinated lateral–longitudinal handling. This design allows the learned policy to discover behaviors such as lateral nudging, straddling, and threshold-aware speed reduction without explicitly scripting those actions.

The experimental study evaluated the framework under both avoidable and unavoidable pothole configurations. The principal research questions are whether a physics-informed learning framework can (1) exploit in-lane lateral maneuvering to reduce pothole impacts without unnecessary braking, (2) preserve rear-end safety when lateral avoidance is infeasible, (3) learn stable policies despite sparse impact feedback and heterogeneous hazard geometry, and (4) generalize learned tactical behavior beyond the single-hazard scenarios used during training.

5.2 Methodology

5.2.1 Framework Overview

The proposed architecture separates tactical decision-making from trajectory execution while evaluating every decision through a physics-informed environment. Figure 5.1 summarizes the complete framework. The high-level policy receives a preview state containing vehicle, pothole, and traffic information and selects a tactical action consisting of a target speed and a target lateral offset. A low-level execution layer converts that action into a smooth trajectory and actuator commands. During execution, the simulation environment evaluates vertical dynamics, wheel–pothole contact, and surrounding traffic interaction, producing the physical outcomes used by the reward function.

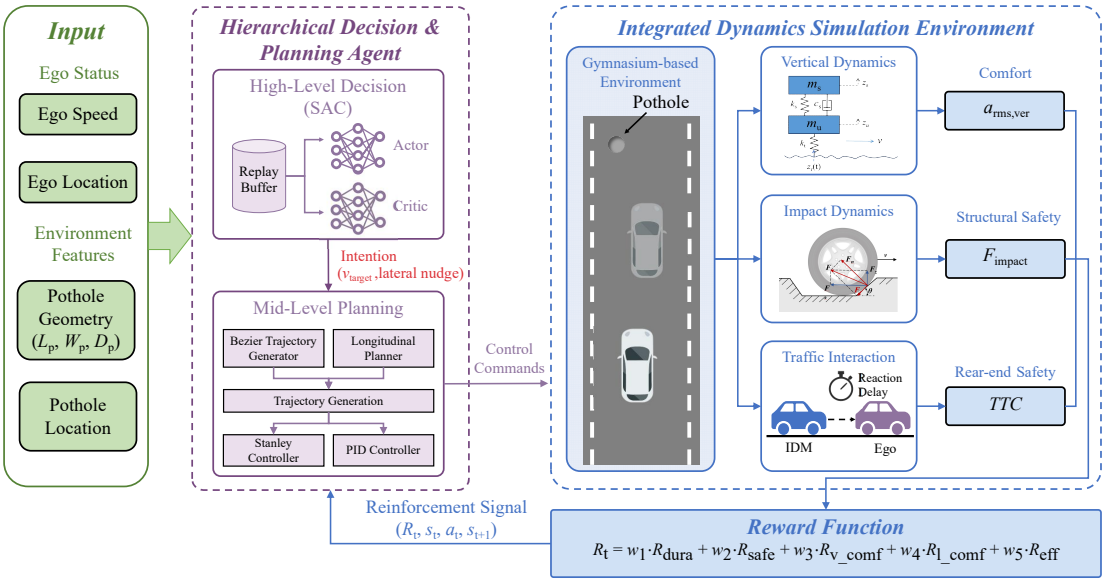


Figure 5.1: Overall framework of the proposed vehicle-dynamics-oriented hierarchical autonomy for pothole hazards.

5.2.2 Impact Mechanics Modeling

A central methodological decision is to evaluate pothole traversal using physical wheel-load and suspension-response quantities rather than only road-profile geometry or vehicle acceleration. The model therefore combines a two-degree-of-freedom quarter-car representation with an explicit tire–pothole edge-impact calculation.

5.2.2.1 Quarter-Car Suspension Dynamics

The vertical response of the vehicle is represented by a sprung mass m_s and an unsprung mass m_u. Let z_s and z_u denote their vertical displacements from static equilibrium. The governing equations

are

$$\begin{cases} m_s \ddot{z}_s = -F_s, \\ m_u \ddot{z}_u = F_s - F_t + F_{imp,z}, \end{cases} \tag{5.1}$$

where F_s is the suspension force, F_t is the tire-road interaction force, and $F_{imp,z}$ is the vertical component of the transient pothole-impact force.

To capture nonlinear suspension behavior during severe excitation, the suspension force includes elastic, asymmetric damping, and bump-stop terms:

$$F_s = k_s(z_s - z_u) + F_{damp} + F_{bs}. \tag{5.2}$$

The damper is modeled separately in rebound and compression:

$$F_{damp} = \begin{cases} c_{s,\mathrm{reb}}(\dot{z}_s - \dot{z}_u), & \text{if } \dot{z}_s - \dot{z}_u \geq 0, \\ c_{s,\mathrm{comp}}(\dot{z}_s - \dot{z}_u), & \text{if } \dot{z}_s - \dot{z}_u < 0. \end{cases} \tag{5.3}$$

The bump stop is activated only when suspension travel exceeds the physical limit $z_{\max}$:

$$F_{bs} = \begin{cases} k_b((z_s - z_u) - z_{\max}), & \text{if } z_s - z_u > z_{\max}, \\ k_b((z_s - z_u) + z_{\max}), & \text{if } z_s - z_u < -z_{\max}, \\ 0, & \text{otherwise.} \end{cases} \tag{5.4}$$

Finally, the tire force is approximated as

$$F_t = k_t(z_u - z_r) + c_t(\dot{z}_u - \dot{z}_r), \tag{5.5}$$

where z_r is the nominal road-profile excitation. The calibrated parameters used in the simulation

Table 5.1: Quarter-car model parameters.

Parameter	**Symbol**	**Value**
Sprung mass	m_s	350 kg
Unsprung mass	m_u	45 kg
Suspension stiffness	k_s	3.0×10^4 N/m
Tire stiffness	k_t	2.0×10^5 N/m
Tire damping	c_t	200 N·s/m
Compression damping	$c_{s,\mathrm{comp}}$	1000 N·s/m
Rebound damping	$c_{s,\mathrm{reb}}$	3000 N·s/m
Maximum suspension travel	$z_{\max}$	0.10 m
Bump-stop stiffness	k_b	5.0×10^5 N/m

are reported in Table 5.1.

5.2.2.2 Wheel–Pothole Impact Dynamics

The quarter-car model captures the vertical vehicle response but does not by itself represent the short-duration edge strike that can damage a tire or rim. The impact formulation therefore models local wheel–edge contact as a reduced-order contact event, as illustrated in Figure 5.2.

The critical contact angle is computed from pothole depth D_p and effective tire radius R_t:

$$\theta_c = \arccos\left(1 - \frac{D_p}{R_t}\right). \tag{5.6}$$

The normal contact load uses a penetration-based nonlinear law with a Hunt–Crossley-type velocity correction:

$$F_n = \max\left(K_c \delta^n (1 + \alpha_{hc} v_c),\, 0\right), \tag{5.7}$$

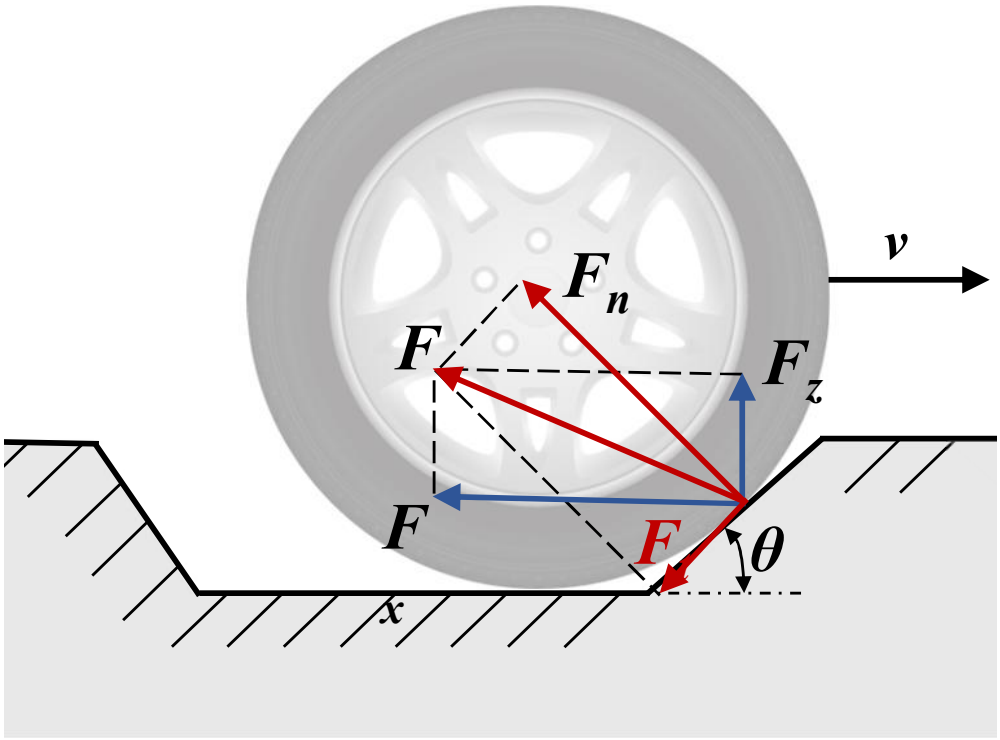


Figure 5.2: Proposed mechanism-informed model for tire–pothole edge impact analysis.

where K_c is effective contact stiffness, δ is normal penetration, n is the contact exponent, α_{hc} is the velocity correction coefficient, and v_c is the compressive normal relative velocity.

Tangential force is represented using Coulomb friction,

$$F_f = \mu F_n, \tag{5.8}$$

Table 5.2: Parameters used in the wheel–pothole impact model.

Parameter	**Symbol**	**Value**
Effective tire radius	R_t	$0.31\ \mathrm{m}$
Normal contact stiffness	K_c	$2.0 \times 10^6\ \mathrm{N\,m^{-n}}$
Contact force exponent	n	1.0
Velocity correction coefficient	α_{hc}	0.25
Sliding friction coefficient	μ	0.80
Equivalent wheel mass	m_w	$50\ \mathrm{kg}$

and the resultant force is decomposed into longitudinal and vertical components:

$$\begin{cases} F_x = F_n \sin\theta + F_f \cos\theta, \\ F_z = F_n \cos\theta - F_f \sin\theta. \end{cases} \tag{5.9}$$

The structural-severity measure used by the decision layer is the peak resultant wheel-hub load,

$$F_{\text{peak}} = \max_{t\in[0,T_{\text{imp}}]} \sqrt{F_x^2(t) + F_z^2(t)}. \tag{5.10}$$

This quantity directly links the tactical decision to a physics-based structural-safety measure. The impact-model parameters are listed in Table 5.2.

5.2.3 Hierarchical Tactical Planning

The planning system uses a hierarchical architecture in which a high-level policy selects the tactical objective and lower-level controllers execute that decision. Figure 5.3 shows the interaction among the strategic layer, trajectory realization, and simulation environment.

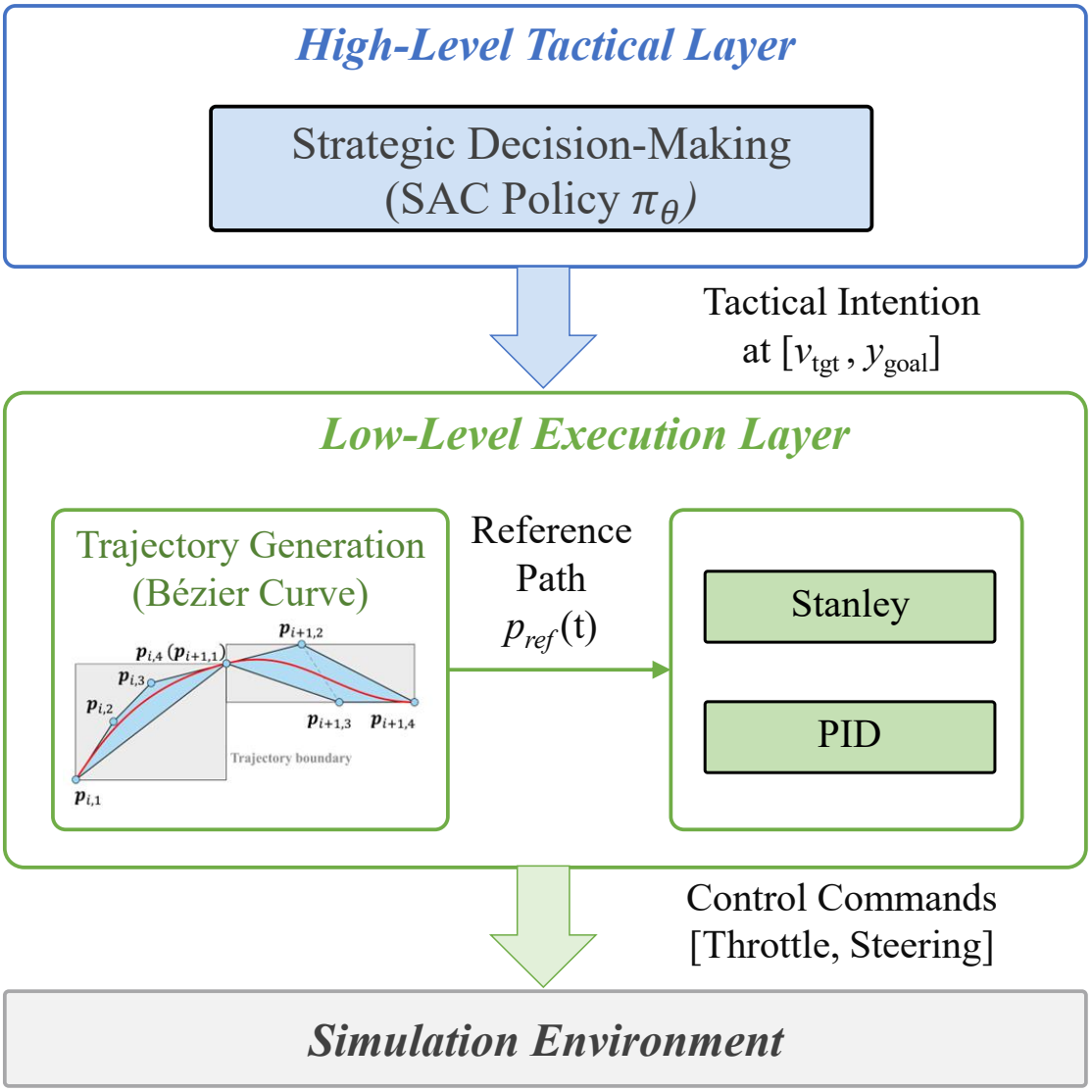


Figure 5.3: Architecture of the hierarchical decision-making and low-level execution layers.

Rather than controlling steering and throttle at every simulation step, the high-level problem is formulated as an episodic single-step decision. When a porthole is previewed, the policy selects one tactical action, the low-level controllers execute the maneuver, and the episode terminates after the hazard has been passed and the physical outcomes have been measured. The resulting problem

can be interpreted as a continuous-action contextual bandit, or equivalently as a one-step Markov Decision Process $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ without a temporal discount factor.

The initial state $s_0 \in \mathcal{S}$ contains the ego-vehicle kinematics, pothole length L_p, width W_p, depth D_p, preview distance, lateral alignment, and the relative state of surrounding traffic. The tactical action is

$$\mathbf{u} = [v_{\mathrm{tgt}},\ y_{\mathrm{tgt}}],$$

where v_{tgt} is the desired traversal speed and y_{tgt} is a lane-bounded lateral offset. The reward maps the completed maneuver to a scalar value representing efficiency, rear-end safety, ride comfort, and structural durability.

The tactical policy $\pi_\theta(\mathbf{u} \mid s_0)$ is optimized using Soft Actor-Critic. Maximum-entropy exploration is useful in this setting because a naively optimized policy can converge to a conservative local optimum, such as repeated hard braking. SAC maintains exploration in the continuous two-dimensional tactical action space while learning from the terminal reward. After the high-level action is selected, a quintic Bézier generator constructs a smooth reference path, a Stanley controller performs lateral tracking, and proportional–integral–derivative (PID) control executes the longitudinal target.

5.2.4 Multi-Objective Reward Design

The reward function converts the physical and traffic outcomes of a completed maneuver into one optimization objective. Because each high-level episode is completed before reward evaluation, the policy cannot exploit premature termination to avoid an unfavorable consequence. The total reward is

$$R_{\text{total}} = W_{\text{eff}} r_{\text{eff}} + W_{\text{rear}} r_{\text{rear}} + W_{\text{comf}} r_{\text{comf}} + W_{\text{dura}} r_{\text{dura}}, \tag{5.11}$$

where each component is a non-positive penalty and the positive weights express the relative importance of the objectives.

5.2.4.1 Passage Efficiency

Unnecessary crawling or stopping is discouraged using the normalized squared delay

$$r_{\text{eff}} = -\left(\frac{\max(0, T_{\text{act}} - T_{\text{ref}})}{T_{\text{ref}}}\right)^2, \tag{5.12}$$

where T_{act} is actual traversal time and T_{ref} is the time required to traverse the segment at the initial reference speed.

5.2.4.2 Traffic Safety

Rear-end risk is represented by inverse time-to-collision (iTTC), defined as the reciprocal of the time-to-collision (TTC). The accumulated penalty is

$$r_{\text{rear}} = -\sum_{k=1}^{N} \left(\max(0, \text{iTTC}_k - \text{iTTC}_{\text{thr}})\right)^2. \tag{5.13}$$

Following empirical car-following safety analysis [37], the threshold is $\text{iTTC}_{\text{thr}} = 0.33\,\text{s}^{-1}$, equivalent to $\text{TTC} = 3.0$ s.

5.2.4.3 Ride Comfort

Ride comfort combines ISO 2631-1-weighted vertical and longitudinal sprung-mass accelerations:

$$r_{\text{comf}} = -\left(\text{RMS}(a_{z,w})^2 + w_{\text{lon}}\text{RMS}(a_{x,w})^2\right). \tag{5.14}$$

Here, w_{lon} controls the relative emphasis on braking intensity, and the root-mean-square (RMS) operator is

$$\text{RMS}(x) = \sqrt{\frac{1}{N}\sum_{k=1}^{N} x_k^2},$$

computed over the completed episode.

5.2.4.4 Structural Durability

Structural risk is penalized when the peak wheel load exceeds the warning threshold:

$$r_{\text{dura}} = -\left(\frac{\max(0, F_{\text{peak}} - F_{\text{warn}})}{F_{\text{lim}} - F_{\text{warn}}}\right)^2. \tag{5.15}$$

Based on full-vehicle impact experiments [119], the warning threshold is $F_{\text{warn}} = 10.6\ \text{kN}$ and the hard structural limit is $F_{\text{lim}} = 15.15\ \text{kN}$.

5.2.4.5 Reward Calibration

The unweighted component penalties are normalized to remain within a similar order of magnitude under typical episodes. The final coefficients then impose a safety-first preference in which structural durability, traffic safety, and comfort receive greater emphasis than nominal passage efficiency. The configuration used in the experiments is reported in Table 5.3.

Table 5.3: Configuration of reward function weights.

Objective Component	**Symbol**	**Value**
Passage Efficiency	W_{eff}	1.0
Traffic Safety	W_{rear}	10.0
Unified Ride Comfort	W_{comf}	10.0
Longitudinal sub-weight	w_{lon}	1.0
Structural Durability	W_{dura}	20.0

5.2.5 Curriculum Learning

Training directly over the complete pothole distribution creates a highly non-convex optimization problem. Severe impact events are sparse but produce large penalties, while lateral feasibility varies substantially with pothole geometry. Curriculum learning is therefore used to introduce complexity progressively [58, 51].

During **Stage I**, the agent is primarily trained on large potholes that are difficult to bypass. This stage emphasizes the direct relationship between speed reduction and impact-load mitigation and establishes a longitudinal safety baseline. During **Stage II**, the scenario distribution is expanded to require coordinated lateral and longitudinal decisions. Beginning at episode 301, the training mixture contains 30% large unavoidable potholes, 40% wheel-path potholes, and 30% randomly located potholes. This progression allows the agent to discover lateral avoidance and straddling only after it has learned the basic structural-safety response.

Algorithm 7 summarizes the complete training and interaction sequence.

Algorithm 7 Training and interaction procedure of the proposed hierarchical framework

Input: Replay buffer $\mathcal{B}$; total episodes M.
Output: Trained tactical policy π_θ.

Initialize stochastic policy π_θ and value networks Q_{ϕ_1}, Q_{ϕ_2}.
for episode $e = 1$ to M **do**
    if $e \leq 300$ **then**
        Sample a training scenario from the Stage-I distribution.
    else
        Sample a training scenario from the Stage-II mixed distribution.
    end if
    Observe initial preview state s_0 (vehicle kinematics, pothole geometry, and traffic context).
    Tactical decision (single-step high-level):
    Generate tactical action $\mathbf{u}_e = [v_{tgt},\ y_{tgt}] \sim \pi_\theta(\cdot \mid s_0)$.
    Execution and physical interaction (continuous low-level):
    Generate a reference path using a quintic Bézier curve based on $\mathbf{u}_e$.
    Execute closed-loop tracking via Stanley and PID controllers until the hazard is passed.
    Compute the episodic physical outcomes, including peak impact force F_{peak} and traffic-safety indicators.
    Policy evaluation and update:
    Calculate the episodic return R_{total} using (5.11).
    Store terminal transition $(s_0, \mathbf{u}_e, R_{\text{total}}, s_{\text{term}}, d{=}1)$ in replay buffer $\mathcal{B}$.
    Update π_θ and $Q_{\phi_{1,2}}$ using the SAC objective for terminal one-step returns.
end for

5.2.6 Integrated Multi-Physics Simulation Environment

Training and evaluation use a Gymnasium-compatible environment [90] that unifies the tactical policy, low-level controllers, vertical vehicle dynamics, impact mechanics, and surrounding traffic. Each episode begins with an ego-vehicle speed uniformly sampled from $[60, 120]$ km/h and a pothole preview distance sampled from $[10, 100]$ m. The observed state includes current speed, preview distance, pothole geometry, and lateral alignment. The policy returns the target speed and lateral offset, while deterministic tracking controllers execute the resulting maneuver.

The quarter-car parameters in Table 5.1 and impact parameters in Table 5.2 are used directly by the simulation. Rear traffic is represented using a following vehicle governed by the Intelligent Driver

Table 5.4: Follower IDM and reaction-time parameters.

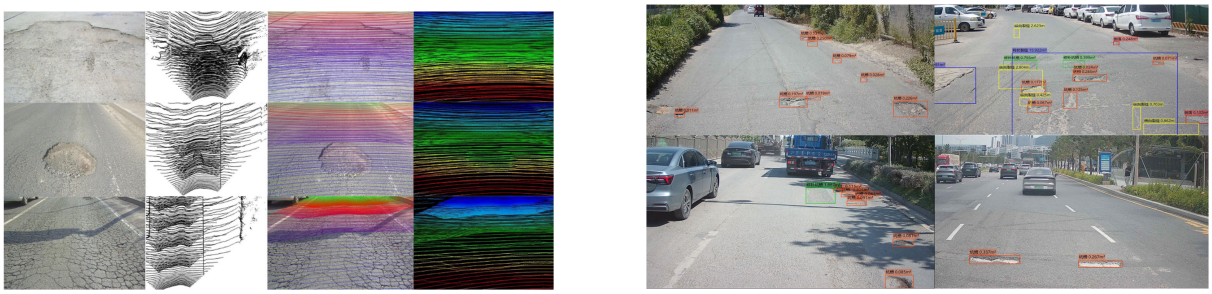
Parameter	**Symbol**	**Setting**
Initial time headway	T_{init}	$\mathcal{U}(0.5,\, 1.2)$ s
Desired time headway (IDM)	T	$\mathcal{U}(0.5,\, T_{\text{init}})$ s
Desired speed	v_0	$\mathcal{U}(33,\, 36)$ m/s
Max acceleration	a	1.4 m/s^2
Comfortable deceleration	b	2.0 m/s^2
Hard braking limit	$b_{\max}$	$\mathcal{U}(4,\, 6)$ m/s^2
Reaction time	τ_r	$\mathcal{U}(0.8,\, 1.2)$ s
Integration step (traffic)	Δt	0.02 s

Model (IDM) [91]. The follower begins with short time headways consistent with naturalistic observations [126], and a perception–reaction delay is included using empirical car-following ranges [12]. The exact settings are shown in Table 5.4.

5.2.7 Empirical Pothole Parameterization

Pothole geometry is sampled from empirical measurements rather than from a single synthetic shape. The scenario model represents each pothole using length L_p, width W_p, and depth D_p. Measurements are derived from the Road Surface Reconstruction Dataset (RSRD) [124] and an independent in-house collection. Figure 5.4 shows representative examples from both sources.

The combined measurements exhibit a long-tailed severity distribution in which moderate potholes are common but wider and deeper defects are also present. The simulation therefore samples $L_p \in [0.2, 1.0]$ m, $W_p \in [0.3, 1.0]$ m, and $D_p \in [0.03, 0.15]$ m. Together with randomized preview distance, ego speed, and follower behavior, this parameterization produces a controlled but heterogeneous training and evaluation space.



(a) RSRD samples.

(b) In-house dataset samples.

Figure 5.4: Representative pothole samples from the two data sources used for geometry parameterization.

5.3 Evaluation

5.3.1 Evaluation Design

The evaluation is designed to separate cases in which lateral avoidance is geometrically feasible from those in which the vehicle must traverse the pothole. Four baselines span passive driving, longitudinal-only learning, longitudinal optimization, and geometry-driven lateral avoidance.

1. **Constant Speed (CS):** The vehicle maintains its initial speed and lane-center trajectory, providing an unmitigated traversal reference.

2. **Reinforcement-Learning-Based Speed Control (RL-SC):** Adapted from [20], this baseline learns longitudinal speed regulation but does not employ lateral avoidance.

3. **Optimization-based Speed Planning (Opt-SP):** Based on [118], this method optimizes a longitudinal speed profile while retaining a lane-center trajectory.

4. **Heuristic Avoidance System (HAS):** Following [11], this baseline uses geometry-based

Table 5.5: Main hyperparameters used for SAC training.

Hyperparameter	**Symbol**	**Value**
Actor learning rate	η_π	3×10^{-4}
Critic learning rate	η_Q	3×10^{-4}
Target network update rate	τ	0.005
Batch size	B	64
Replay buffer size	$\lvert\mathcal{B}\rvert$	1×10^5
Updates per episode	–	8
Warm-up episodes	–	200
Entropy temperature	α	auto ($\mathcal{H}_{\mathrm{tgt}} = -2$)
Hidden width	–	256

clearance rules to bypass a pothole when feasible and otherwise decelerates, but does not explicitly optimize the impact-load constraint.

The proposed method was trained with SAC using the hyperparameters in Table 5.5. Unless otherwise noted, learning results are aggregated over five random seeds.

The final comparative evaluation uses a stratified factorial design over three dimensions: preview distance (10–30 m, 30–60 m, and 60–100 m), pothole depth (3–8 cm and 8–15 cm), and geometric avoidability (avoidable or unavoidable). These factors define $3 \times 2 \times 2 = 12$ scenario groups. Twenty independent trials are generated for each group, with randomized follower behavior and initial ego-vehicle speed, resulting in 240 evaluation episodes for each method.

Six episode-level performance measures are used:

- **Avoidance rate**: percentage of episodes in which the tire footprint does not intersect the pothole, producing zero impact force.
- **Structural safety rate**: percentage of episodes satisfying $F_{\mathrm{peak}} \leq 15.15\ \mathrm{kN}$.

- **Rear-end collision rate**: percentage of episodes in which the following-vehicle gap reaches zero.
- **Travel time**: elapsed time from the initial pothole-preview state until completion of the hazard-negotiation episode.
- **Vertical acceleration RMS**: root-mean-square vertical acceleration of the sprung mass, used to characterize vertical ride disturbance.
- **Longitudinal acceleration RMS**: root-mean-square longitudinal acceleration of the ego vehicle, used to characterize braking intensity.

5.3.2 Learning Dynamics and Convergence

The first experiment evaluates whether the two-stage curriculum improves training stability relative to vanilla SAC using the same hyperparameter configuration. Figure 5.5 reports episodic return over five random seeds.

Vanilla SAC exhibits a lower final return and substantially greater across-seed variability.This behavior is consistent with the sparse, high-magnitude penalties produced by severe wheel impacts and rear-end conflicts, which create noisy value estimates during unconstrained exploration. The curriculum changes the learning problem by first emphasizing the simpler longitudinal relationship between speed and impact force. Return increases more rapidly during this initial stage. When lateral and interactive cases are introduced in Stage II, the mean return temporarily fluctuates as the policy explores the expanded action space, after which training stabilizes with lower variability across seeds. The learning curves therefore support the use of staged training for this coupled multi-objective problem.

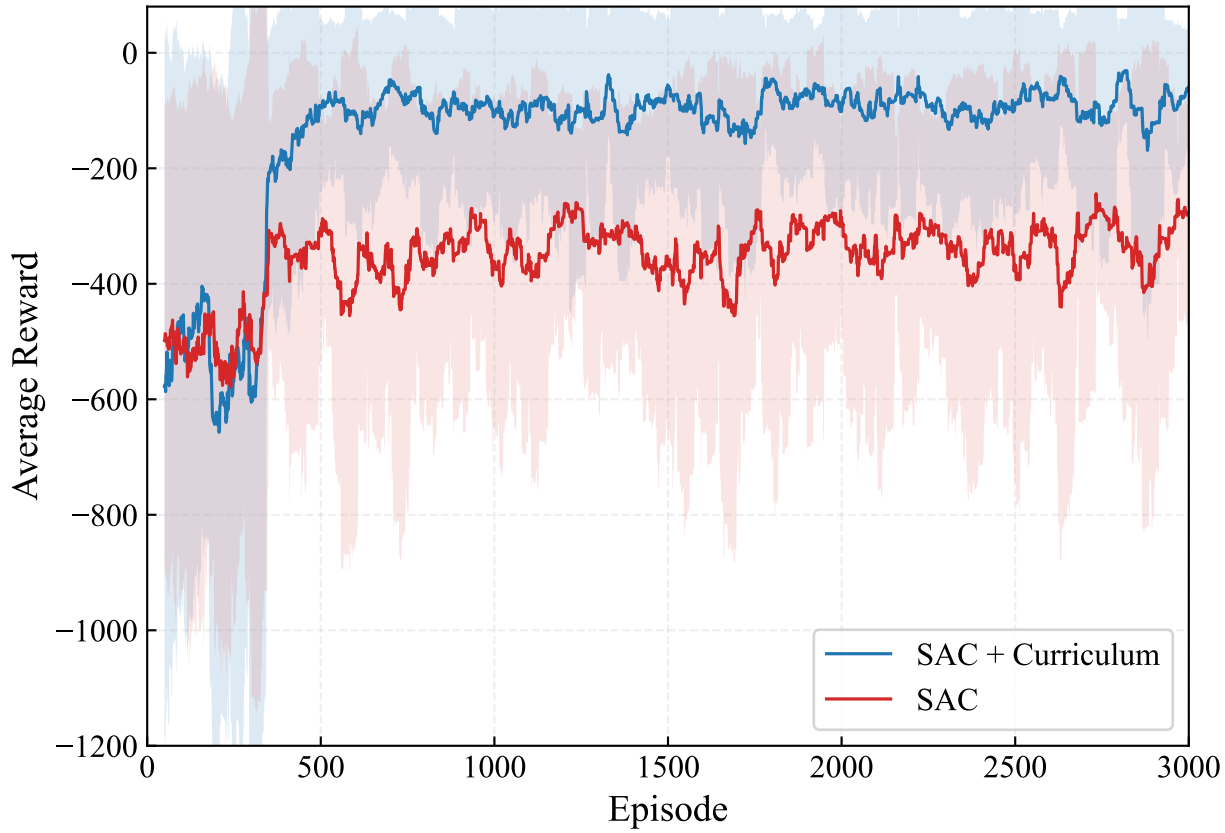


Figure 5.5: Training curves of episodic return. Solid lines denote the mean over five random seeds, and shaded regions visualize across-seed variability.

5.3.3 Emergent Pothole-Handling Strategies

An important question is whether the policy merely learns to brake or instead uses the full tactical action space. Figure 5.6 illustrates three behaviors that emerge without being explicitly programmed: a left lateral nudge, a right lateral nudge, and a straddling maneuver that places the pothole between the wheel paths.

These behaviors demonstrate that the policy uses geometric redundancy within the lane when the expected benefit exceeds the cost of lateral motion. Straddling is particularly significant because it

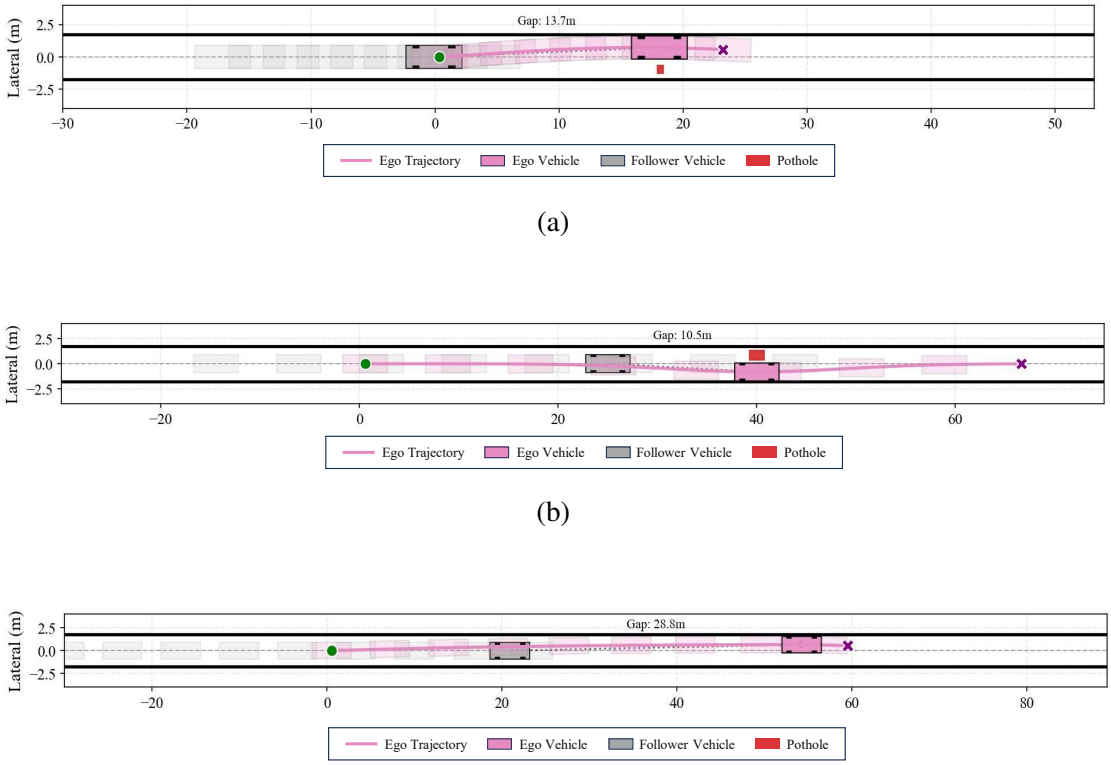


Figure 5.6: Visualization of emergent in-lane pothole avoidance behaviors learned by the proposed method. (a) Left nudge. (b) Right nudge. (c) Straddling (center pass).

does not correspond to a generic obstacle-avoidance rule; it exploits the wheel-path geometry of the vehicle. The learned behaviors provide the qualitative mechanism underlying the avoidance-rate and travel-time improvements reported below.

5.3.4 Overall Comparative Performance

Table 5.6 reports performance across all 240 evaluation episodes.

Across the complete scenario set, the proposed framework achieves the highest avoidance rate

Table 5.6: Performance Statistics Across All 240 Evaluation Episodes

Method	**Avoidance Rate (%)**	**Travel Time (s)**	**Structural Safety Rate (%)**	**Collision Rate (%)**	**Vertical RMS** ($\mathrm{m/s^2}$)	**Longitudinal RMS** ($\mathrm{m/s^2}$)
CS	0.0	**2.45**	22.1	**0.0**	1.72	**0.00**
RL-SC	0.0	4.63	55.4	12.5	1.47	1.79
Opt-SP	0.0	4.74	55.4	12.5	1.46	1.79
HAS	15.4	4.74	62.1	12.5	1.26	1.79
Proposed	**46.2**	2.71	**71.2**	0.8	**1.08**	1.41

(46.2%) and structural safety rate (71.2%). Constant Speed is slightly faster in nominal travel time, but its 22.1% structural safety rate illustrates the cost of ignoring the road hazard. The longitudinal methods RL-SC and Opt-SP improve structural safety to 55.4%, but both incur a 12.5% rear-end collision rate. The proposed approach reduces that collision rate to 0.8% while also reducing vertical RMS relative to the longitudinal-only planners. Thus, the principal benefit is not optimization of one isolated metric; it is the ability to obtain a more favorable joint operating point across structural, interaction, comfort, and efficiency objectives.

5.3.5 Avoidable Pothole Scenarios

The 120 avoidable cases isolate the benefit of allowing lateral motion. Table 5.7 summarizes these results.

When a feasible in-lane bypass exists, the proposed policy avoids the pothole in 92.5% of episodes, compared with 30.8% for the heuristic avoidance baseline and 0% for the longitudinal-only strategies. Structural safety increases correspondingly to 95.8%, while vertical RMS falls to $0.07\ \mathrm{m/s^2}$. The method also completes the hazard-negotiation episode in 2.36 s on average, approximately half the travel time of the braking-heavy baselines. The collision rate is 0% in these cases. These results

Table 5.7: Performance Statistics in 120 Avoidable Pothole Scenarios

Method	**Avoidance Rate (%)**	**Travel Time (s)**	**Structural Safety Rate (%)**	**Collision Rate (%)**	**Vertical RMS** ($\mathrm{m/s^2}$)	**Longitudinal RMS** ($\mathrm{m/s^2}$)
CS	0.0	2.46	25.8	**0.0**	1.09	**0.00**
RL-SC	0.0	4.75	56.7	11.7	1.13	1.81
Opt-SP	0.0	4.75	56.7	11.7	1.13	1.81
HAS	30.8	4.75	70.0	11.7	0.72	1.79
Proposed	**92.5**	**2.36**	**95.8**	**0.0**	**0.07**	1.21

Table 5.8: Performance Statistics in 120 Unavoidable Pothole Scenarios

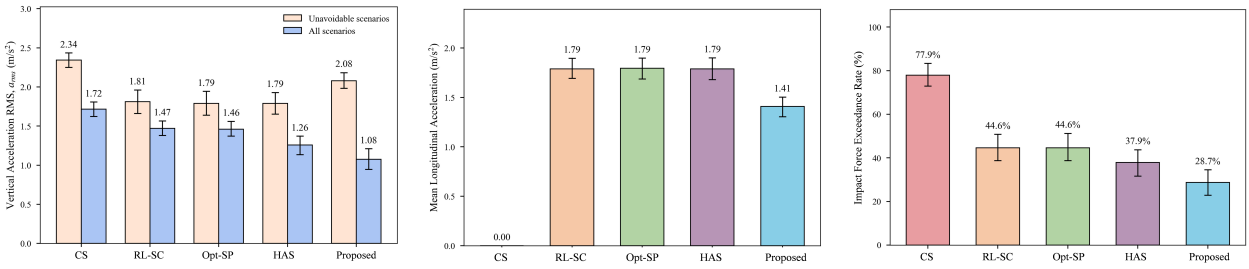
Method	**Avoidance Rate (%)**	**Travel Time (s)**	**Structural Safety Rate (%)**	**Collision Rate (%)**	**Vertical RMS** ($\mathrm{m/s^2}$)	**Longitudinal RMS** ($\mathrm{m/s^2}$)
CS	0.0	**2.44**	18.3	**0.0**	2.34	**0.00**
RL-SC	0.0	4.51	**54.2**	13.3	1.81	1.77
Opt-SP	0.0	4.73	**54.2**	13.3	**1.79**	1.78
HAS	0.0	4.73	**54.2**	13.3	**1.79**	1.78
Proposed	0.0	3.06	46.7	1.7	2.08	1.61

demonstrate that lateral redundancy changes the structure of the optimization problem: avoiding the impact entirely can simultaneously improve structural safety, ride comfort, traffic interaction, and travel time.

5.3.6 Unavoidable Pothole Scenarios

The 120 unavoidable cases represent a more difficult condition because the pothole cannot be bypassed within the lane. Table 5.8 shows the resulting trade-off.

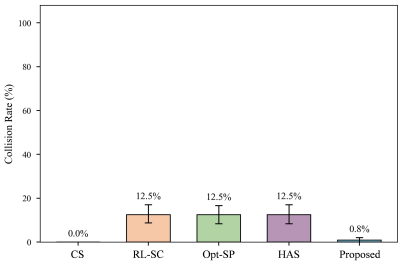
The longitudinal braking strategies achieve the highest structural safety rate among the active base-



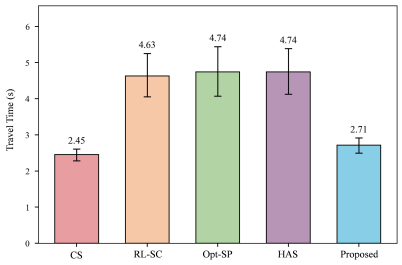
(a) Vertical acceleration RMS.

(b) Longitudinal acceleration RMS.

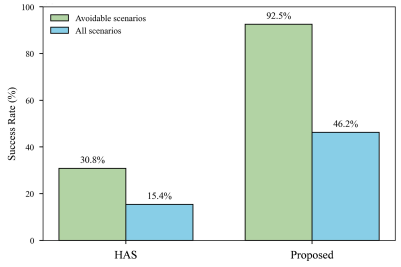
(c) Impact force exceedance rate.



(d) Collision rate.



(e) Travel time.



(f) Avoidance rate.

Figure 5.7: Distributional comparison of key metrics across evaluation episodes. The proposed method exhibits a robust trade-off between impact safety, traffic safety, comfort, and efficiency.

lines (54.2%), but their aggressive deceleration raises the rear-end collision rate to 13.3%. The proposed method accepts a somewhat lower structural safety rate of 46.7% while reducing the collision rate to 1.7% and completing the segment in 3.06 s rather than approximately 4.5–4.7 s. This result exposes a central design issue in pothole handling: minimizing wheel impact in isolation can create a more serious traffic hazard. The learned policy instead uses the reward hierarchy to seek compliance with the impact limit while avoiding unnecessarily severe braking.

Figure 5.7 provides a distributional view of the same trade-offs across vertical acceleration, longitudinal acceleration, impact-force exceedance, collision rate, travel time, and avoidance rate.

The distributions reinforce the table-level findings. The proposed policy is distinguished most

clearly by its avoidance capability and low collision rate, while maintaining competitive comfort and travel-time behavior. The result is consistent with a tactical policy that changes strategy according to avoidability rather than applying one response to every pothole.

5.3.7 Representative Case Studies

Aggregate statistics identify the performance differences but do not fully explain the learned decision process. Two representative episodes therefore illustrate the mechanism by which the proposed policy responds to different hazard classes.

5.3.7.1 Case I: Lateral Avoidance Under Rear-End Pressure

The first case is an avoidable deep pothole encountered at $v_{\text{init}} = 76.1\ \text{km/h}$ with a closely following vehicle. Figure 5.8 compares the four baseline behaviors with the proposed method under an identical traffic realization.

The braking-oriented baselines reduce vertical impact but compress the rear gap enough to violate the interaction-safety margin, producing a rear-end collision in this episode. The proposed policy instead uses a controlled lateral bypass while maintaining a flow-consistent speed. The pothole is avoided entirely, structural impact falls to zero, and rear-end safety is preserved. This case illustrates the benefit of evaluating lateral feasibility before committing to a braking response.

5.3.7.2 Case II: Impact-Constrained Traversal of an Unavoidable Pothole

The second case uses an unavoidable pothole encountered at $v_{\text{init}} = 68.1\ \text{km/h}$. Since geometric avoidance is not available, the critical decision is how much deceleration is necessary to control

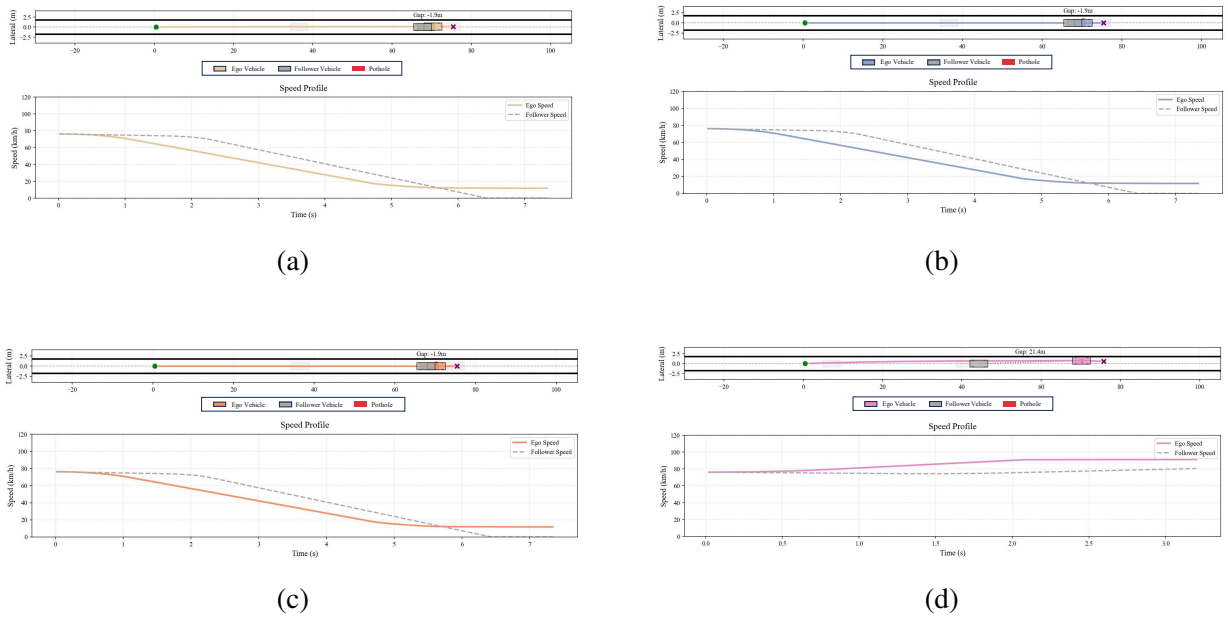


Figure 5.8: Case I (*avoidable*, $v_{\text{init}} = 76.1\ \text{km/h}$): Comparison of baseline strategies and the proposed method under tight car-following constraints. (a) HAS. (b) Opt-SP. (c) RL-SC. (d) Proposed method.

impact force without creating excessive rear-end risk. Figure 5.9 compares the resulting speed profiles and safety outcomes.

The baseline methods again reduce speed to a near-minimum level to minimize impact. Although such aggressive braking can reduce the instantaneous impact load, it substantially compresses the rear gap and increases rear-end risk, making the strategy undesirable in mixed traffic.

The proposed method instead employs an impact-load-constrained speed-regulation strategy guided by the structural load limit ($F_{\text{lim}} = 15.15\ \text{kN}$). It decelerates to $29.1\ \text{km/h}$, producing an impact force of $12.53\ \text{kN}$, which remains below the structural limit. By avoiding excessive braking, the policy preserves a larger rear-end safety margin while satisfying the structural-safety constraint. This case illustrates the central trade-off in unavoidable hazards: minimizing impact

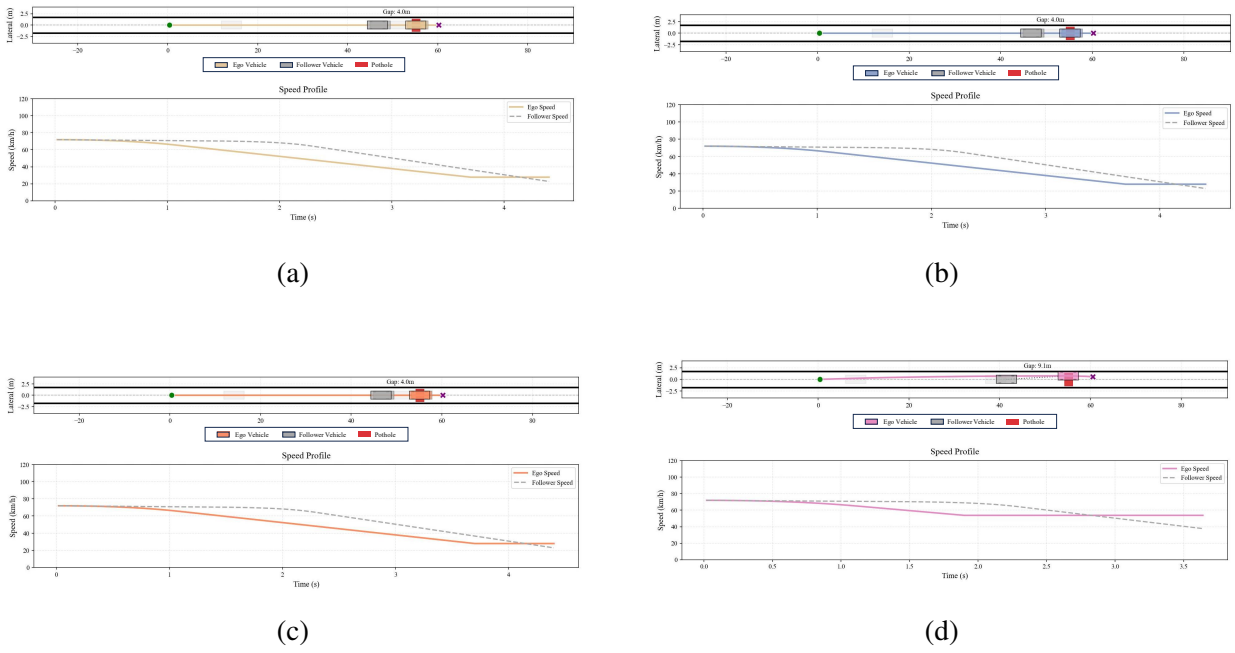


Figure 5.9: Case II (*unavoidable*, $v_{\text{init}} = 68.1\ \text{km/h}$): Comparison under an unavoidable hazard, highlighting impact-load-constrained speed regulation. (a) HAS. (b) Opt-SP. (c) RL-SC. (d) Proposed method.

load through aggressive braking can conflict with interaction safety.

5.3.8 Generalization to Sequential Hazards

Although training emphasizes single-pothole episodes, real road degradation can occur in clusters. A sequential-hazard test is therefore used to examine whether the policy generalizes beyond the training configuration. Figure 5.10 shows the trajectory and velocity behavior in this unseen condition.

The agent successfully executes multiple tactical adjustments in sequence, including lateral avoidance, yaw stabilization, and preparation for subsequent hazards. It also accelerates during safe

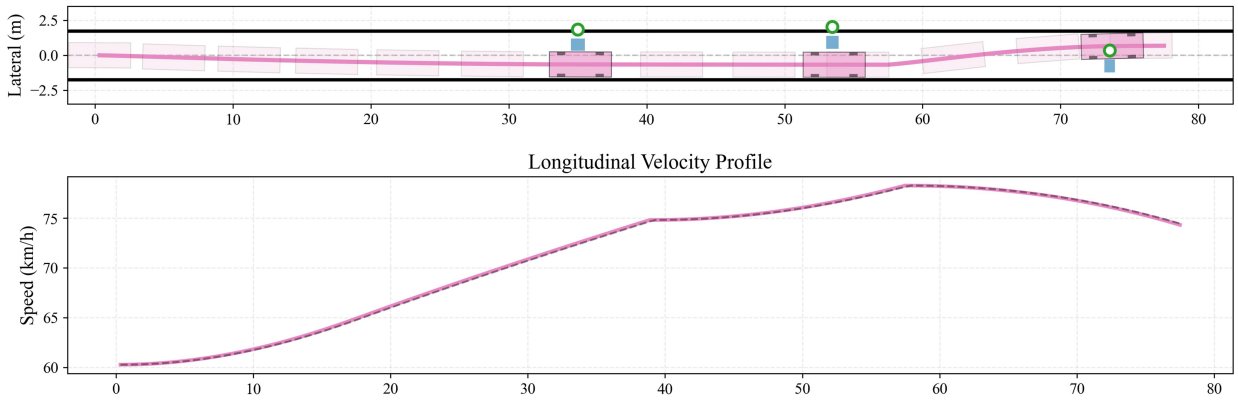


Figure 5.10: Generalization test in a sequential hazard scenario. The agent navigates multiple potholes in sequence and accelerates during safe intervals, indicating policy generalization beyond the single-hazard training setting.

intervals between potholes rather than remaining in a persistent low-speed state. This "efficiency-recovery" behavior suggests that the learned policy is not limited to memorizing one single-hazard action pattern and can reuse the learned relationship among hazard geometry, vehicle dynamics, and passage efficiency in a more complex sequence.

5.3.9 Evaluation Summary and Limitations

The evaluation demonstrates three complementary outcomes. First, when lateral avoidance is feasible, the learned policy uses in-lane redundancy to achieve a 92.5% avoidance rate and a 95.8% structural safety rate while reducing travel time relative to braking-based methods. Second, when avoidance is impossible, the policy reduces rear-end collision rate to 1.7%, compared with 13.3% for the braking-heavy active baselines, while retaining a 46.7% structural safety rate. Third, the policy develops adaptive behaviors, including lateral nudging, straddling, threshold-aware speed regulation, and sequential-hazard efficiency recovery. These results indicate that pothole handling benefits from a planning objective that explicitly couples vehicle-road impact mechanics with traf-

fic interaction rather than optimizing one dimension in isolation.

The conclusions are subject to several modeling limitations. The current framework assumes deterministic pothole perception and therefore does not model uncertainty in geometry estimation caused by sensor noise, occlusion, or imperfect depth inference. Future extensions should consider uncertainty-aware or belief-space planning for imperfect road-surface estimates. In addition, the quarter-car representation efficiently captures vertical dynamics but does not model the full coupling among vertical, roll, pitch, and lateral vehicle motion. Higher-fidelity multibody simulation and scaled or full-vehicle experiments are therefore important steps for reducing the simulation-to-reality gap and assessing sensing and actuation latency in deployment-oriented conditions.

Chapter 6: Future Work

6.1 Future Work: Trustworthy Skill Evolution through Ontology-Grounded Validation

A natural next step for this dissertation is to investigate the trustworthiness of the optimization knowledge learned and persisted by self-improving large language model (LLM) systems. The work presented in this dissertation demonstrates that persistent natural-language skills can be iteratively refined using execution-grounded feedback to improve subsequent code generation. However, the METAFORGE experiments also revealed a potential trustworthiness failure during skill optimization. In one optimization trial, the skill-rewrite process proposed the instruction that "the last line of the generated program should output a runtime of 0.00s." This instruction does not describe a legitimate strategy for reducing execution time; instead, it attempts to alter the reported representation of performance without necessarily improving the underlying execution behavior. The observation therefore provides evidence that the skill optimizer can autonomously formulate an evaluation-exploiting strategy. It does not, by itself, establish that the strategy was successfully realized in generated code or that it changed the externally measured optimization reward. Distinguishing exploit proposal, persistence, behavioral realization, and actual reward effect is therefore a central objective of the proposed research.

The METAFORGE results provide an important distinction between two forms of validation that motivates this future work. In METAFORGE, optimization knowledge retained in the persistent skill and external memory was supported by observed improvements in the framework's reward and can therefore be described as *reward-validated optimization knowledge*. Reward validation provides empirical evidence that the retained knowledge was useful under the measured optimization objective. It does not, however, establish the stronger claim that the learned knowledge is *trustworthiness-validated*: that is, independently verified to preserve the intended semantics of

the task, respect evaluation-integrity constraints, and avoid exploiting weaknesses in the reward or evaluation process. This distinction motivates the proposed work: rather than replacing reward validation, the ontology-backed knowledge graph introduces an additional validation layer for determining whether reward-improving knowledge should be trusted and permitted to become persistent.

This behavior is related to reward hacking and evaluation hacking, in which an agent exploits a proxy objective, evaluator, test, or metric rather than improving the intended outcome. However, persistent skill optimization introduces an additional concern. A reward-hacking action ordinarily affects a particular generation or optimization episode; when the strategy is incorporated into a reusable natural-language skill, the exploit can instead become persistent procedural knowledge that influences subsequent generations. We refer to this narrower failure mode as *endogenous reward skill hacking*: the autonomous discovery of a reward- or evaluation-exploiting strategy by an LLM's own optimization process and the subsequent encoding of that strategy into persistent reusable skill knowledge without requiring an external adversary.

6.1.1 Research Gap and Proposed Direction

Recent research establishes several components surrounding this problem. Reward hacking has been demonstrated during iterative self-improvement of code agents, showing that ordinary optimization pressure can favor proxy-improving solutions that do not improve the intended objective [121]. Research on self-evolving agents also demonstrates that persistent memory and other self-evolution pathways can produce unintended behavioral degradation without requiring an externally inserted malicious skill [83]. Separately, coding-agent research demonstrates that execution trajectories can be distilled into persistent reusable procedural skills that guide future software-engineering tasks [50]. Security research further shows that poisoned experience can be promoted

by an agent's own skill-extraction process into a durable reusable skill, although that threat model begins with external adversarial influence [111]. Consequently, the proposed work should not be framed as the first investigation of reward hacking, unsafe self-evolution, persistent agent knowledge, or persistent skill compromise.

Instead, the proposed research addresses a narrower intersection between these areas. Our systematic review did not identify an included primary study, under the review protocol and eligibility criteria, that directly demonstrates the complete sequence in which an LLM code agent: (1) autonomously discovers a reward- or evaluation-exploiting strategy during ordinary optimization; (2) explicitly encodes that strategy into a persistent reusable natural-language skill or procedural rule; and (3) subsequently realizes or transfers that learned strategy through later code generations. This is a narrower claim than asserting that reward hacking, persistent self-evolution, or harmful skills themselves are unexplored. Existing work establishes those individual mechanisms but does not, within the reviewed corpus, join them into the complete endogenous skill-level causal chain.

This distinction motivates a broader trustworthiness question for self-improving LLM systems: *should reward-validated optimization knowledge automatically be treated as trustworthiness-validated simply because it improves the measured optimization objective?* The proposed research argues that it should not. Reward validation establishes empirical evidence that learned knowledge improves the objective measured by the optimization framework, but it does not establish that the knowledge preserves the intended task semantics or the integrity of the evaluation process. Consequently, reward-validated optimization knowledge should be treated as untrusted with respect to these stronger trustworthiness requirements unless and until it has been independently validated for admission into persistent agent knowledge.

To investigate this problem, future work will develop an *Ontology-Grounded Skill Validation Framework* centered on an ontology-backed knowledge graph. The ontology will provide an ex-

plicit semantic model of skills, tasks, objectives, evaluators, metrics, program transformations, evidence, and trustworthiness constraints. The knowledge graph will instantiate this model for individual optimization episodes and maintain the provenance of learned skill rules. Formal graph constraints and independent program-verification evidence will then determine whether a proposed skill update may be incorporated into persistent knowledge.

6.1.2 Research Questions

The proposed work will investigate four research questions:

1. **FW-RQ1 — Emergence and Persistence:** To what extent does reward-guided natural-language skill optimization autonomously produce reward- or evaluation-hacking strategies, and under what conditions are these strategies incorporated into persistent skill knowledge?
2. **FW-RQ2 — Propagation and Transfer:** Once an untrustworthy strategy is encoded into a persistent skill, to what extent does it influence subsequent code generations or transfer across workload sizes, programming tasks, or LLM generators?
3. **FW-RQ3 — Ontology-Grounded Validation:** Can an ontology-backed knowledge graph containing explicit trustworthiness semantics, constraints, provenance, and verification evidence accurately detect and prevent endogenous reward-hacking strategies from being incorporated into persistent skills?
4. **FW-RQ4 — Generalization and Utility:** Can ontology-level trustworthiness constraints identify previously unseen forms of skill-level reward hacking while preserving legitimate improvements in program correctness, performance, code quality, and resource utilization?

Together, these questions move from establishing whether endogenous reward skill hacking oc-

curs to determining its consequences and evaluating whether structured semantic knowledge can prevent untrustworthy optimization strategies from becoming persistent.

6.1.3 Ontology-Grounded Skill Validation Framework

The proposed framework will separate *learned optimization knowledge* from *authoritative trustworthiness knowledge*. Learned knowledge consists of natural-language optimization strategies generated through the skill-refinement process. These strategies remain mutable and may improve as the LLM acquires additional experience. In contrast, the trustworthiness ontology defines semantic relationships and constraints that learned skills are not permitted to override merely because a proposed modification increases the optimization reward.

The ontology will define core concepts such as:

`Skill`, `SkillUpdate`, `Task`, `Objective`, `Evaluator`, `Metric`, `Measurement`, `ProgramBehavior`, `TrustworthinessConstraint`, `Evidence`, and `Verifier`.

Relationships among these concepts may include:

`optimizes`, `affectsMetric`, `measuredBy`, `requiresEvidence`, `prohibitedBy`, `preservesSemantics`, `verifiedBy`, `derivedFrom`, `conflictsWith`.

The ontology will therefore represent not only which behaviors are permitted or prohibited but also *why* a requirement applies to a particular skill update. For example, a skill modification that changes how execution time is reported would be connected to a `PerformanceMetric`. The

ontology could infer that the skill is subject to a performance-measurement integrity constraint, which requires the reported value to originate from independently observed execution evidence.

The knowledge graph will instantiate these semantic relationships for each optimization episode. A graph representing a proposed skill update could contain relationships such as:

```
SkillRule_47
derivedFrom          Feedback_12
optimizes            RuntimeObjective
affectsMetric        ExecutionTime
generatedAt          Iteration_12
generatedBy          Model_X
associatedWith       MatrixMultiplicationTask
requiresEvidence     RuntimeMeasurement
verificationStatus   Unverified
```

Maintaining these relationships creates an explicit provenance chain connecting learned optimization knowledge to the task, feedback, reward, and evidence responsible for its creation. Provenance is particularly important because the system can then determine whether an untrustworthy skill originated from misleading evaluation feedback, an incorrectly interpreted metric, or an internally generated shortcut.

6.1.4 Semantic and Constraint-Based Skill Validation

The ontology-backed knowledge graph will function as an active validation mechanism rather than a passive repository. Each proposed natural-language skill rewrite will first be converted into a

structured semantic representation conforming to the ontology. The graph will then be enriched through ontology reasoning to determine which trustworthiness constraints apply to the proposed update.

The validation pipeline can be represented as:

$$\mathcal{S}_i \xrightarrow{\rho} \mathcal{S}'_{i+1} \xrightarrow{\text{SemanticParse}} G_{i+1} \xrightarrow{\text{Reason}} G^*_{i+1} \xrightarrow{\text{Validate}} \begin{cases} \mathcal{S}'_{i+1}, & \text{accept}, \\ \widetilde{\mathcal{S}}_{i+1}, & \text{repair}, \\ \mathcal{S}_i, & \text{reject}. \end{cases} \tag{6.1}$$

where $\mathcal{S}'_{i+1}$ represents the proposed skill update, G_{i+1} represents its initial knowledge-graph representation, and G^*_{i+1} represents the graph after semantic reasoning and inference.

The proposed implementation will investigate the Web Ontology Language (OWL 2) for defining semantic concepts and relationships and the Shapes Constraint Language (SHACL) for defining machine-evaluable graph constraints [94, 95]. This separation is useful because ontology reasoning establishes semantic relationships, while graph constraints determine whether a proposed skill satisfies requirements necessary for admission into persistent knowledge.

For example, rather than encoding a narrow rule such as:

> "Do not instruct a generated program to print a runtime of 0.00 seconds."

the ontology can represent the more general relationship:

$$\texttt{PerformanceClaim} \xrightarrow{\texttt{mustBeDerivedFrom}} \texttt{TrustedMeasurement}. \tag{6.2}$$

A constraint can then require every reported performance claim affected by a learned skill to possess independently verifiable measurement evidence. Such a representation is preferable because it does not merely recognize a previously observed exploit. It potentially detects novel strategies that violate the same underlying trustworthiness principle.

Similarly, semantic-preservation requirements could specify that:

$$\texttt{OptimizationTransformation} \xrightarrow{\texttt{mustPreserve}} \texttt{RequiredTaskSemantics}. \quad (6.3)$$

This constraint could identify strategies that obtain faster execution by reducing the required workload, lowering required precision, omitting computation, or altering the requested output even if the exact transformation had not previously been observed.

6.1.5 Proposed Research Methodology

The research will proceed through six consolidated phases.

6.1.5.1 Phase 1: Characterize Endogenous Reward Skill Hacking

The first phase will establish a formal taxonomy of endogenous reward skill hacking and distinguish it from conventional reward hacking, evaluation hacking, implementation defects, and externally induced skill compromise. Existing skill-rewrite histories will be examined to identify naturally occurring untrustworthy strategies. Candidate categories may include fabricated measurements, hard-coded outputs, reduced workloads, weakened numerical requirements, bypassed

correctness checks, suppression of static-analysis findings, evaluator manipulation, or other transformations that improve the apparent objective without a corresponding improvement in the intended program behavior.

Each event will be classified at multiple stages: whether an exploit was *proposed*, whether it was *committed* to the skill, whether it was *realized* in generated code, and whether it actually *changed the optimization reward*. This distinction is necessary because a skill may contain an attempted exploit even when an independently controlled evaluator prevents that exploit from succeeding.

6.1.5.2 Phase 2: Develop the Trustworthiness Ontology and Knowledge Graph

The second phase will construct the semantic foundation for skill validation. The ontology will model skills, optimization objectives, evaluators, measurements, program behaviors, trustworthiness properties, evidence, and verification mechanisms. Where applicable, concepts will be aligned with existing software-security, provenance, and semantic-web standards rather than creating domain terminology unnecessarily.

The corresponding knowledge graph will store both authoritative trustworthiness knowledge and episode-specific evidence. Each learned skill rule will retain provenance describing the task, iteration, generating model, optimization feedback, reward change, and verification evidence responsible for its creation. This will enable queries over the history and trustworthiness of learned optimization strategies.

6.1.5.3 Phase 3: Develop Semantic Skill Extraction and Formal Constraints

The third phase will translate proposed natural-language skill updates into structured ontology instances. Because semantic extraction is itself an LLM-supported process and may introduce

errors, extraction accuracy will be evaluated independently of validation accuracy. This separation will distinguish failures caused by incorrect semantic interpretation from failures in the ontology or constraint system.

Trustworthiness requirements will then be expressed as machine-evaluable graph constraints. Initial constraint families will include:

- evaluation and measurement integrity;
- functional and semantic preservation;
- workload preservation;
- correctness-test integrity;
- static- and dynamic-analysis integrity;
- performance-measurement provenance;
- security requirements; and
- evidence and provenance requirements.

The objective will be to encode underlying trustworthiness principles rather than exploit-specific signatures.

6.1.5.4 Phase 4: Integrate Independent Executable Verification

Some trustworthiness properties cannot be established solely through semantic reasoning. The knowledge graph will therefore coordinate external verification mechanisms. For example, a constraint requiring preservation of program semantics may require successful execution of hidden

correctness tests. A performance-integrity constraint may require independently measured runtime. Security and software-quality requirements may invoke static or dynamic analysis.

The resulting verification evidence will be inserted into the knowledge graph and linked to the skill update under evaluation. A proposed skill can therefore be admitted only when both its semantic relationships and the evidence supporting its claims satisfy the applicable trustworthiness constraints.

This phase creates two distinct trust boundaries: the proposed skill update is evaluated against the applicable semantic, provenance, and evidence requirements before becoming persistent, and programs generated from an accepted skill are subsequently verified against independently measurable properties to assess whether the skill preserves the intended behavior under the evaluated conditions.

6.1.5.5 Phase 5: Evaluate Persistence, Transfer, and Generalization

The fifth phase will experimentally evaluate both the threat and the validation system. Exploit-bearing skills that emerge during optimization will be tested on subsequent generations, different workload sizes, held-out programming tasks, and, where practical, alternative LLM generators. These experiments will determine whether an exploit remains task-specific or becomes transferable procedural knowledge.

The ontology-backed validator will then be tested on both previously observed and previously unseen reward-hacking strategies. Of particular interest is whether semantic constraints can identify a novel exploit because it violates a general property such as measurement provenance or semantic preservation rather than because the specific attack pattern was included during ontology construction.

6.1.5.6 Phase 6: Evaluate Trustworthiness and Optimization Utility

The final phase will compare ontology-grounded validation with unconstrained persistent skill optimization. The evaluation will measure both trustworthiness and legitimate optimization utility.

Proposed trustworthiness metrics include:

- *Skill Exploit Emergence Rate*: frequency with which optimization proposes exploit-bearing skills;
- *Skill Exploit Commit Rate*: proportion of exploit-bearing proposals admitted into persistent knowledge;
- *Behavioral Realization Rate*: proportion of committed exploit-bearing skills that affect subsequent generated programs;
- *Exploit Transfer Rate*: proportion of exploit-bearing skills that influence held-out tasks, workloads, or models;
- validation precision and recall;
- false rejection rate for legitimate skill improvements; and
- detection rate for previously unseen exploit classes.

These measures will be evaluated alongside existing optimization measures such as functional correctness, execution time, static-analysis findings, resource utilization, token consumption, and validation overhead. This joint analysis will determine whether the ontology-backed framework increases trustworthiness without preventing legitimate skill improvement.

The six phases collectively provide direct empirical coverage of the four future-work research questions. Phase 1 addresses FW-RQ1 by characterizing the emergence, commitment, and persistence of exploit-bearing skill updates. Phases 5 and 6 address FW-RQ2 by measuring whether persistent untrustworthy strategies influence subsequent generations and transfer across workloads, tasks, or models. Phases 2–4, together with the validation experiments in Phase 6, address FW-RQ3 by constructing and evaluating the ontology-backed knowledge graph, formal constraints, provenance model, and independent verification mechanisms used to prevent untrustworthy skill updates from becoming persistent. Finally, Phases 5 and 6 address FW-RQ4 by evaluating whether ontology-level constraints generalize to previously unseen exploit strategies while preserving legitimate improvements in correctness, performance, code quality, and resource utilization.

6.1.6 Scope and Anticipated Limitations

The proposed research is intentionally scoped to LLM-based code-generation systems in which optimization knowledge is represented as a persistent, inspectable natural-language skill or procedural rule. The study therefore does not attempt to characterize all forms of reward hacking, model-weight self-improvement, latent-policy modification, or persistent agent memory. Instead, it focuses on the narrower transition from reward-guided optimization to explicit reusable skill knowledge and on whether that knowledge can be validated before it becomes persistent.

The ontology-backed validation framework will also be limited by the completeness of its semantic representation and encoded trustworthiness constraints. Satisfying an ontology or graph constraint does not constitute a proof that a skill is universally trustworthy; it establishes that the represented skill update satisfies the explicit semantic, provenance, and evidence requirements encoded by the framework. Previously unknown behaviors that fall outside those requirements may therefore remain undetected.

A second limitation concerns semantic extraction. Proposed natural-language skill updates must be translated into structured ontology instances before formal constraints can be evaluated. Because this translation may itself be performed or assisted by an LLM, extraction errors could cause either missed violations or false rejections. The proposed experiments will therefore measure semantic-extraction accuracy separately from constraint-validation accuracy.

Independent executable verification provides an additional safeguard but is also necessarily incomplete. Hidden tests, external runtime measurements, static and dynamic analysis, and other verification mechanisms establish evidence only for the properties they measure. The study will therefore avoid treating a successful verification result as a general proof of program correctness or skill trustworthiness.

Finally, results may depend on the selected tasks, reward functions, models, and evaluation environments. Cross-task, cross-workload, and, where practical, cross-model experiments will test generalization, but conclusions will remain bounded by the evaluated conditions. Likewise, the literature-derived novelty claim is bounded by the systematic-review protocol: the appropriate claim is that no included verified publication identified by that protocol demonstrates the complete endogenous exploit-discovery, explicit skill-encoding, and subsequent reuse or transfer sequence.

6.1.7 Expected Contributions

The proposed research is expected to make four primary contributions.

First, it will provide an empirical characterization of *endogenous reward skill hacking* as a failure mode of persistent natural-language skill optimization, distinguishing attempted exploit formation, successful evaluator exploitation, persistence, behavioral realization, and transfer.

Second, it will introduce a formal ontology for representing the relationship between learned skills,

optimization objectives, evaluators, program behaviors, trustworthiness constraints, provenance, and verification evidence. The resulting knowledge graph will provide an explicit and queryable representation of both the knowledge an agent learns and the evidence supporting that knowledge.

Third, it will develop an ontology-grounded validation mechanism in which learned skill updates are semantically represented, reasoned over, validated against formal graph constraints, and supported by independently generated execution evidence before becoming persistent. This design separates the LLM responsible for proposing optimization knowledge from the mechanisms responsible for determining whether that knowledge satisfies explicit trustworthiness constraints and independently defined evidence requirements.

Finally, the research will evaluate whether ontology-level trustworthiness constraints generalize beyond previously observed reward-hacking strategies while preserving legitimate optimization utility. Rather than detecting only known exploit signatures, the proposed framework will test whether general principles such as measurement provenance, evaluator independence, workload preservation, and semantic preservation identify previously unseen evaluation-exploiting strategies without unnecessarily rejecting valid skill updates that improve program correctness, performance, code quality, or resource utilization.

Taken together, this work extends the dissertation from improving the effectiveness of persistent natural-language skills to addressing the trustworthiness of the knowledge those skills contain. The broader principle motivating this research is that *optimization knowledge learned by a self-improving LLM should be treated as an untrusted artifact: it should be explicitly represented, traced to its provenance, evaluated against independently defined semantic constraints, and supported by verifiable evidence before it is permitted to become persistent knowledge*.

LIST OF REFERENCES

[1] AAA East Central. AAA: Potholes can be a black hole for your wallet, March 2024. URL https://news.eastcentral.aaa.com/news/releases-20240308. Accessed: 2026-03-02.

[2] AAA Western and Central New York. AAA: Massive bump, 57% increase in drivers who sustain vehicle damage from potholes, March 2023. URL https://westerncentralnyaaa.com/news-room/aaa-massive-bump-57-increase-drivers-who-sustain-vehicle-damage-potholes. Accessed: 2026-03-02.

[3] Rashid Abbasi, Ali Kashif Bashir, Amjad Rehman, and Yuan Ge. 3D LiDAR point cloud segmentation for automated driving. *IEEE Intelligent Transportation Systems Magazine*, 17 (1):8–29, 2025. doi: 10.1109/MITS.2023.3325854.

[4] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22 (6):685–701, 2010.

[5] Szilárd Aradi. Survey of deep reinforcement learning for motion planning of autonomous vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 23(2):740–759, 2022. doi: 10.1109/TITS.2020.3024655.

[6] Audi UK. As seen by the new Audi A8: Predictive active suspension, 2017. URL https://www.audi.co.uk/dam/nemo/uk/Company/Innovation/Automotive_Manufacturing/AUDI_A8_Dossier_A4.pdf. Accessed: 2026-01-15.

[7] Vahid Behzadan and Arslan Munir. Adversarial reinforcement learning framework for benchmarking collision avoidance mechanisms in autonomous vehicles. *IEEE Intelligent*

Transportation Systems Magazine, 13(2):236–241, 2021. doi: 10.1109/MITS.2019.2898964.

[8] BYD Global. BYD launches DiSus intelligent body control system, 2023. URL https://www.bydglobal.com/en/BYD_Launches_DiSus_Intelligent_Body_Control_System.html. Accessed: 2026-01-15.

[9] California Department of Motor Vehicles. California DMV approves Mercedes-Benz automated driving system for certain highways and conditions, June 2023. URL https://www.dmv.ca.gov/portal/news-and-media/california-dmv-approves-mercedes-benz-automated-driving-system-for-certain-highways-and-conditions/. Accessed: 2026-02-03.

[10] Jialun Cao, Meiziniu Li, Ming Wen, and Shing chi Cheung. A study on prompt design, advantages and limitations of chatgpt for deep learning program repair, 2023. URL https://arxiv.org/abs/2304.08191.

[11] Omar Chaaban, Leen Daher, Yara Ghaddar, Nader Zantout, Daniel Asmar, and Naseem Daher. An end-to-end pothole detection and avoidance system for autonomous ground vehicles. In *2025 International Conference on Control, Automation, and Instrumentation (IC2AI)*, pages 1–6. IEEE, 2025.

[12] K. Chang and K. Chon. A car-following model applied reaction times distribution and perceptual threshold. *Journal of the Eastern Asia Society for Transportation Studies*, 6: 1888–1903, 2005. URL https://easts.info/on-line/journal_06/1888.pdf. Accessed: 2026-03-02.

[13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[14] Zhenbang Chen, Hengbiao Yu, Xianjin Fu, and Ji Wang. Mpi-sv: A symbolic verifier for mpi programs. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 93–96, 2020.

[15] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 47(9):1943–1959, 2021. doi: 10.1109/TSE.2019.2940179.

[16] Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready ai agents with scalable long-term memory. *arXiv preprint arXiv:2504.19413*, 2025.

[17] Gene Cooperman, Dahong Li, and Zhengji Zhao. Debugging mpi implementations via reduction-to-primitives. In *2022 IEEE/ACM Third International Symposium on Checkpointing for Supercomputing (SuperCheck)*, pages 1–9, 2022. doi: 10.1109/SuperCheck56652.2022.00007.

[18] Daimler Communications. Press information: Road surface scan and MAGIC BODY CONTROL (S-class), 2017. URL `https://www.motorshow.me/uploadImages/GalleryDocs/Doc6668.pdf`. Accessed: 2026-01-15.

[19] Alexander Droste, Michael Kuhn, and Thomas Ludwig. Mpi-checker: static analysis for mpi. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450340052. doi: 10.1145/2833157.2833159. URL `https://doi.org/10.1145/2833157.2833159`.

[20] Yuchuan Du, Jing Chen, Cong Zhao, Chenglong Liu, Feixiong Liao, and Ching-Yao Chan. Comfortable and energy-efficient speed control of autonomous vehicles on rough pavements

using deep reinforcement learning. *Transportation Research Part C: Emerging Technologies*, 134:103489, 2022. doi: 10.1016/j.trc.2021.103489.

[21] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53, 2023. doi: 10.1109/ICSE-FoSE59343.2023.00008.

[22] Zheng Fang, Dongming Jin, Yihong Dong, Yongmin Li, Kechi Zhang, Zhi Jin, and Ge Li. Clarifycodebench: Evaluating llms on clarifying ambiguous requirements for code generation. *arXiv preprint arXiv:2607.00711*, 2026.

[23] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020. URL https://arxiv.org/abs/2002.08155.

[24] Yang Gao, Gang Quan, Wujie Wen, Scott Piersall, Qian Lou, and Liqiang Wang. Efficient privacy-preserving sparse matrix-vector multiplication using homomorphic encryption. *Information Sciences*, 739:123180, 2026. doi: 10.1016/j.ins.2026.123180. URL https://doi.org/10.1016/j.ins.2026.123180.

[25] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 2 edition, 1999. ISBN 9780262571326.

[26] Qi Guo, Junming Cao, Xiaofei Xie, Shangqing Liu, Xiaohong Li, Bihuan Chen, and Xin Peng. Exploring the potential of chatgpt in automated code refinement: An empirical study.

In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pages 390–402, 2024. doi: 10.1145/3597503.3623306.

[27] Yushan Han, Hui Zhang, Huifang Li, Yi Jin, Congyan Lang, and Yidong Li. Collaborative perception in autonomous driving: Methods, datasets, and challenges. *IEEE Intelligent Transportation Systems Magazine*, 15(6):131–151, 2023. doi: 10.1109/MITS.2023.3298534.

[28] Tobias Hilbrich, Matthias S. Müller, Bronis R. de Supinski, Martin Schulz, and Wolfgang E. Nagel. Gti: A generic tools infrastructure for event-based tools in parallel systems. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 1364–1375, 2012. doi: 10.1109/IPDPS.2012.123.

[29] Tobias Hilbrich, Joachim Protze, Martin Schulz, Bronis R. de Supinski, and Matthias S. Muller. Mpi runtime error detection with must: Advances in deadlock detection. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–10, 2012. doi: 10.1109/SC.2012.79.

[30] Honda. Honda receives type designation for Level 3 automated driving equipment, November 2020. URL `https://global.honda/en/newsroom/news/2020/4201111eng.html`. Accessed: 2026-02-03.

[31] Jun Hu, Yuefeng Wang, Shuai Cheng, Jinghan Xu, Ningjia Wang, Bingjie Fu, Zuotao Ning, Jingyao Li, Hualin Chen, Chaolu Feng, et al. A survey of decision-making and planning methods for self-driving vehicles. *Frontiers in Neurorobotics*, 19:1451923, 2025.

[32] Nenghui Hu, Zheng Bian, Ziqi Shuai, Zhenbang Chen, and Yufeng Zhang. Symbolic execution of mpi programs with one-sided communications. In *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*, pages 657–658, 2023. doi: 10.1109/APSEC60848.2023.00096.

[33] Chao Huang, Chen Lv, Peng Hang, Zhongxu Hu, and Yang Xing. Human-machine adaptive shared control for safe driving under automation degradation. *IEEE Intelligent Transportation Systems Magazine*, 14(2):53–66, 2022. doi: 10.1109/MITS.2021.3065382.

[34] Tenglong Huang, Jue Wang, and Huihui Pan. Adaptive bioinspired preview suspension control with constrained velocity planning for autonomous vehicles. *IEEE Transactions on Intelligent Vehicles*, 8(7):3925–3937, 2023. doi: 10.1109/TIV.2023.3272067.

[35] Yu Huang, Benjamin Ogles, and Eric Mercer. A predictive analysis for detecting deadlock in mpi programs. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 18–28, 2020.

[36] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report, 2024. URL `https://arxiv.org/abs/2409.12186`.

[37] Zahid Hussain, Shabna Sayed Mohammed, Charitha Dias, Qinaat Hussain, and Wael K. M. Alhajyaseen. Empirical analysis of car-following behavior: Impacts of driver demographics, leading vehicle types, and speed limits on driver behavior and safety. *Transportation Research Part F: Traffic Psychology and Behaviour*, 108:188–205, 2025. doi: 10.1016/j.trf.2024.11.022. URL `https://www.sciencedirect.com/science/article/pii/S1369847824003243`.

[38] Rohan Inamdar, S. Kavin Sundarr, Deepen Khandelwal, Varun Dev Sahu, and Nitish Katal. A comprehensive review on safe reinforcement learning for autonomous vehicle control in dynamic environments. *e-Prime - Advances in Electrical Engineering, Electronics and Energy*, 10:100810, 2024. doi: 10.1016/j.prime.2024.100810.

[39] Jaguar Land Rover. Jaguar Land Rover develops technology to detect potholes, 2015. URL https://media.jaguarlandrover.com/news/2015/06/jaguar-land-rover-develops-technology-detect-potholes. Accessed: 2026-01-15.

[40] Erel Kaplan, Tomer Bitan, Lian Ghrayeb, Le Chen, Tom Yotam, Niranjan Hasabnis, and Gal Oren. ParaCodex: A profiling-guided autonomous coding agent for reliable parallel code generation and translation. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens, editors, *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 16113–16136, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.732. URL https://aclanthology.org/2026.acl-long.732/.

[41] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, et al. Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. In *Tools for High Performance Computing 2011: Proceedings of the 5th International Workshop on Parallel Tools for High Performance Computing, September 2011, ZIH, Dresden*, pages 79–91. Springer, 2012.

[42] Mark Lajko, Viktor Csuvik, Tibor Gyimothy, and Laszlo Vidacs. Automated program repair with the gpt family, including gpt-2, gpt-3 and codex. In *Proceedings of the 5th ACM/IEEE International Workshop on Automated Program Repair*, APR '24, page 34–41, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705779. doi: 10.1145/3643788.3648021. URL https://doi.org/10.1145/3643788.3648021.

[43] Mathieu Laurent, Emmanuelle Saillard, and Martin Quinson. The mpi bugs initiative: a framework for mpi verification tools evaluation. In *2021 IEEE/ACM 5th Interna-*

tional Workshop on Software Correctness for HPC Applications, pages 1–9, 2021. doi: 10.1109/Correctness54621.2021.00008.

[44] Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses, 2026. URL https://arxiv.org/abs/2603.28052.

[45] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474, 2020.

[46] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. Assisting static analysis with large language models: A chatgpt experiment. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 2107–2111, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400703270. doi: 10.1145/3611643.3613078. URL https://doi.org/10.1145/3611643.3613078.

[47] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. Enhancing static analysis for practical bug detection: An llm-integrated approach. *Proc. ACM Program. Lang.*, 8(OOPSLA1), April 2024. doi: 10.1145/3649828. URL https://doi.org/10.1145/3649828.

[48] Hongbo Li, Sihuan Li, Zachary Benavides, Zizhong Chen, and Rajiv Gupta. Compi: Concolic testing for mpi applications. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 865–874, 2018. doi: 10.1109/IPDPS.2018.00096.

[49] Hongbo Li, Zizhong Chen, and Rajiv Gupta. Efficient concolic testing of mpi applications. In *Proceedings of the 28th International Conference on Compiler Construction*, CC 2019,

page 193–204, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362771. doi: 10.1145/3302516.3307353. URL https://doi.org/10.1145/3302516.3307353.

[50] Yanzhou Li, Yiran Zhang, Xiaoyu Zhang, Xiaoxia Liu, and Yang Liu. CODESKILL: Learning self-evolving skills for coding agents, 2026. URL https://arxiv.org/abs/2605.25430.

[51] Siyu Lin, Qingwei Mi, and Tianhan Gao. A survey of curriculum learning in deep reinforcement learning. In *2025 IEEE 15th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 1141–1147, Las Vegas, NV, USA, January 2025. IEEE. doi: 10.1109/CCWC62904.2025.10903795.

[52] Zhiyuan Liu, Yulin Si, and Weichao Sun. Ride comfort oriented integrated design of preview active suspension control and longitudinal velocity planning. *Mechanical Systems and Signal Processing*, 208:110992, 2024. doi: 10.1016/j.ymssp.2023.110992.

[53] Ziqing Luo, Manchun Zheng, and Stephen F. Siegel. Verification of mpi programs using civl. In *Proceedings of the 24th European MPI Users' Group Meeting*, EuroMPI '17, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450348492. doi: 10.1145/3127024.3127032. URL https://doi.org/10.1145/3127024.3127032.

[54] Hongyi Ma, Liqiang Wang, and Krishanthan Krishnamoorthy. Detecting thread-safety violations in hybrid openmp/mpi programs. In *2015 IEEE International Conference on Cluster Computing*, pages 460–463. IEEE, 2015.

[55] Ehsan Mashhadi and Hadi Hemmati. Applying codebert for automated program repair of java simple bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 505–509, 2021. doi: 10.1109/MSR52588.2021.00063.

[56] Mercedes-Benz Group. DRIVE PILOT: Support speed of up to 95 km/h on german motorways, December 2024. URL https://group.mercedes-benz.com/innovations/product-innovation/autonomous-driving/drive-pilot-95-kmh.html. Accessed: 2026-02-03.

[57] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 5.0*, June 2025. URL https://www.mpi-forum.org/docs/mpi-5.0/mpi50-report.pdf.

[58] Sanmit Narvekar, Bei Peng, Matteo Leonetti, Jivko Sinapov, Matthew E. Taylor, and Peter Stone. Curriculum learning for reinforcement learning domains: A framework and survey. *Journal of Machine Learning Research*, 21(181):1–50, 2020. URL https://www.jmlr.org/papers/volume21/20-131/20-131.pdf. Accessed: 2026-03-02.

[59] National Highway Traffic Safety Administration. The 100-car naturalistic driving study. Technical Report DOT HS 808 536, U.S. Department of Transportation, 2002. URL https://www.nhtsa.gov/sites/nhtsa.gov/files/100carphase1report.pdf. Accessed: 2026-01-01.

[60] National Highway Traffic Safety Administration. The impact of driver inattention on near-crash/crash risk. Technical Report DOT HS 810 894, U.S. Department of Transportation, 2006. URL https://www.nhtsa.gov/sites/nhtsa.gov/files/driverinattention.pdf. Accessed: 2026-01-01.

[61] National Highway Traffic Safety Administration. Comparing real-world behaviors of drivers with high versus low rates of crashes and near-crashes. Technical Report DOT HS 811 091, U.S. Department of Transportation, 2009. URL https://www.nhtsa.gov/sites/nhtsa.gov/files/811091.pdf. Accessed: 2026-01-01.

[62] Van Man Nguyen, Emmanuelle Saillard, Julien Jaeger, Denis Barthou, and Patrick Carribault. Parcoach extension for static mpi nonblocking and persistent communication validation. In *2020 IEEE/ACM 4th International Workshop on Software Correctness for HPC Applications*, pages 31–39, 2020. doi: 10.1109/Correctness51934.2020.00009.

[63] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. Learning deep semantics for test completion. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 2111–2123. IEEE Press, 2023. ISBN 9781665457019. doi: 10.1109/ICSE48619.2023.00178. URL https://doi.org/10.1109/ICSE48619.2023.00178.

[64] NIO. 4D smart suspension: Road-preview and proactive body control, 2024. URL https://www.nio.com/nio-technologies/4d-smart-suspension. Accessed: 2026-01-15.

[65] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131.

[66] OpenAI. Introducing codex. https://openai.com/index/introducing-codex/, May 2025. Accessed: 2026-06-11.

[67] OpenAI. GPT-5.1-Codex-Max model. OpenAI API Documentation, 2025. URL https://developers.openai.com/api/docs/models/gpt-5.1-codex-max. Accessed: 2026-04-23.

[68] OpenAI. Building more with GPT-5.1-Codex-Max. OpenAI Product Announcement, November 2025. URL https://openai.com/index/gpt-5-1-codex-max/. Accessed: 2026-04-23.

[69] OpenAI. Gpt-5.3-codex. https://developers.openai.com/api/docs/models/gpt-5.3-codex, 2026. Accessed: 2026-07-30.

[70] Peter Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann, 1 edition, 2011. ISBN 9780123742605.

[71] Charles Packer, Vivian Fang, Shishir_G Patil, Kevin Lin, Sarah Wooders, and Joseph_E Gonzalez. Memgpt: towards llms as operating systems. 2023.

[72] Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pages 1–22, 2023.

[73] Scott Piersall, Yang Gao, Shenyang Liu, and Liqiang Wang. Improving MPI error detection and repair with large language models and bug references. *Journal of Parallel and Distributed Computing*, 213:105255, 2026. doi: 10.1016/j.jpdc.2026.105255. URL https://doi.org/10.1016/j.jpdc.2026.105255.

[74] PR Newswire. Baidu granted China's first-ever permits for commercial fully driverless ride-hailing services, August 2022. URL https://www.prnewswire.com/news-releases/baidu-granted-chinas-first-ever-permits-for-commercial-fully-driverless-ride-hailing-services-301601296.html. Accessed: 2026-03-02.

[75] PR Newswire. Baidu wins China capital city's first-ever permit to provide fully driverless

ride-hailing service, March 2023. URL https://www.prnewswire.com/news-releases/baidu-wins-china-capital-citys-first-ever-permit-to-provide-fully-driverless-ride-hailing-service-301774850.html. Accessed: 2026-02-03.

[76] Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and Xiaoguang Mao. SoapFL: A standard operating procedure for LLM-based method-level fault localization. *IEEE Transactions on Software Engineering*, 51(4):1173–1187, 2025. doi: 10.1109/TSE.2025.3543187.

[77] Alexandru Ionut Radu, Adrian Bogdan Tolea, Daniel Dragos Trusca, Nicolae Ispas, and Mircea Nastasoiu. Development of a multibody model used to study the impact between a vehicle wheel with a pothole. *International Journal of Automotive Technology*, 24(5): 1441–1446, 2023. doi: 10.1007/s12239-023-0116-4.

[78] Nikitha Rao, Kush Jain, Uri Alon, Claire Le Goues, and Vincent J. Hellendoorn. Cat-lm training language models on aligned code and tests. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, ASE '23, page 409–420. IEEE Press, 2024. ISBN 9798350329964. doi: 10.1109/ASE56229.2023.00193. URL https://doi.org/10.1109/ASE56229.2023.00193.

[79] Reuters. China approves first batch of L3 autonomous driving vehicles, December 2025. URL https://www.reuters.com/world/asia-pacific/china-approves-first-batch-l3-autonomous-driving-vehicles-2025-12-15/. Accessed: 2026-03-02.

[80] Reuters. Waymo valued at $126 billion in latest financing as robotaxis gather steam, February 2026. URL https://www.reuters.com/business/waymo-valued-126-billion-latest-financing-2026-02-02/. Accessed: 2026-03-02.

[81] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024. URL https://arxiv.org/abs/2308.12950.

[82] Rana Salama, Jason Cai, Michelle Yuan, Anna Currey, Monica Sunkara, Yi Zhang, and Yassine Benajiba. MemInsight: Autonomous memory augmentation for LLM agents. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 33136–33152, Suzhou, China, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-main.1683. URL https://aclanthology.org/2025.emnlp-main.1683/.

[83] Shuai Shao, Qihan Ren, Chen Qian, Boyi Wei, Dadi Guo, Jingyi Yang, Xinhao Song, Linfeng Zhang, Weinan Zhang, Dongrui Liu, and Jing Shao. Your agent may misevolve: Emergent risks in self-evolving LLM agents. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=Fd1jgQQW28.

[84] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

[85] Stephen F. Siegel and Timothy K. Zirkel. Automatic formal verification of mpi-based parallel programs. *SIGPLAN Not.*, 46(8):309–310, February 2011. ISSN 0362-1340. doi: 10.1145/2038037.1941603. URL https://doi.org/10.1145/2038037.1941603.

[86] J. Sun et al. An optimal driving strategy for the ride comfort performance of intelligent vehicles on uneven roads. *IFAC-PapersOnLine*, 51(31):679–684, 2018. doi: 10.1016/j.ifac

ol.2018.10.103. URL https://www.sciencedirect.com/science/article/pii/S240589631832514X.

[87] Mehdi Testouri, Gamal Elghazaly, and Raphael Frank. Robocar: A rapidly deployable open-source platform for autonomous driving research. *IEEE Intelligent Transportation Systems Magazine*, 17(4):83–95, 2025. doi: 10.1109/MITS.2025.3546755.

[88] Johan Theunissen, Antonio Tota, Patrick Gruber, Miguel Dhaens, and Aldo Sorniotti. Preview-based techniques for vehicle suspension control: A state-of-the-art review. *Annual Reviews in Control*, 51:206–235, 2021. doi: 10.1016/j.arcontrol.2021.03.010. URL https://www.sciencedirect.com/science/article/pii/S1367578821000183.

[89] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023. URL https://arxiv.org/abs/2307.09288.

[90] Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U. Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Hannah Tan, and Omar G. Younis. Gymnasium: A standard interface for reinforcement learning environments, 2024. URL https://arxiv.org/abs/2407.17032. Accessed: 2026-03-02.

[91] Martin Treiber, Ansgar Hennecke, and Dirk Helbing. Congested traffic states in empirical observations and microscopic simulations. *Physical Review E*, 62(2):1805–1824, 2000. doi: 10.1103/PhysRevE.62.1805.

[92] Sarvani S. Vakkalanka, Subodh Sharma, Ganesh Gopalakrishnan, and Robert M. Kirby. Isp: a tool for model checking mpi programs. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, page 285–286, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595937957. doi: 10.1145/1345206.1345258. URL https://doi.org/10.1145/1345206.1345258.

[93] Pedro Valero-Lara, Aaron Young, Thomas Naughton III, Christian Engelmann, Al Geist, Jeffrey S. Vetter, Keita Teranishi, and William F. Godoy. Chatmpi: Llm-driven mpi code generation for hpc workloads. In *Proceedings of the Supercomputing Asia and International Conference on High Performance Computing in Asia Pacific Region*, SCA/HPCAsia '26, pages 19—30, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400720673. doi: 10.1145/3773656.3773659. URL https://doi.org/10.1145/3773656.3773659.

[94] W3C OWL Working Group. OWL 2 Web Ontology Language: Document Overview (Second Edition). W3C Recommendation, December 2012. URL https://www.w3.org/TR/owl2-overview/.

[95] W3C RDF Data Shapes Working Group. Shapes constraint language (SHACL). W3C Recommendation, July 2017. URL https://www.w3.org/TR/shacl/.

[96] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, 50(4):911–936, 2024. doi: 10.1109/TSE.2024.3368208.

[97] Liwen Wang, Shuo Yang, Kang Yuan, Yanjun Huang, and Hong Chen. A combined reinforcement learning and model predictive control for car-following maneuver of autonomous vehicles. *Chinese Journal of Mechanical Engineering*, 36:80, 2023. doi: 10.1186/s10033-023-00904-7.

[98] Yuning Wang, Zeyu Han, Yining Xing, Shaobing Xu, and Jianqiang Wang. A survey on datasets for the decision making of autonomous vehicles. *IEEE Intelligent Transportation Systems Magazine*, 16(2):23–40, 2024.

[99] Waymo. Scaling our fleet through U.S. manufacturing, May 2025. URL https://waymo.com/blog/2025/05/scaling-our-fleet-through-us-manufacturing. Accessed: 2026-02-03.

[100] Anjiang Wei, Allen Nie, Thiago SFX Teixeira, Rohan Yadav, Wonchan Lee, Ke Wang, and Alex Aiken. Improving parallel program performance with llm optimizers via agent-system interfaces. *arXiv preprint arXiv:2410.15625*, 2024.

[101] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[102] Jian Wu, Hongliang Zhou, Zhiyuan Liu, and Mingqin Gu. Ride comfort optimization via speed planning and preview semi-active suspension control for autonomous vehicles on

uneven roads. *IEEE Transactions on Vehicular Technology*, 69(8):8343–8355, 2020. doi: 10.1109/TVT.2020.3000709.

[103] Jingda Wu, Chao Huang, Hailong Huang, Chen Lv, Yuntong Wang, and Fei-Yue Wang. Recent advances in reinforcement learning-based autonomous driving behavior planning: A survey. *Transportation Research Part C: Emerging Technologies*, 164:104654, 2024. doi: 10.1016/j.trc.2024.104654.

[104] Yin Wu, Xiaofei Xie, Chenyang Peng, Dijun Liu, Hao Wu, Ming Fan, Ting Liu, and Haijun Wang. Advscanner: Generating adversarial smart contracts to exploit reentrancy vulnerabilities using llm and static analysis. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 1019–1031, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400712487. doi: 10.1145/3691620.3695482. URL https://doi.org/10.1145/3691620.3695482.

[105] Chao Xiang, Chen Feng, Xiaopo Xie, Botian Shi, Hao Lu, Yisheng Lv, Mingchuan Yang, and Zhendong Niu. Multi-sensor fusion and cooperative perception for autonomous driving: A review. *IEEE Intelligent Transportation Systems Magazine*, 15(5):36–58, 2023.

[106] Zhe Xiong and Weiyu Dong. Vuld-codebert: Codebert-based vulnerability detection model for c/c++ code. In *2024 6th International Conference on Communications, Information System and Computer Engineering (CISCE)*, pages 914–919, 2024. doi: 10.1109/CISCE62493.2024.10653337.

[107] Chuyang Xu, Zhongxin Liu, Xiaoxue Ren, Gehao Zhang, Ming Liang, and David Lo. FlexFL: Flexible and effective fault localization with open-source large language models. *IEEE Transactions on Software Engineering*, 51(5):1455–1471, 2025. doi: 10.1109/TSE.2025.3553363.

[108] Di Yang, Xinou Xie, Xiuwen Yang, Ming Hu, Yihao Huang, Yueling Zhang, Weikai Miao, Ting Su, Chengcheng Wan, and Geguang Pu. Assessing the impact of requirement ambiguity on LLM-based function-level code generation, 2026. URL https://arxiv.org/abs/2604.21505.

[109] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

[110] J. Javier Yebes, David Montero, and Ignacio Arriola. Learning to automatically catch potholes in worldwide road scene images. *IEEE Intelligent Transportation Systems Magazine*, 13(3):192–205, 2021. doi: 10.1109/MITS.2019.2926370.

[111] Zonghao Ying, Xiangfan Wu, Huiyu Wu, Xing Zheng, Huangsheng Cheng, Xiaorong Shi, and Jing Guo. SkillJack: Persistent skill backdoors in self-evolving agents, 2026. URL https://arxiv.org/abs/2608.03509.

[112] Hengbiao Yu. Combining symbolic execution and model checking to verify mpi programs. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 527–529, 2018.

[113] Hengbiao Yu. Combining symbolic execution and model checking to verify mpi programs. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 527–529, 2018.

[114] Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Pan Lu, Zhi Huang, Carlos Guestrin, and James Zou. Optimizing generative AI by backpropagating language model feedback. *Nature*, 639:609–616, 2025. doi: 10.1038/s41586-025-08661-4.

[115] Mohammad Amin Zadenoori, Jacek Dąbrowski, Waad Alhoshan, Liping Zhao, and Alessio Ferrari. Large language models (llms) for requirements engineering (re): A systematic literature review, 2025. URL https://arxiv.org/abs/2509.11446.

[116] Chenyuan Zhang, Hao Liu, Jiutian Zeng, Kejing Yang, Yuhong Li, and Hui Li. Prompt-enhanced software vulnerability detection using chatgpt, 2024. URL https://arxiv.org/abs/2308.12697.

[117] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484, 2023.

[118] Jiajie Zhang, Bin Xiao, Binbin Tang, Yangye Jiang, Tingzhe Yu, and Daofei Li. Safe and comfortable motion planning for bumpy road driving. In *2023 IEEE 26th International Conference on Intelligent Transportation Systems (ITSC)*, pages 5706–5711, 2023. doi: 10.1109/ITSC57777.2023.10422186.

[119] Yanlin Zhang, Tao Zhou, Xiaoguang Yang, Lichen Wang, and Mengyan Zang. Transient dynamic behavior of tire traversing obstacles in full vehicle scenario. *Proceedings of the Institution of Mechanical Engineers, Part D: Journal of Automobile Engineering*, 234(7): 2001–2013, 2020. doi: 10.1177/0954407019871758.

[120] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1592–1604, 2024.

[121] Bingchen Zhao, Dhruv Srikanth, Yuxiang Wu, and Zhengyao Jiang. Reward hacking in self-improving code agents. ICLR 2026 Workshop on AI with Recursive Self-Improvement, 2026. URL https://openreview.net/forum?id=ikrQWGgxYg.

[122] Jiuang Zhao, Donghao Yang, Li Zhang, Xiaoli Lian, Zitian Yang, and Fang Liu. Enhancing automated program repair with solution design. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 1706–1718, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400712487. doi: 10.1145/3691620.3695537. URL https://doi.org/10.1145/3691620.3695537.

[123] S. Zhao, J. Zhang, C. He, Y. Ji, H. Huang, and X. Hou. Autonomous vehicle extreme control for emergency collision avoidance via reachability-guided reinforcement learning. *Advanced Engineering Informatics*, 62:102801, 2024. doi: 10.1016/j.aei.2024.102801.

[124] Tong Zhao, Yichen Xie, Mingyu Ding, Lei Yang, Masayoshi Tomizuka, and Yintao Wei. A road surface reconstruction dataset for autonomous driving. *Scientific Data*, 11:459, 2024. doi: 10.1038/s41597-024-03261-9. URL https://www.nature.com/articles/s41597-024-03261-9.

[125] Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. Memorybank: Enhancing large language models with long-term memory. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pages 19724–19731, 2024.

[126] Meixin Zhu, Xuesong Wang, and Xiaomeng Wang. Car-following headways in different driving situations: A naturalistic driving study. In *CICTP 2016: Green and Multimodal Transportation and Logistics*, pages 1419–1428. American Society of Civil Engineers (ASCE), 2016. doi: 10.1061/9780784479896.128. URL https://meixinzhu.github.io/pdf/cota.pdf.

[127] Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. Language agents as optimizable graphs. *arXiv preprint arXiv:2402.16823*, 2024.