The New Compiler Stack: A Survey on the Synergy of LLMs and Compilers

Shuoming Zhang[1,2], Jiacheng Zhao[1,2*], Qiuchu Yu[1,2], Chunwei Xia[3], Zheng Wang[3], Xiaobing Feng[1,2], Huimin Cui[1,2]

[1*]SKLP, Institute of Computing Technology, CAS, 6th Kexueyuan South Rd, Beijing, China.
[2]University of Chinese Academy of Sciences, Beijing, China.
[3]University of Leeds, UK.

*Corresponding author(s). E-mail(s): zhaojiacheng@ict.ac.cn;
Contributing authors: zhangshuoming21s@ict.ac.cn; yuqiuchu19@mails.ucas.ac.cn;
C.Xia@leeds.ac.uk; Z.Wang5@leeds.ac.uk; fxb@ict.ac.cn; cuihm@ict.ac.cn;

Abstract

This survey has provided a systematic overview of the emerging field of LLM-enabled compilation by addressing several key research questions. We first answered how LLMs are being integrated by proposing a comprehensive, multi-dimensional taxonomy that categorizes works based on their Design Philosophy (Selector, Translator, Generator), LLM Methodology, their operational Level of Code Abstraction, and the specific Task Type they address. In answering what advancements these approaches offer, we identified three primary benefits: the democratization of compiler development, the discovery of novel optimization strategies, and the broadening of the compiler's traditional scope. Finally, in addressing the field's challenges and opportunities, we highlighted the critical hurdles of ensuring correctness and achieving scalability, while identifying the development of hybrid systems as the most promising path forward. By providing these answers, this survey serves as a foundational roadmap for researchers and practitioners, charting the course for a new generation of LLM-powered, intelligent, adaptive and synergistic compilation tools.

Keywords: survey, LLM, compiler, code translation, code optimization

1 Introduction

For decades, the compiler has stood as a cornerstone of the computing stack, undertaking the critical and complex task of translating human-readable source code into efficient machine-executable programs. A primary challenge in this process is optimization, a domain traditionally governed by intricate, handcrafted heuristics designed by human experts to navigate a vast and complex decision space. The advent of machine learning (Krizhevsky et al. 2012; He et al. 2016) introduced a new paradigm, employing data-driven models for tasks like phase-ordering and flag selection (Wang and O'Boyle 2018). However, these traditional machine learning approaches often rely on an intensive process of feature engineering, where experts must meticulously design and extract program features to train a model, leaving the core components of compiler design largely unchanged.

The recent emergence of Large Language Models (LLMs) represents a fundamental shift in this landscape. Pre-trained on vast corpora of text and code, LLMs have demonstrated a remarkable capacity to understand, generate, and transform programming languages as raw text, largely eliminating the need for explicit feature engineering. By training on codebases orders of magnitude larger than any human could study, LLMs internalize a deep understanding of programming patterns, syntax, and semantics across numerous languages. These capabilities have been rapidly integrated into the software development workflow through LLM-driven chatbots (OpenAI 2024; Gemini Team and Google 2023; Anthropic 2024), code assistants (Chen et al. 2021b; Tabnine 2022), and semi-automated agents (Anysphere, Inc. 2023; google-gemini 2025; anthropics 2025), boosting developer efficacy at every stage. This wave of innovation has, in turn, catalyzed new research within the broader compiler domain. The scope of tasks has expanded from narrow problems like pass selection to ambitious, end-to-end objectives like code transpilation and automated program repair. Consequently, the compiler's role is being reimagined from a static tool to a dynamic, interactive partner in software development.

This rapid evolution has led to a surge of new research across a wide spectrum of compiler-related tasks. Researchers are now applying LLMs to ambitious goals, including source-to-source Code Transpilation to migrate code across different languages (Roziere et al. 2020; Jana et al. 2024; Zhang et al. 2025; Ibrahimzada et al. 2025) and architectures (Wen et al. 2022; Tehrani-Jamsaz et al. 2024; Palkowski and Gruzewski 2024; Dong et al. 2025), high-level Code Optimization (Gao et al. 2025; Peng et al. 2025; Purschke et al. 2025; Xu et al. 2025; Lin et al. 2025) to surpass traditional heuristics, low-level IR Optimization (Deng et al. 2024; Cummins et al. 2025; Cui et al. 2025) to fine-tune performance and code size, and even the notoriously difficult problem of using LLMs to act as compilers (Gao et al. 2024; Zhang et al. 2024; Zhong et al. 2025) or decompilers (Xu et al. 2023; Armengol-Estape et al. 2024; Tan et al. 2024; Hu et al. 2024; Wong et al. 2025). While promising, the sheer volume and diversity of these studies can hinder a clear understanding of the current landscape. To address this gap, this paper provides a systematic review of recent advancements in LLM-enabled compiler research. We offer a comprehensive multi-dimensional taxonomy of the state-of-the-art and a clear analysis of its core advancements and challenges, establishing a roadmap for this exciting direction in compiler research.

This paper makes the following primary contributions:

- We conduct a systematic literature review, identifying and curating a corpus of 159 primary studies that represent the state-of-the-art in LLM-enabled compilation.
- We propose a novel, multi-dimensional taxonomy to structure the field. This taxonomy classifies existing work based on the LLM's **Design Philosophy** (Selector, Translator, Generator), the **LLM Methodology**, its operational **Level of Code Abstraction**, and the specific **Task Type** it addresses.
- We provide a comprehensive analysis of the primary advancements offered by LLM-based approaches, highlighting their role in democratizing compiler development, discovering novel optimizations, and broadening the scope of compilation.
- We synthesize the common challenges facing the field—including correctness, scalability, and interpretability—and discuss promising future research directions, such as the development of hybrid systems and self-improving compilers.

The remainder of this paper is organized as follows. § 2 details the research questions we target to answer and our systematic methodology for literature selection. § 3 and § 4 present our multi-dimensional taxonomy in detail, categorizing the existing work. § 5 systematically presents representative datasets and benchmarks in this field, and associated state-of-the-art advancements. § 6 provides an in-depth discussion of the field's advancements, challenges, and future opportunities. Finally, § 7 concludes the paper.

2 Methodology

This section details the systematic methodology we employed to survey the landscape of LLM-enabled compiler researches. A rigorous and transparent protocol is essential for ensuring that our review is both comprehensive and reproducible. To that end, we first present the research questions that steered our investigation in §§ 2.1. Following that, we describe the structured, three-phase literature search and selection protocol used to assemble the final corpus of primary studies for our analysis in §§ 2.2.

In this section, we introduce the detailed steps of conducting a literature review. To ensure a comprehensive and unbiased review of the field, we adopted a systematic literature review (SLR) protocol following standard practice (Kitchenham et al. 2007). This protocol defines the research questions that guide our survey, the search process for identifying relevant studies, and the criteria for including or excluding papers.

2.1 Research Questions (RQs)

To guide our literature review and provide a clear structure for our analysis, we formulated the following four key research questions regarding the application of LLMs in the compiler domain:

- **RQ1: How are LLMs being integrated into the compilation process?** We explore this question from two complementary perspectives: first, the **Design Philosophy**, which defines the LLM's architectural role within the system (task side); and second, the **LLM Methodology**, which details how the model is technically developed and applied to that task (LLM side).
- **RQ2: What are the primary compiler-related tasks addressed by LLMs?** This question aims to identify and categorize the specific compiler-related tasks, such as code optimization, transpilation, and code generation, that are being targeted by recent research.
- **RQ3: What are the primary advancements offered by LLM-based approaches?** This question focuses on summarizing the novel contributions and breakthroughs that these techniques bring to both the compiler and machine learning communities.
- **RQ4: What are the common challenges and future opportunities in this emerging field?** This question seeks to synthesize the key obstacles reported in the literature and identify promising directions for future work.

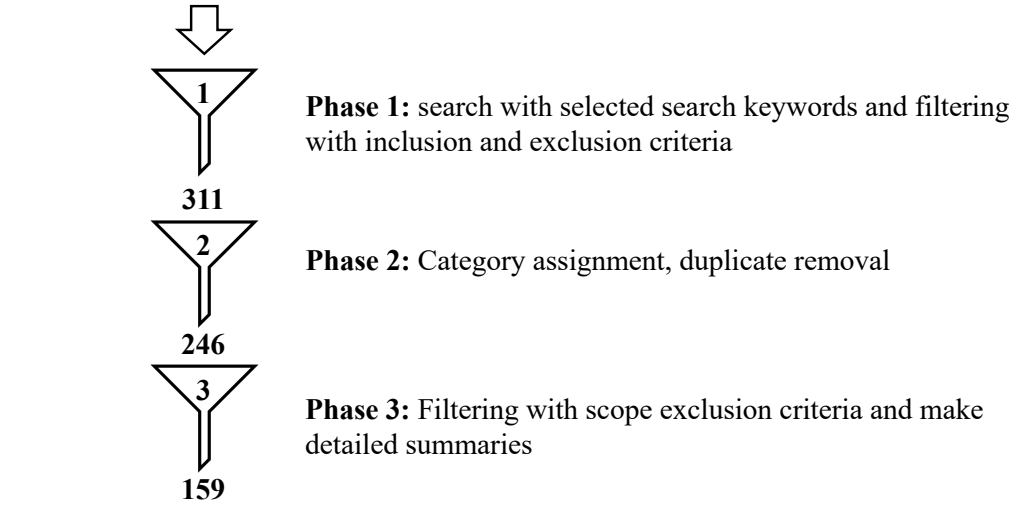

Fig. 1: Literature Search and Selection Protocol Overview

2.2 Literature Search and Selection Protocol

To systematically answer the research questions defined above, we designed and executed a comprehensive literature search and selection protocol. This process was organized into three distinct phases, which progressively filtered a large pool of initial candidates down to a focused and relevant corpus of 159 papers.

1. **Phase 1: Initial Search and Candidate Collection.** The first phase involved an extensive search for candidate papers across four major digital libraries: arXiv, Google Scholar, the ACM Digital Library, and IEEE Xplore. We utilized a broad set of search queries, including keywords such as "LLM compiler", "large language model for code optimization", "AI in compilers", "transformer for compilation", and "LLM for code generation". To complement this automated search, we also manually included several seminal papers identified by domain experts. We then applied a snowballing technique, examining the forward citation chain of these core papers to ensure comprehensive coverage and include other closely related studies who cited them.
2. **Phase 2: Deduplication and Initial Screening.** In the second phase, the initial pool of papers was processed to remove duplicates. Subsequently, we performed a preliminary screening of the remaining articles. This step involved reviewing titles and abstracts to filter out studies that were clearly outside the scope of our survey, such as those not related to compilers or not utilizing LLMs.
3. **Phase 3: Full-Text Review and Final Selection.** The final phase consisted of a full-text review of each paper that passed the initial screening. This in-depth analysis allowed us to determine a study's final eligibility based on a strict set of inclusion and exclusion criteria.

   Inclusion Criteria:
   - The study must primarily focus on using a Large Language Model (specifically a Transformer-based model) for a task related to compilation or code optimization.
   - The study must be a peer-reviewed conference paper, journal article, or a significant, relavant pre-print manuscript (to include the latest advancements often published on arXiv).
   - The full text of the study must be publicly available through digital libraries or websites for open access.

   Exclusion Criteria:
   - Studies that use traditional machine learning (e.g., SVM, Decision Trees) without language models (LMs).
   - Articles that are not technical research papers, such as editorials, keynotes, tutorials, posters or extended abstracts with insufficient detail.
   - Studies where the core application of the model was not related to the interest of broader compilation domain.

Through this rigorous three-phase process, we curated a final collection of 159 primary studies. This curated corpus forms the foundation for the systematic review, categorization, and analysis presented in the subsequent sections of this paper.

2.3 Scope of the Taxonomy

Before presenting our detailed taxonomy, it is essential to define the scope of this survey. The term "LLM-enabled compiler" encompasses a wide range of emerging research. For the purpose of this review, we define our scope to include any study that utilizes a large, pre-trained, Transformer-based language model to perform or augment a task traditionally associated with the

compilation workflow or the broader code development and optimization lifecycle.

This definition deliberately draws a distinction between the new paradigm of LLM-based approaches and prior work. Specifically, **our survey excludes**:

- **Traditional ML Techniques**: Studies that rely on classic machine learning models (e.g., Support Vector Machines, Decision Trees, Random Forests) that require extensive, handcrafted feature engineering from source code (such as code quality metrics) and have been well-studied and surveyed.
- **Purely NLP-based SE Tasks**: Studies that apply language models to software engineering artifacts without directly analyzing, transforming, or generating code are not included. For example, the classification of bug reports or the summarization of developer comments, while related, fall outside our compiler-centric focus.

By establishing this scope, we aim to provide a focused and coherent review of the state-of-the-art in the specific, transformative paradigm of applying large language models to the science and engineering of compilation. The core function of any compiler is fundamentally rooted in two processes: `translation` and `optimization`. Consequently, these two areas form the primary focus of our survey. We include a significant body of work on **Code Optimization**, where LLMs are used to rewrite programs to improve performance or reduce code size, operating at either high-level source code and low-level Intermediate Representation (IR).

Equally important is **Code Transpilation**, which we define broadly as the translation between different program representations. This includes narrowly-defined compilation (e.g., programming language to assembly), decompilation, source-to-source transpilation, and binary translation. This category also covers unique tasks like translating programming languages into hardware description languages (HDLs) or even into neural network weights.

It is important to note that these task categories are not always mutually exclusive; in fact, they often overlap. A prominent example is the translation of C code to CUDA, which can be viewed simultaneously as a Code Transpilation task for language migration and a Code Optimization task aimed at unlocking parallel performance on specific hardware.

Beyond these core functions, a significant portion of our survey is dedicated to tasks related to correctness and verification. This focus is directly motivated by a fundamental characteristic of LLMs: they are powerful, but not perfectly reliable, generative models. Unlike traditional deterministic compilers, any workflow that uses an LLM as a direct operator to modify code necessitates a corresponding validation process to ensure the correctness of its output.

Consequently, we include a comprehensive review of **Automated Program Repair** and **Bug Fixes**, where LLMs are employed to correct defects. Furthermore, we cover **Program Synthesis and Code Generation** with a special emphasis on tasks that bolster this verification ecosystem related to compilers, such as generating effective test cases for compiler fuzzing or aiding in the development of the compiler's own source code.

The 159 primary studies within our scope are highly diverse. To bring structure to this landscape, we propose a multi-dimensional taxonomy that classifies research along four key axes:

- **Design Philosophy**, which describes how the LLM is architecturally integrated into the compilation workflow.
- **LLM Methodology**, which details how the model is technically developed and applied to that task.
- **Level of Code Abstraction**, which characterizes the representational level of the code being processed (e.g., source code, IR).
- **Task Type**, which defines the specific compiler-related goal being accomplished (e.g., optimization, transpilation).

To provide a clear analysis, we group these four dimensions based on the research questions they answer.

§ 3 will analyze the first two dimensions, Design Philosophy (the "task side") and LLM Methodology (the "LLM side"), to comprehensively answer **RQ1: How are LLMs integrated?**. Subsequently, § 4 will analyze the final two dimensions, Level of Code Abstraction and Task Type. We observe that these two are highly correlated (e.g., source-to-source transpilation operates at a different abstraction level than IR optimization). We therefore analyze them together to provide a cohesive overview and answer **RQ2: What tasks are addressed?**.

3 Dimension 1&2: Design Philosophy & LLM Methodology

This section addresses our first research question—*How are LLMs being integrated into the compilation process?*—by analyzing two complementary dimensions: **Design Philosophy** and **LLM Methodology**. The former describes the LLM's architectural role, while the latter details the technical methods used to apply the model to that role.

The first dimension, **Design Philosophy**, classifies approaches based on the conceptual role the LLM plays within the broader compilation system. This choice is a critical architectural decision, as it fundamentally determines how the LLM's capabilities are leveraged and constrained. It significantly influences the system's degree of autonomy, its trustworthiness, and the complexity of verifying its outputs. As we will detail, we identify three dominant philosophies: LLM as selector (§§ 3.1), LLM as translator (§§ 3.2), and LLM as generator (§§ 3.3).

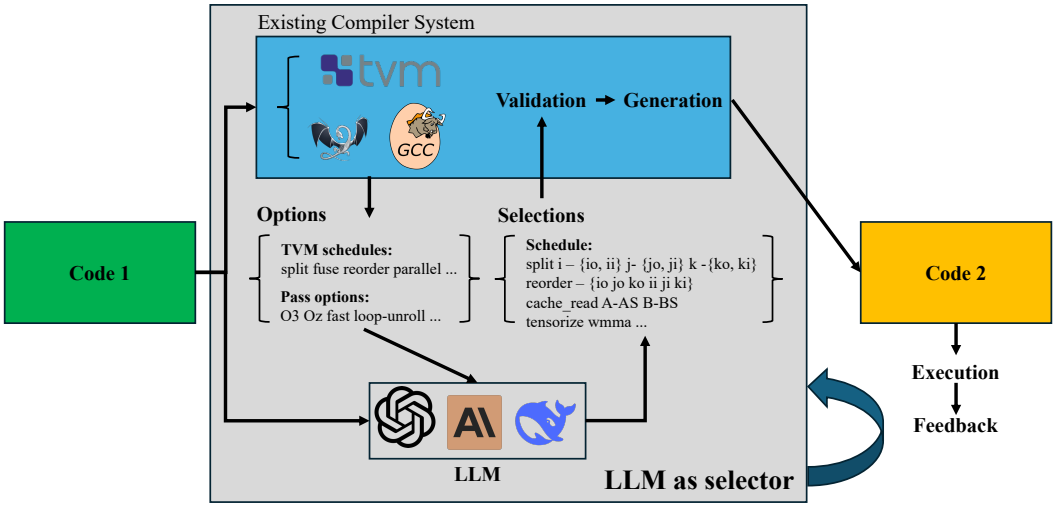

Fig. 2: Overview of LLM-as-Selector methodology

Complementing this "task-side" view, the second dimension, **LLM Methodology**, examines the "LLM-side" of the integration. This dimension addresses the technical methods used to make an LLM capable of its designated task. These methods exist on a spectrum, from "training-free" approaches (e.g., in-context learning, prompt engineering, Retrieval Augmented Generation) to "training-required" adaptations (e.g., fine-tuning, domain-specific pre-training, reinforcement learning). We will analyze these methodologies and their trade-offs in detail in §§ 3.4.

3.1 LLM as selector

In this design, the LLM functions as a sophisticated policy engine or a "hyper-optimizer". As depicted in Figure 2, its primary role is not to generate new code, but to select the best course of action from a predefined and finite set of options or operations provided by the compilation system. The LLM is prompted with the source code context and a set of valid choices, and it uses its deep contextual understanding to make an informed decision. The compilation system itself then performs the corresponding transformation based on the LLM's selection.

This approach is the most direct evolution from traditional ML-in-compiler techniques, with the key difference being the replacement of models trained on handcrafted features with a powerful LLM that can reason directly over raw source code (can be finetuned as well). Common applications include selecting optimal compiler flags or determining the most effective sequence of optimization passes for a given program. The main advantage of this model is its inherent safety and control; since the LLM only chooses from a list of valid, human-defined actions, the resulting transformation is guaranteed to be valid if a valid selection is generated by LLM. However, its creative potential is limited by this predefined search space, and it cannot discover entirely novel optimizations that are not already encoded in the available options.

The works in this category can be broadly grouped based on their primary goal and methodology:

Replacing or Augmenting Compiler Heuristics: This is the most common application, where the goal is to find better compiler flag or pass sequences than the default heuristics (e.g., -O2, -O3). This area has a long history, with earlier works using techniques like NeuroEvolution (Heckel 2023) and Mammadli et al. (2020) to tune compiler heuristics, Tavarageri et al. (2021) using DNNs to optimize polyhedral loop generation strategy or graph-based algorithms (Sajjadinasab et al. 2024) to optimize GCC flag settings. Modern approaches now use LLMs to apply this same principle to more complex, domain-specific areas like selecting optimization strategies for efficient model serving (Tang et al. 2025), optimizing quantum compilation (Ren et al. 2024), or guiding the generation of high-performance tensor programs (Zhai et al. 2024).

Improving the Efficiency of the Search Process: Rather than just selecting the final configuration, several works use the LLM to make the search for good configurations more intelligent and efficient. For example, CompilerDream (Deng et al. 2024) uses an LLM to help build a "world model" of the optimization space, which then guides a more effective search. Others use an LLM for priority sampling (Grubisic et al. 2024) to decide which compiler options are most promising to test, or to enhance black-box fuzzing (Wang et al. 2024) by intelligently mutating flags based on feedback from previous compilation attempts.

Agentic and Reinforcement Learning (RL) Frameworks: The most advanced selector systems employ LLMs as the core of an autonomous agent that can interact with the compilation environment. These agents can perform a series of selections to achieve a goal. Several works use RL for compiler auto-tuning, such as Compiler-R1 (Pan et al. 2025), an agentic framework for exploring the option space, and DeCOS (Cui et al. 2025), which uses an LLM to "ignite" the RL process for more data-efficient learning. Taking this a step further, CompileAgent (Hu et al. 2025) demonstrates a high-level agent that selects and

orchestrates a series of command-line tools (git, make, etc.) to automate the complex task of compiling entire real-world software repositories.

3.2 LLM as translator

Using LLM as translator is the most direct and ambitious application, where the LLM itself acts as a partial or complete translator. It directly performs one or more transformations, such as translation, optimization, or refactoring, on a given scope of a program or code snippet. This philosophy treats code transformation as a generative, sequence-to-sequence task, leveraging the full power of the LLM to rewrite the input program into a new version.

Figure 3 describes an ideal LLM translator system. First of all, the translator must scale to enough code input to be used in real scenarios. Thanks to programming language's composibility nature, a large project-level can be decomposed into multiple function-level code snippets, or even some finer-granularity like basicblocks or statements, with proper context management to make sure the decomposed translation results can be combined again. Later, each code fragment is prompted to LLM to perform code translation, during this stage, multiple techniques could be applied to improve translation quality, for example, Chain-of-Thought prompting (Wei et al. 2022), which decomposes a large translation task into several sequential subtasks to reduce translation complexity. Few-shot examples or retrieval augmented generation (RAG) (Lewis et al. 2020) can improve LLM's hallucination by providing necessary information needed for the translation, while output format of LLM (Macedo et al. 2024) can also impact LLM's generation quality. During the translation, LLM can also generates other useful code, such as test code (if no golden test is provided), and assertions/preconditions needed in verification.

After LLM translation, a post verification is needed to verify some quality-important attributes, for example, check if the code can compile is the most trivial attribute, as for more specific attributes, such as memory safety check, overflow check and for-loop index check, requiring sophisticated verification methods like SMT solver (De Moura and Bjørner 2008) or transform validator like alive2 (Lopes et al. 2021). Translation fails to pass verification will need to be repaired through an automated program repair process using compiler,runtime or behavioral feedbacks.

This is the dominant approach for a variety of source-to-source tasks, which can be categorized as follows:

- **Language Transpilation:** This involves translating code from one high-level language to another. Examples in the literature are diverse, including:
  - Translating from a high-resource language like Java to a low-resource one such as OCaml (Cassano et al. 2024).
  - Converting sequential C into parallel CUDA to leverage GPU architectures (Wen et al. 2022; TehraniJamsaz et al. 2024).
  - Migrating between different parallel paradigms, such as from SIMT-parallel CUDA to SIMD-parallel BANG C (Dong et al. 2025).
- **Code Optimization:** This focuses on rewriting specific code segments to improve performance or other non-functional properties. A common example is automatically rewriting `for`-loop pragmas to achieve better performance on a given target hardware (Taneja et al. 2025).
- **Automated Program Repair:** In this context, bug fixing is treated as a translation problem. The goal is to leverage the LLM to translate a flawed program into a semantically correct version (Wei et al. 2023; Xia et al. 2023).

The key advantage of this methodology is its immense potential to learn and generate complex, non-trivial transformations that may surpass human-designed rule-based translators. However, the primary disadvantage is the significant challenge of ensuring correctness. The probabilistic nature of LLMs means they can "hallucinate" and produce code that is syntactically correct but semantically flawed, making rigorous verification a critical and difficult component of this approach.

Notably, we only list some remarkable studies in this category and will detail the rest in § 4, as the level of code abstraction can help categorize the major studies of this survey better.

3.3 LLM as generator

This is a more indirect, meta-level approach where the LLM's role is to generate the source code of a program that, in turn, performs the desired code transformation. As depicted in Figure 4, the workflow is typically a two-step process: first, the LLM writes a script or program that implements the transformation logic, which can be seen as a compiler development process if applied to a compiler system development; second, this generated program is compiled and executed to apply the changes to the target code, which can also be seen as a compiler testing process.

Currently, this design philosophy is most popularly realized in the form of AI-powered code assistants and agents, which generate functional code snippets, scripts, or entire applications based on natural language prompts. Prominent examples include tools like GitHub Copilot (Chen et al. 2021b), Cursor (Anysphere, Inc. 2023), and command-line interfaces applications like gemini-cli (google-gemini 2025) and claude-code (anthropics 2025).

The same principle can be extended to the compiler domain, where an LLM could theoretically generate new optimization passes or other compiler components. However, due to the immense complexity and the need for deep, specialized knowledge of compiler internals, data structures, and APIs, applying this approach

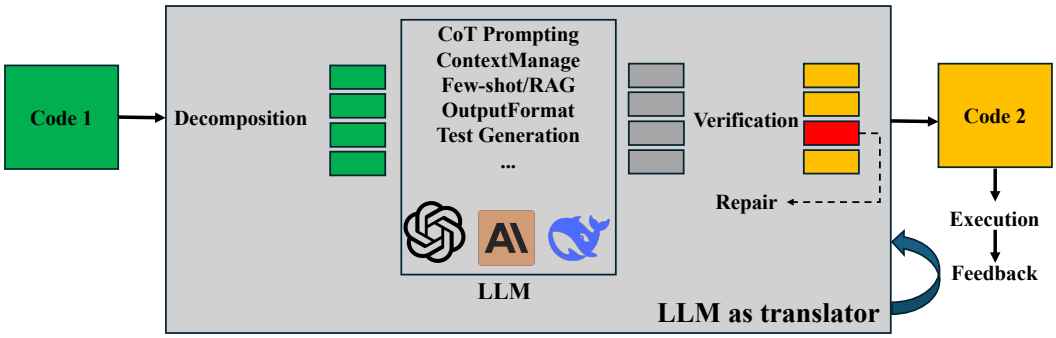

Fig. 3: Overview of LLM-as-Translator methodology

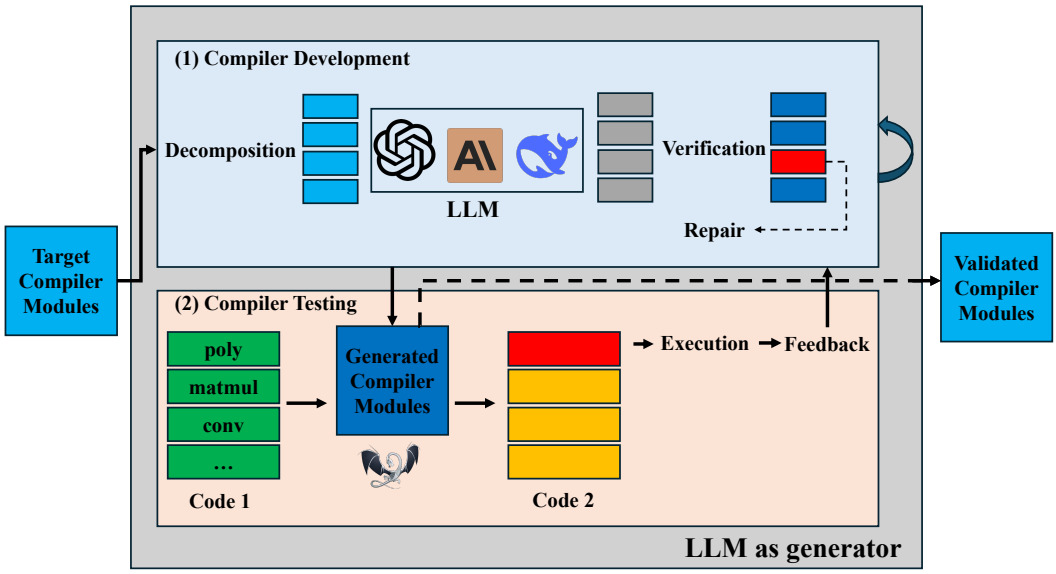

Fig. 4: Overview of LLM-as-Generator methodology

effectively is a significant challenge. As a result, well-designed studies that successfully employ the LLM-as-generator paradigm for core compiler tasks are still scarce. Nonetheless, some pioneering works have begun to explore this frontier. For example, VEGA (Zhong et al. 2025) demonstrates a method for automatically generating compiler backends by using a pre-trained transformer model, while ComBack (Zhong et al. 2024) provides a versatile dataset specifically designed to facilitate research on generating and enhancing compiler backend code. On the source code side, CodeTransform (Cummins et al. 2024) also preliminarily explores the code rewriting potential, in a word: *Don't Transform the Code, Code the Transforms.*

This hybrid approach combines the pattern-recognition strength of LLMs with the rigor of a traditional, deterministic program. The main advantage is that the generated artifact—the transformation script—can be inspected, verified, and reused, offering a higher degree of trust than a direct generative model. The primary challenges are the increased complexity of the workflow and the requirement for the LLM to possess a sophisticated understanding of the specific compiler APIs and frameworks it needs to use in the code it generates.

In summary, the design philosophy dictates the fundamental architecture of an LLM-enabled compiler. The three paradigms discussed—**Selector**, **Translator**, and **Generator**—represent a spectrum of integration strategies, each presenting a distinct trade-off between control and creative potential. The Selector model offers the highest degree of safety by operating within a predefined action space, while the Translator model unleashes the full generative power of LLMs to discover novel transformations, albeit with significant verification challenges. The Generator provides a hybrid approach, balancing generative flexibility with the determinism of traditional code system.

3.4 LLM Methodology

Complementing the architectural *Design Philosophy*, *LLM Methodology* details the technical methods used to make a model capable of its designated task. This dimension provides a crucial comparative analysis of *how* LLMs are technically employed, distinguishing the "LLM-side" development from the "task-side" role. These methods can be broadly categorized into two main methodologies: **Training-Required** approach that adapt the model's weights, and **Training-Free** approach that guide a pre-trained model's inference

Aspect	Training-Required	Training-Free
Core Principle	Adapt Parameters	Guide Inference
Primary Cost	Data Curation Training Computation	Prompting Effort Inference w/ Extra Context
Performance	High (Specialized)	Varies (Generalized)
Flexibility	Low (Task-Locked)	High (Adaptable)
Knowledge Source	Internal (Weights)	External (Context)
Typical Examples	• Domain Pre-training LLMCompiler (Cummins et al. 2025) • Supervised Fine-Tuning VirtualCompiler (Gao et al. 2024) • Reinforcement Learning CUDA-L1 (Li et al. 2025)	• Prompt Engineering LEGOCompiler (Zhang et al. 2025) • Retrieval Augmented Generation CoCoGen (Bi et al. 2024) • Agentic Workflows Compiler-R1 (Pan et al. 2025)

Table 1: Qualitative Comparison of LLM Methodologies

process. The choice between them involves a fundamental trade-off between the cost of data curation and training versus the complexity of prompt engineering and inference-time systems.

3.4.1 Training-Required: Adapting Model Weights

These methods modify the LLM's parameters to instill domain-specific knowledge and specialize its behavior. This is often necessary when the task is highly specialized or when the target representation (like compiler IR) is scarce in the model's original pre-training data.

- **Domain-Specific Pre-training:** This is the most resource-intensive approach, where a model is trained from scratch or undergoes continued pre-training on a massive, domain-specific corpus. This is essential for tasks operating on representations unseen by general models. For example, **Meta LLMCompiler** (Cummins et al. 2025) created foundation models pre-trained specifically on LLVM IR, while Zhai et al. (2024) trained their **TLM** model on a large corpus of TVM's search space to act as a selector. The **TransCoder** series (Roziere et al. 2020; anne Lachaux et al. 2021; Roziere et al. 2022; Szafraniec et al. 2023) also rely on this to learn cross-lingual representations before tackling translation.
- **Supervised Fine-Tuning (SFT):** This is the most common adaptation methodology. It takes a general-purpose, pre-trained foundation model (e.g., CodeLlama, GPT) and fine-tunes it on a curated dataset of "input-output" examples for a specific task. This adaptation can be performed using full finetuning or, more commonly, through parameter-efficient finetuning (PEFT) techniques like LoRA. This approach has proven highly effective across various tasks, such as fine-tuning models for C-to-x86 neural compilation (Zhang et al. 2024), Verilog generation (Thakur et al. 2024), and source-level optimization (Xu et al. 2025). **VirtualCompiler** (Gao et al. 2024) also leverages SFT to translate source code to assembly by matching semantics. The **CUDA-L1** study (Li et al. 2025) also begins with SFT to teach the model the basics of CUDA optimization.
- **Reinforcement Learning (RL) & Feedback-Based Tuning:** When a task has a clear, non-differentiable metric for success (e.g., performance speedup, code size reduction, or passing a test suite), RL or other feedback mechanisms can be used to optimize the model directly for that objective. This is often applied after SFT. For instance, **PerfRL** (Duan et al. 2025) uses RL with metric feedback for code optimization. Similarly, **VerilogLLM** (Wang et al. 2025) uses feedback from testbenches, **CoTran** (Jana et al. 2024) uses compiler and symbolic execution feedback, and **CUDA-L1** (Li et al. 2025) employs GRPO (Shao et al. 2024) to surpass its initial SFT performance. **DeCOS** (Cui et al. 2025) also uses an LLM to "ignite" an RL process for data-efficient learning.

3.4.2 Training-Free: Guiding Model Inference

These methods leverage a powerful, general-purpose foundation model's existing capabilities without altering its weights. The focus shifts from data curation and training to designing sophisticated inference-time systems that provide the model with the necessary context and guidance.

- **Prompt Engineering:** This involves crafting a detailed prompt that instructs the model (zero-shot) or provides it with a few in-context examples (few-shot learning) of the task. For more complex reasoning tasks, this is often extended to multi-step prompting strategies like Chain-of-Thought (CoT). For example, **LEGOCompiler** (Zhang et al. 2025) employs both few-shot learning and CoT-prompting to perform neural compilation, **CodeOptCoT** (Xu et al. 2024) and **SBLLM** (Gao et al. 2025) both explicitly leverage CoT to improve code optimization by forcing the model to "think step-by-step." Similarly, **RTLLM** (Lu et al. 2024) uses a "self-planning" prompting strategy, and **BuiltRome** (Nakkab et al. 2024) also finds that hierarchical prompt

structuring is critical for LLM-based hardware design.

- **Retrieval-Augmented Generation (RAG):** To combat hallucinations and provide the model with specialized knowledge it may not have been trained on (e.g., obscure APIs, project-specific context, or optimization rules), a RAG system coule be used. This system first retrieves relevant documents from an external knowledge base and adds them to the model's context. This is used by **Autoiot** (Shen et al. 2025) for background knowledge, **CoCoGen** (Bi et al. 2024) for project-level context retrieval, and **SBLLM** (Gao et al. 2025) for retrieving optimization knowledge.
- **Agentic & Iterative Workflows:** This advanced methodology treats the LLM as a reasoning engine within a larger loop. The "agent" can plan, use external tools (like compilers, verifiers, or SMT solvers), and iteratively refine its output based on feedback from these tools. This approach is central to systems like **Compiler-R1** (Pan et al. 2025) and **CompileAgent** (Hu et al. 2025). It is also the core principle behind iterative repair loops, such as those in **DecLLM** (Wong et al. 2025) and **ProblemOriented** (Ye et al. 2024), and multi-agent frameworks like **WhiteFox** (Yang et al. 2024). **Qimeng-Xpiler** (Dong et al. 2025) also uses an MCTS-based search, which can be seen as a form of guided, iterative generation.

3.4.3 Methodology Comparison

The choice between Training-Required and Training-Free methodologies presents a fundamental design trade-off, which we summarize qualitatively in Table 1.

Training-Required methods focus on **adapting model parameters (weights)**, which incurs significant upfront **cost** in data curation and training computation. However, this internalizes domain knowledge, leading to high performance on specialized or narrow tasks (e.g., Meta LLMCompiler (Cummins et al. 2025) is specialized on LLVM IR code size optimization). The trade-off is low flexibility, as the resulting model is "locked-in" to its specific trained task.

Conversely, Training-Free methods **guide a fixed model's inference** process. This shifts the **cost** to prompt engineering and inference-time resources. These approaches offer high flexibility, as prompts and tools can be quickly adapted. Their performance is more generalized and highly dependent on the base model's capabilities, with knowledge externalized and injected at runtime via context (e.g., Xu et al. (2024), Gao et al. (2025), Hu et al. (2025)).

In practice, these methodologies are not mutually exclusive. A common pattern is to use a *Training-Required* method (like SFT) to create a specialized model, which is then deployed within a *Training-Free* methodology (like an agentic workflow with RAG). This combination leverages the model's specialized knowledge while simultaneously grounding its output with real-time context and verification.

Finally, combining the analyzed two dimensions, we can now provide a comprehensive answer to our first research question (**RQ1**), as summarized below.

> **RQ1**: How are LLMs being integrated into the compilation process?
> **Answer**:
> On the task side, LLMs have been integrated into compilation systems through three primary design philosophies that define their role: (1) as a **Selector** to choose from predefined compiler actions (§§ 3.1), (2) as a direct **Translator** to rewrite code (§§ 3.2), or (3) as a **Generator** to create new compiler tools and components (§§ 3.3).
> On the LLM side, these roles are realized through two primary **LLM Methodologies**: (1) **Training-Required** methods (e.g., fine-tuning, RL) that adapt model weights for specialized tasks, and (2) **Training-Free** methods (e.g., prompt engineering, RAG, agentic workflows) that guide a general model's inference.

4 Dimension 3&4: Level of Code Abstraction & Task Type

This section details the third and fourth dimensions of our taxonomy: the Level of Code Abstraction and the specific Task Type. We analyze these two axes together as they are deeply intertwined; the level of program representation fundamentally dictates the nature of the tasks that can be performed. As illustrated in Figure 5, we categorize these representations into three primary strata:

- Natural Language (NL), the highest level of human intent;
- High-Level Programming Language (PL), the source code developers write;
- Low-Level Intermediate Representation and Assembly (ASM), the machine-centric representations used by the compiler backend and executed by computers.

The choice of abstraction level fundamentally dictates the nature of the tasks that can be performed, which in turn forms the third dimension of our taxonomy. In this section, we first detail the tasks that operate within a single level of abstraction (intra-level transformations) and then discuss the more complex tasks that bridge these different levels (cross-level transformations).

4.1 Intra-Level Transformations

Intra-level transformations are those where the input and output of the process remain at the same level of abstraction. These tasks typically focus on refinement, optimization, or migration within a given representation.

Acronym	Citation	Tasks	Code level
Qimeng-Xpiler	Dong et al. (2025)	Transpile	VNNI,CUDA,HIP,BANG
G-TransEval	Jiao et al. (2023)	Transpile	C++,Java
CoTran	Jana et al. (2024)	Transpile	Java,Python
Oxidizer	Zhang et al. (2025)	Transpile	Go-Rust
OpenCLGen	Palkowski and Gruzewski (2024)	Transpile	PolyC-OpenCL
LLMLift	Bhatia et al. (2024)	Transpile	C,C++,Java-DSLs
Rectifier	Yin et al. (2024)	Transpile	C++,Java,Python
AlphaTrans	Ibrahimzada et al. (2025)	Transpile	Java-Python
OutputFormat	Macedo et al. (2024)	Transpile	C,C++,Go,Java,Python
LostInTranslation	Pan et al. (2024)	Transpile	C,C++,Go,Java,Python
KnowTransfer	Cassano et al. (2024)	Transpile	Java,Python,JS-Ocaml,Racket
CanLLMParallel	Nichols et al. (2024)	Transpile	C++-MPI+OpenMP
SALLM	Siddiq et al. (2024)	Transpile	python+test
TransCoder	Roziere et al. (2020)	Transpile	C++,Java,Python
DOBF	anne Lachaux et al. (2021)	Transpile	C++,Java,Python
TransCoder-ST	Roziere et al. (2022)	Transpile	C++,Java,Python
TransCoder-IR	Szafraniec et al. (2023)	Transpile	LLVM IR-C++,Java,Rust,Go
BabelTower	Wen et al. (2022)	Transpile	C-CUDA
CodeRosseta	TehraniJamsaz et al. (2024)	Transpile	C-CUDA
HPCTrans	Lv et al. (2025)	Dataset Generator	C-CUDA
UnsuperBinTrans	Ahmad and Luo (2023)	Binary Transpile	ARM-x86
TFix	Berabi et al. (2021)	Code Repair	JavaScript
LLMAPR	Xia et al. (2023)	Code Repair	Java,Python,C
ChatGPTRepair	Zhang et al. (2023)	Code Repair	Java
EISP	Chen et al. (2024)	Code Repair	Python,JavaScript
MacroConfig	Albuquerque et al. (2024)	Code Repair	C/C++/Java
CoCoGen	Bi et al. (2024)	Code Repair	Python
ZS4C	Kabir et al. (2025)	Code Repair	Python
Repilot	Wei et al. (2023)	Code Repair	Java
RustAssistant	Deligiannis et al. (2024)	Code Repair	Rust
LIBRO	Kang et al. (2023)	CVE test Generation	Defects4J
SecurityTestGen	Zhang et al. (2023)	CVE test Generation	Java CVE test
GeneticImprove	Brownlee et al. (2023)	Compiler Fuzzing	Java
BenchDirect	Tsimpourlas et al. (2023)	Compiler Fuzzing	OpenCL
WhiteFox	Yang et al. (2024)	Compiler Fuzzing	PT-Inductor/XLA/TF-Lite
MetaMut	Ou et al. (2024)	Compiler Fuzzing	C/C++
ClozeMaster	Gao et al. (2025)	Compiler Fuzzing	Rust
FMCSO	Italiano and Cummins (2025)	Compiler Fuzzing	C/C++
CORL	Mammadli et al. (2020)	Pass Optimization	LLVM pass
GraphFlagOpt	Sajjadinasab et al. (2024)	Pass Optimization	GCC pass
NeuroEvolution	Heckel (2023)	Pass Optimization	LLVM pass
CompilerR1	Pan et al. (2025)	Pass Optimization	LLVM pass
TLM	Zhai et al. (2024)	Autotuning Optimization	TVM IR
ReasoningCompiler	Tang et al. (2025)	Autotuning Optimization	TVM IR
Effi-Learner	Huang et al. (2024)	Code Optimization	Python
OMPar	Kadosh et al. (2024)	Code Optimization	C/C++ + OpenMP
ProblemOriented	Ye et al. (2024)	Code Optimization	C++
CodeOptCoT	Xu et al. (2024)	Code Optimization	Python
LangProp	Ishida et al. (2024)	Code Optimization	Python
AutoComp	Hong et al. (2025)	Code Optimization	C-C+Intrinsic
SBLLM	Gao et al. (2025)	Code Optimization	Python,C++
PCAOT	Romero Rosas et al. (2025)	Code Optimization	C+OpenMP
CompilerGPT	Pirkelbauer and Liao (2025)	Code Optimization	C/C++
Perfcodegen	Peng et al. (2025)	Code Optimization	Python
SpeedGen	Purschke et al. (2025)	Code Optimization	Python
CodeOPT	Xu et al. (2025)	Code Optimization	C/C++
PerfRL	Duan et al. (2025)	Code Optimization	C++,Java,Python
ECO	Lin et al. (2025)	Code Optimization	C++
CUDA-L1	Li et al. (2025)	Code Optimization	CUDA/Pytorch
LLMVectorizer	Taneja et al. (2025)	Code Optimization	C/C++
RACL	Wang et al. (2025)	Code Optimization	C/C++
CodeTransform	Cummins et al. (2024)	Code Optimization	Python

Table 2: Summary of LLM for intra-level code transformation

Acronym	Citation	Tasks	Code level
NatGen	Chakraborty et al. (2022)	Code Optimization	Java,Python
CodeOptEdu	Rong et al. (2025)	Code Optimization	Python
RTLrewriter	Bhatia et al. (2024)	Code Optimization	RTL
SymRTLO	Wang et al. (2025)	Code Optimization	RTL
InstCombiner	Mannarswamy and Das (2022)	ASM optimization	arm
peephole	Fang and Mukhanov (2024)	ASM optimization	arm
VeriLOCC	Jin et al. (2025)	ASM optimization	SASS,RDNA
CompilerDream	Deng et al. (2024)	IR Optimization	LLVM IR
MetaLLMCompiler	Cummins et al. (2025)	IR Optimization	LLVM IR
PrioritySampling	Grubisic et al. (2024)	IR Optimization	LLVM IR
DeCOS	Cui et al. (2025)	IR Optimization	LLVM IR

Table 2: (Continued)

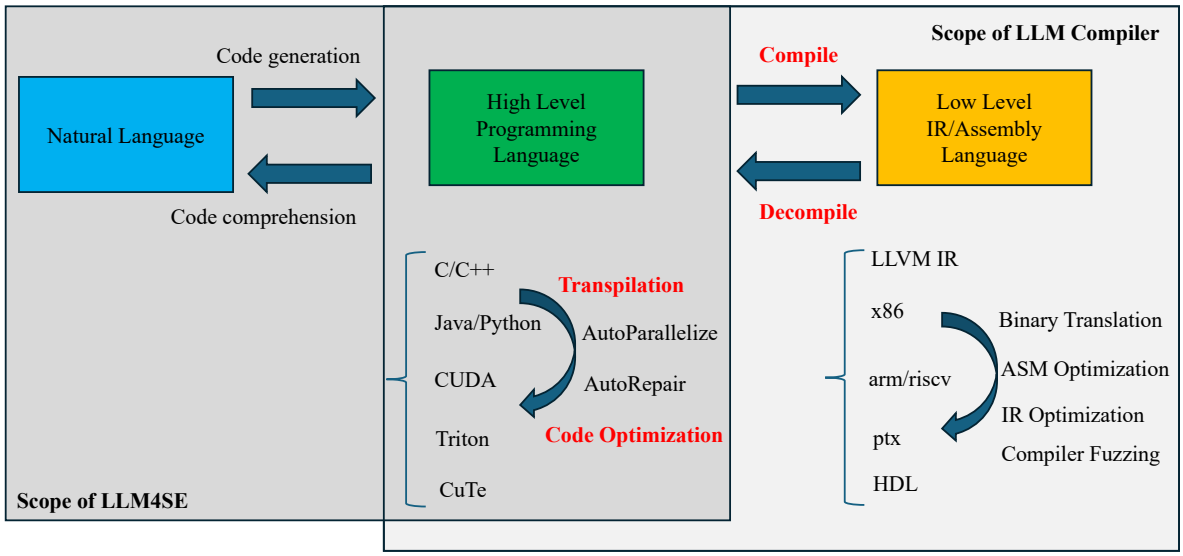

Fig. 5: Code level view of LLM Compiler

- **High-Level Programming Language (PL) Transformations:** This is currently the most active area of research, focusing on source-to-source transformations, a joint hotspot for both software engineering, programming language and LLM communities. LLMs are applied to a wide array of PLs, from general-purpose languages like C/C++, Java, and Python to specialized, domain-specific languages for parallel computing such as CUDA, Triton (Tillet et al. 2019), and CuTe (NVIDIA Corporation 2025). Key tasks at this level include:
  - Transpilation: Translating between different PLs, a task crucial for migrating legacy codebases or converting programs between different programming ecosystems.
  - Code Repair: A major goal of production code design is maintainability, therefore, LLM-generated code should be tested more or less, and automated test-generation, or fuzzing test is an important direction. Besides, the flawed code should be automated repaired with minimal human intervention.
  - Source Optimization: Rewriting PL code to improve its quality, readability, or performance without changing the language.
- **Low-Level IR / Assembly Transformations:** At this level, LLMs perform transformations on machine-centric representations, often invisible to the original programmer but critical for final performance. These operate on common compiler IRs like LLVM IR and instruction set architectures such as x86, PTX, ARM, and RISC-V. The main tasks include:
  - IR Optimization: Applying optimization passes directly on an intermediate representation to improve the efficiency of the generated code.
  - Assembly Optimization: Applying assembly-level optimizations like register-allocation, instruction scheduling and peephole optimizations to optimize the code performance or code size.
  - Binary Translation: Translating low-level code from one instruction set architecture (ISA) to another, for example, from x86 to ARM.

As summarized in Table 2, the application of LLMs to intra-level code transformation has become a highly active area of research. These tasks, which operate within the same level of abstraction (e.g., source-to-source), primarily leverage the generative capabilities of LLMs to rewrite programs. In this section, we categorize and review these studies based on their primary objective: Code Transpilation, which focuses on migrating between languages; Code Repair, which aims to correct defects; and Code Optimization, which seeks to improve program performance.

4.1.1 Transpile

Code transpilation, or source-to-source translation, is one of the most prominent tasks in this domain. The core challenge is to accurately translate a program from a source language to a target language while preserving its semantic correctness and functionality. This capability is critical for modernizing legacy codebases, improving interoperability between systems, and migrating applications to new hardware ecosystems, such as converting sequential C code to parallel CUDA for GPUs. The following studies demonstrate the breadth of this task, tackling a wide range of language pairs and programming paradigms.

OpenCLGen (Palkowski and Gruzewski 2024) uses LLM to translate polyhedral C kernels into OpenCL. Rectifier (Yin et al. 2024) introduces an LLM correcter to handle different transpilation errors to improve the code transpilation accuracy. OutputFormat (Macedo et al. 2024) studies the impact of LLM output format in code translation tasks. LostInTranslation (Pan et al. 2024) empirically summarizes bugs introduced by LLM transpilation into detailed categories across multiple programming languages. SALLM (Siddiq et al. 2024) benchmarks the capabilities of LLMs to generate secure Python code by examining the Common Weakness Enumeration (CWE)-related test generation.

Besides using LLMs directly, earlier work focuses on a data-centric problem: *how to obtain parallel code corpora to train a model?* To solve this, TransCoder-series (Roziere et al. 2020; anne Lachaux et al. 2021; Roziere et al. 2022; Szafraniec et al. 2023) have established how to learn a transpiler language model unsupervisedly: (1) using mask language modeling (MLM) to learn cross-lingual languages first, (2) using back translation (BT) to learn the translation rules later. Other techniques like denoising autoencoding (Roziere et al. 2020), deobfuscation (anne Lachaux et al. 2021), unittest filtering (Roziere et al. 2022) and utilizing compiler representations (Szafraniec et al. 2023) gradually improve the learned transpiler's capability. These studies are typically between general programming languages like C++, Java and Python.

Besides transpiling between general programming languages, BabelTower (Wen et al. 2022) proposes an improved TransCoder-based approach to unsupervisedly learn a transpiler between C++ and CUDA with new parallel metrics. CodeRosseta (TehraniJamsaz et al. 2024) further improves BabelTower with AST Entity Recognition and customized denoising auto-encoding. Now these learning-based methods are more used as dataset generator for more powerful LLM training. Besides the learning-based approaches, HPCTrans (Lv et al. 2025) modifies the AI compiler TVM (Chen et al. 2018) to auto-generate semantically equivalent CUDA and C code without TVM dependency, thereby creating rich CUDA-to-CPU transpilation corpora.

CanLLMParallel (Nichols et al. 2024) studies the auto-parallel capability within LLMs by using LLM to translate sequential C++ code into parallel code like MPI, OpenMP and CUDA.

CoTran (Jana et al. 2024) proposes a RL-based feedback mechanism using compiler feedback and symbolic execution to improve both compilation accuracy and functional accuracy in Python-to-Java translation.

AlphaTrans (Ibrahimzada et al. 2025) proposes a LLM-based project-level transpiler within GraalVM to support partial transpilation verification, they also implement a divide-and-conquer strategy to decompose both source code and test to achieve scalable Java-to-Python transpilation. Oxidizer (Zhang et al. 2025) also presents a LLM-based Go-to-Rust transpiler for entire projects, where a project partition module divides-and-conquers the translation complexity and a type-checking feature mapping module handles to migrate semantics between languages.

KnowTransfer (Cassano et al. 2024) proposes that transpilation to low-resource languages (e.g. Ocaml, Racket) can be improved and verified with deterministic test case transpilation from high-resource languages (e.g. Java, Python), thereby synthesizing rich bilingual corpora effectively.

LLMLift (Bhatia et al. 2024) proposes that LLM can be used to transpile sequential programs into Tensor-processing framework DSLs like PyTorch and NumPy, with Floyd-Hoare Logic (FHL) to validate generated programs.

Qimeng-Xpiler (Dong et al. 2025) uses LLMs to transpile parallel code between different architectures, including SIMT GPUs (CUDA and HIP), SIMD CPU (C+VNNI) and SIMD NPU (BANG C), using SMT-solver to check key transpilation results correctness and Monte-Carlo Tree Search (MCTS) to explore higher performance candidates within the target architecture.

Besides source code transpilation, there are also some interesting work like UnsuperBinTrans (Ahmad and Luo 2023) that performs binary code translation to preliminarily study the vulnerability discovery problem in a low-resource ISA from high-resource ISA (e.g. x86). However, many binary level work cannot be directly neurally transpiled due to code difficulties, instead, many work focuses on other utility tasks, such as binary similarity detection or compiler property identification, which we will detail in §§ 4.3.

4.1.2 Code Repair

Beyond translating between different languages, another critical application is translating a program

from a flawed state to a correct one. Automated Program Repair (APR) using LLMs treats bug fixing as a specialized translation task: from an incorrect source program to a corrected version in the same language. This approach aims to automate the often tedious and error-prone process of debugging by leveraging the model's learned knowledge of common programming mistakes and their corresponding fixes. The works reviewed here showcase various strategies for applying LLMs to automatically identify and repair bugs. Additionally, generating specialized tests to detect bugs (fuzzing) is also highly-related to this category, which can be seen as either an intra-level code transformation (sythesizing tests) task or a utility-based task.

TFix (Berabi et al. 2021) learns a transformer model to perform specialized sequence-to-sequence code repair task. LLMAPR (Xia et al. 2023) perform the first extensive study on directly applying LLMs for APR, suggesting that both sample size increase and incorporation with fix template information can help improve LLM-based APR. ChatGPTRepair (Zhang et al. 2023) reveals an important overlooked data leakage issue of automated program repair (APR), and finds ChatGPT is more powerful in APR on its proposed decontaminated benchmark EvalGPTFix than PLBart and CodeT5. EISP (Chen et al. 2024) introduces a test-free semantic mistakes localization framework using LLM-based static-analysis. MacroConfig (Albuquerque et al. 2024) studies LLM's capabilities in resolving configurable macros errors with compilation feedback. CoCoGen (Bi et al. 2024) uses an iterative generation and verification process to do project-level code repair with careful context retrieval. ZS4C (Kabir et al. 2025) proposes a zero-shot LLM-based synthesizer to autocomplete incomplete code to a compilable one.

Except using LLMs for automated code repair, LLMs can also be used in a repair assistant way. Repilot (Wei et al. 2023) uses LLM as completion engine for program repair by aggresively synthesizing valid patches during the repair process. RustAssistant (Deligiannis et al. 2024) is another assistant tool to suggest potential Rust code fixes based on compiler feedbacks.

On the fuzzing test side, LIBRO (Kang et al. 2023) showcases LLMs can be used to reproduce bugs by synthesizing test programs from bug reports. SecurityTestGen (Zhang et al. 2023) also studies the capabilities of LLMs to generate security tests from CVEs.

GeneticImprove (Brownlee et al. 2023) finds that LLMs can be used as mutation operators combined with genetic improvment to generate diverse valid programs. BenchDirect (Tsimpourlas et al. 2023) trains a language model to generate compiler testing benchmarks with high readability. WhiteFox (Yang et al. 2024) adopts a multi-agent framework to automatically synthesize white-box compiler fuzzing tests. MetaMut (Ou et al. 2024) proposes to guide an LLM as mutator, it uses LLM to fill-in a carefully crafted templates to generate non-trivial mutator designs, then randomly applied to test programs to synthesize fuzzing programs, which successfully harvested 131 GCC/Clang compiler bugs. ClozeMaster (Gao et al. 2025) similarly uses LLMs to do fuzzing tests for Rust compiler, where it first generates cloze-masked code snippets then fill-in-the-blank to generate diverse compiler test programs. FMCSO (Italiano and Cummins 2025) presents a mutation testing methodology by using LLMs to iteratively modify starting code seed then develop differential testing strategies to find missing code size optimizations in LLVM compilers.

4.1.3 Code Optimization

The third major category of intra-level transformation is Code Optimization. This can be viewed as translating a program from a semantically correct but sub-optimal version to an improved version that is more efficient in terms of performance, memory usage, or code size. While this is a classic compiler goal, the application of LLMs is particularly broad in this domain. Unlike transpilation and repair which primarily operate on source code, optimization tasks are being explored at multiple levels of abstraction. This includes not only high-level, source-to-source rewriting but also critical low-level tasks such as IR Optimization and assembly-level tuning. Across all these levels, LLMs offer a novel, data-driven approach to discovering complex optimization strategies that may be difficult for traditional heuristic-based compilers to identify.

First, like previous studies in ML-powered compiler studies, using language model as selector is also feasible and can be learned through proper feature engineering. CoRL (Mammadli et al. 2020), NeuroEvolution (Heckel 2023), GraphFlagOpt (Sajjadinasab et al. 2024) and CompilerR1 (Pan et al. 2025) are used as selector to optimize compiler pass ordering. TLM (Zhai et al. 2024) uses an LLM trained from scratch to generate predicted schedule sequences from the large schedule space to accelerate TVM autotuning searching. ReasoningCompiler (Tang et al. 2025) similarly uses MCTS to explore the the search space through LLM reasoning and also integrated to TVM to accelerate the sampling efficiency.

Besides using LLM as selector, there are more work exploring to use LLM to perform generative optimizations, acting as a direct translator, where the vast pretrained code knowledge in LLM is believed to guide the LLM to perform reasonable optimizations. The majority of these optimizations happen at the source code level.

Effi-Learner (Huang et al. 2024) proposes a self-optimization framework utilizing execution overhead profiles to improve the efficiency of LLM-generated code. OMPar (Kadosh et al. 2024) studies the capability of OpenMP pragma generation to auto-parallelize C/C++ code. ProblemOriented (Ye et al. 2024) proposes an anchor verification mechanism, first synthesizing test inputs based on slow code, constructing verified test case through executing with slow code, and iteratively refining the optimized code with execution feedback. CodeOptCoT (Xu et al. 2024) applies the Chain-of-Thought techniques to augment the structure

understanding and self-checking to improve code optimization. LangProp (Ishida et al. 2024) has a specific focus on autodriving code optimization in CARLA.

In 2025, studies within this task have boomed. AutoComp (Hong et al. 2025) studies using LLM to optimize C code into different tensor accelerators with distinct intrinsics. SBLLM (Gao et al. 2025) proposes a search-based LLM framework using optimization knwoledge retrieval and genetic operator-inspired CoT prompting. PCAOT (Romero Rosas et al. 2025) compares LLM-based optimization with traditional optimization compilers and finds LLMs struggle to optimize large programs (measured in LOCs) directly. CompilerGPT (Pirkelbauer and Liao 2025) proposes using LLMs to analyze and act on compiler optimization reports, making tailored code rewriting optimizations to fit the optimization reports.

Perfcodegen (Peng et al. 2025) proves that small LLMs with proper execution feedback can generate performant code compared to naively prompted flagship LLMs. SpeedGen (Purschke et al. 2025) uses LLM to optimize Python code performance by rewriting to use high performance libraries like PyTorch and NumPy. CodeOPT (Xu et al. 2025) finetuned a LoRA adapter to perform optimization strategies like loop unroll, inline expansion, constant folding and dead code elimination in the source code level in C/C++. PerfRL (Duan et al. 2025) trains a small language model within a reinforcement learning environment with direct metric feedback to obtain better code optimization results.

ECO (Lin et al. 2025) uses historical commits to record anti-patterns addressed, and using a finetuned LLM to automatically refactor code, auto-verify, and submit to code review, constructing an automated code optimization pipeline that scales to warehouse-scale computers in Google production. CUDA-L1 (Li et al. 2025) reveals a remarkable capability of RL in autonomous learning for CUDA optimization, with a SFT+GRPO (Shao et al. 2024) finetuned model, capable of generating 249 out of 250 KernelBench (Ouyang et al. 2025) cases and optimizing 240 of them better than native PyTorch. LLMVectorizer (Taneja et al. 2025) studies to auto-vectorize C code with SIMD intrinsics in source code level and formally verified through symbolic verification with Alive2 (Lopes et al. 2021). RACL (Wang et al. 2025) uses reductive analysis to divide-and-conquer program complexity and perform input-centric code optimizations, significantly reduces profiling efforts.

CodeTransform (Cummins et al. 2024) explores to preliminarily use LLMs to code the transform instead of directly transforming code, using the LLM-as-generator methodology.

Besides pure performance optimization, there are work like NatGen (Chakraborty et al. 2022) which introduces a novel task of "code naturalization" to transform unnatural code into natural one that fits programming paradigms, where it pretrained a LM to naturalize code well in Java and Python. Other work like CodeOptEdu (Rong et al. 2025) is used to refine code for educational purposes.

4.2 Cross-Level Transformations

Cross-level transformations are those that bridge the different strata of abstraction. These tasks are fundamental to the very definition of a compiler and its related tools, representing some of the most complex and impactful applications of LLMs in this domain. As outlined in Table 3, we will list these cross-level transformations in the following section.

4.2.1 Code Generation: NL-PL

This is the process of translating a Natural Language specification into a structured Programming Language program. In this role, the LLM acts as the ultimate compiler front-end, directly converting a developer's description of a problem into a working solution.

ANPL (Huang et al. 2023) introduces an interactive way to decompose program generation problem to sketch and holes, and fill later with flexible user interruption.

AIOSCompiler (Xu et al. 2024) uses LLM as interpreter for natural language programming and flow programming of AI agents, where they introduce a domain-specific language CoRE to unify natural language programming, pseudo-code programming and flow programming. Unicoder (Sun et al. 2024) also proposes a code representation to unify different programming languages and is LLM-friendly for code generation.

Autoiot (Shen et al. 2025) focuses on AIoT applications code generation using natural language programming, supported by a background knowledge retrieval module, a CoT program synthesis module and an automated code improvement module.

ArabicLLMCompiler (Sibaee et al. 2024) uses LLM as interpreter to translate Arabic-based programming languages into executable Python code.

The reverse of code generation: code comprehension is also an important task for software engineering, however, as our study has a compiler-centric focus, studies with code comprehension goal are therefore treated as out of scope in this survey.

4.2.2 Compilation: PL-ASM

This is the core classic compilation task, where an LLM is used to translate a human-readable PL into a low-level representation like IR or assembly language. This is a core challenge where LLMs are being explored to augment or even replace components of traditional compiler backends. Some similar subtasks, like high-level synthesis, HDL generation, also fall into this category.

Transformer-x86 (Armengol-Estape and O'Boyle 2021) preliminarily studies to learn a transformer from compiler-generated C-x86 corpora. Transformer-llvm (Guo and Moses 2022) similarly studies C-LLVM and C-x86 with optimization level setting. Neither of them achieve fair translation accuracy, typically for optimized setting, the translation accuracy is 0%.

VirtualCompiler (Gao et al. 2024) models the LLM compilation as similar task of assembly code search and

Acronym	Citation	Tasks	Code level
ANPL	Huang et al. (2023)	Code Generation	NL-Python
AIOSCompiler	Xu et al. (2024)	Code Generation	NL-CoRE DSL
Autoiot	Shen et al. (2025)	Code Generation	NL-AIoT code
ArabicLLMCompiler	Sibaee et al. (2024)	Code Generation	Arabic-Python
Unicoder	Sun et al. (2024)	Code Generation	Pseudo Code-Python,JS
transformer-x86	Armengol-Estape and O'Boyle (2021)	Compilation	C-x86
transformer-llvm	Guo and Moses (2022)	Compilation	C-LLVM IR
llm-x86	Zhang et al. (2024)	Compilation	C-x86
LEGO-Compiler	Zhang et al. (2025)	Compilation	C-x86,arm,riscv
VirtualCompiler	Gao et al. (2024)	Compilation	C-x86
VeriLOCC	Jin et al. (2025)	Compilation	MIR-SASS,RDNA
InstCombiner	Mannarswamy and Das (2022)	Compilation	arm
VEGA	Zhong et al. (2025)	Compilation	LLVM C++,TableGen
ComBack	Zhong et al. (2024)	Compilation	LLVM C++, TableGen
AutoChip	Thakur et al. (2023)	RTL Generation	Verilog
Origen	Cui et al. (2024)	RTL Generation	Verilog
VeriGen	Thakur et al. (2024)	RTL Generation	Verilog
VerilogLLM	Wang et al. (2025)	RTL Generation	Verilog
MakeMoveCount	DeLorenzo et al. (2024)	RTL Generation	Verilog
RTLLM	Lu et al. (2024)	RTL Generation	Verilog
BuiltRome	Nakkab et al. (2024)	RTL Generation	Verilog
HLSPilot	Xiong et al. (2024)	HLS	C++-HLS C++
Slade	Armengol-Estape et al. (2024)	Decompilation	x86,arm-C
LLM4Decompile	Tan et al. (2024)	Decompilation	(obfuscated) x86-C
Degpt	Hu et al. (2024)	Decompilation	x86-C
Lmpa	Xu et al. (2023)	Decompilation	x86-C
BoostDecompile	Cao et al. (2022)	Decompilation	x86-C
BinSum	Jin et al. (2023)	Decompilation	x86-NL
DecLLM	Wong et al. (2025)	Decompilation	x86-C
IR-LLM	Jiang et al. (2025)	IR Decompilation	LLVM IR-C
Forklift	Armengol-Estape et al. (2024)	Decompilation to IR	x86,arm,riscv-LLVM IR
NeuralShapeCompiler	Luo et al. (2023)	Multimodal	Text-PointCloud-Code
RASP	Weiss et al. (2021)	NeuralCompilation	RASP DSL
Tracr	Lindner et al. (2023)	NeuralCompilation	RASP-transformer weight
ALTA	Shaw et al. (2025)	NeuralCompilation	ALTA-transformer weight
TransformerProgram	Friedman et al. (2023)	NeuralDecompilation	transformer weight-RASP
AlgorithmicLM	Saldyt and Kambhampati (2025)	NeuralDecompilation	transformer weight-python

Table 3: Summary of LLM for inter-level code transformation

proposes using an LLM model to compile any source code of any language to assembly code by matching assembly results.

Llm-x86 (Zhang et al. 2024) uses a data-centric augmentation pipeline to improve compiler-generated corpora quality, with specific focus on numerical representation and label generation, using the improved corpora to finetune an LLM, achieve substantial C-x86 compilation improvement, over 91%.

LEGO-Compiler (Zhang et al. 2025) studies the scalability problem of neural compilation, decomposing function-level code into finer granularities like basicblock-level or statement-level with managed context, and compiles each fragment accordingly with necessary symbol information. Together with a stepwise workflow that can verify intermediate results and an auto repair loop, it achieves over 99% neural compilation accuracy across three architectures on O0 setting and achieve an order of magnitude scalability improvement without model advancement. However, LLM-based compilation is still preliminary, because it fails to outperform existing compiler systems.

Besides full compilation process performed by LLM, there are also works on specific step within the compilation pipeline. VeriLOCC (Jin et al. 2025) studies the register allocation capabilities using LLMs on both CUDA SASS and AMD RDNA assembly generation. While InstCombiner (Mannarswamy and Das 2022) studies the instruction combination capabilities on ARM64 assembly using LLMs.

Except LLM-as-translator methodology, there are also interesting works investigating to generate specific compiler code. VEGA (Zhong et al. 2025) proposes a LLVM backend generation method using pretrained LM with carefully crafted compiler backend dataset ComBack (Zhong et al. 2024). By treating the LLVM code as features to learn and different backends as training data, the model can generate sketches of over 60% of the LLVM backend functions without human intervention. However, fully automated compiler code generation is still far from reality, as compilers are one of the most complicated softwares to maintain.

Except compilation to assembly code, there are also works focusing on RTL code generation, which can

also be seen as a broader compiler task. Among them, AutoChip (Thakur et al. 2023) proposes a self-reflection loop to fix trivial generation errors. Origen (Cui et al. 2024) proposes a data augmentation pipeline using claude3 distilled data and a similar self-reflection mechanism. VeriGen (Thakur et al. 2024) finetunes a series of verilog generation LLMs using supervised finetuning. VerilogLLM (Wang et al. 2025) further trains a verilog generation LLM with reinforcement learning (RL) with testbench feedback.

MakeMoveCount (DeLorenzo et al. 2024) studies to perform MCTS on RTL code generation. RTLLM (Lu et al. 2024) proposes both benchmark for LLM-based RTL generation and a simple-yet-effective self-planning prompting strategy to boost performance of RTL generation with GPT3.5. BuiltRome (Nakkab et al. 2024) also found that hierarchical prompt structuring can dramatically improve LLM performance on hardware design tasks, enabling successful generation of complex modules that would otherwise be impossible.

HLSPilot (Xiong et al. 2024) instead of focusing on verilog RTL generation, it generates High-Level Synthesis (HLS) code translated from software C++ code, it outperforms manually written FPGA kernels with the integration of profiling tools and DSE tools to enable automatic hardware/software partition and pragma tuning.

4.2.3 Decompilation: ASM-PL

This involves the reverse process of compilation, where an LLM attempts to reconstruct a human-readable and semantically meaningful PL from a low-level representation like a binary or assembly file. This is a notoriously difficult task for which the pattern recognition capabilities of LLMs are a promising research direction.

Slade (Armengol-Estape et al. 2024) and LLM4Decompile (Tan et al. 2024) are both learned language models used for decompilation tasks, each has a centric for either optimized code recovery and obfuscated code recovery.

Degpt (Hu et al. 2024) focuses on using LLMs to interpret and refine decompiler output to improve readability and simplicity, which can assist the reverse engineering process.

Lmpa (Xu et al. 2023) develops a program analysis assisted method for symbol name recovery in decompilation, by querying ChatGPT with managed context, the model can generate 75% of the recovered names considered good by users.

BoostDecompile (Cao et al. 2022) uses a multi-layer decompilation pipeline, where the recovery of program is jointly performed by rules and neural networks.

BinSum (Jin et al. 2023) focues on assembly code understanding capabilities of LLMs, and finds that stripping debug symbols has a significant loss to the decompilation accuracy.

DecLLM (Wong et al. 2025) proposes a recompilable centric decompilation system, with an iterative LLM-based repair loop to improve decompiler outputs, which combines static recompiling and dynamic runtime feedback.

Besides full decompilation process, there are also works focusing on some steps. IR-LLM (Jiang et al. 2025) focuses on the LLVM IR-to-C decompilation process, while Forklift (Armengol-Estape et al. 2024) focuses on the assembly-to-LLVM IR decompilation process.

4.2.4 Special cases of cross-level transformations

Beyond transformations within the traditional compilation stacks, an emerging line of research explores more special cases of compilation which is not limited to typical text-based or symbolic domains.

For example, NeuralShapeCompiler (Luo et al. 2023) proposes a compiler-inspired, unified framework for translating data between multiple modules. This framework first converts all data into a unified, discrete intermediate code and then uses a transformer model to perform the translation across text, code and pointcloud.

A distinct and emerging research direction, also termed with the name of "neural compilation", explores the direct translation of *programs into the parameters of a neural network*. This approach is not aimed at traditional execution but at understanding the algorithmic capabilities of models like transformers, thereby contributing to the study of explainability in LLMs. A foundational line of work in this area began with RASP (Weiss et al. 2021), an abstract programming language designed to express algorithms using primitives that mirror the core operations of a transformer. Building on this theoretical framework, Tracr (Lindner et al. 2023) was developed as a compiler that translates RASP programs directly into the weights of a standard transformer model. This capability was further extended by ALTA (Shaw et al. 2025), which adds support for dynamic control flow operations like loops.

Complementing this compilation process, some research explores the reverse direction—a form of decompilation. For example, TransformerProgram (Friedman et al. 2023) introduces a method to train a modified transformer that can be automatically converted back into a discrete, human-readable Python program. Furthermore, AlgorithmicLM (Saldyt and Kambhampati 2025) investigates methods for composing these "neurally compiled" weights to execute combined algorithms through an augmented interpreter.

Despite its theoretical importance, this area of research currently also faces significant limitations in both the scale of the programs and the variety of operations that can be compiled, marking it as a key area for further investigation.

As we can see in the number of studies about intra-level (Table 2) and inter-level (Table 3) code transformations, there are more studies handling with high-level code (59+34) than low-level IR/assembly

(14+16). The quantitative imbalance can be viewed in two ways. First, Low-level compiler tasks do receive less attention from the community as there are much more LLM researchers than compiler researchers (who are interested in tasks about low-level representations). Second, low-level code representations are less amenable to LLM approaches, because: 1. there are less low-level code corpora than high-level corpora in most LLMs' pretraining stage; 2. the quality of low-level corpora is relatively low, as many low-level code representations are automatically generated through compilers, which lack readability and is hard for LLMs to learn with.

4.3 Non-Transformed Utilities

Table 4 outlines related utility-based studies. Except for generative tasks, LLMs also enable utility-based tasks in broader compiler domain, such as the vulnerability test generation and compiler fuzzing test, which we have outlined earlier.

As for other utility studies, CompileAgent (Hu et al. 2025) proposes to use LLMs to auto-configure project setup. DCC (Taylor et al. 2024) develops a tool that integrates a Large Language Model into the Debugging C Compiler to generate context-aware, novice-focused explanations for compile- and run-time errors to help introductory programming students. Quantum (Ren et al. 2024) employs a Seq2Seq model to solve the qubit routing problem in quantum compilation, reducing the number of added gates and compilation runtime compared to heuristic algorithms. ComPAT (Cai et al. 2024) introduces a LLM-based teaching assistant for compiler principles courses. HPC-GPT (Ding et al. 2023) finetunes a LLM with automatically generated, domain-specific data to improve performance on High-Performance Computing tasks like data race detection and AI model management. Fair (Niu et al. 2024) proposes a pre-trained model for IR that uses a novel flow-type-aware graph input, a Graph Transformer architecture, and five specific pre-training tasks to improve LLM's ability to understand IR.

There is a growing interest in integrating deep learning models into binary analysis, a domain critical for reverse engineering, vulnerability detection, and software supply chain security. This research primarily focuses on two key tasks: binary similarity detection and compiler provenance identification (CPI).

In binary similarity detection, the goal is to determine if two binary functions are semantically equivalent despite variations introduced by different compilers or optimization settings. Foundational work like OSCAR (Peng et al. 2021) established a pre-training paradigm to learn code representations from LLVM IR, using contrastive learning to handle diverse optimizations. Subsequent approaches have refined this concept. JTrans (Wang et al. 2022), for example, introduces a jump-aware Transformer model to better capture control flow, while DiEmph (Xu et al. 2023) improves robustness by de-emphasizing compiler-induced noise in the binary code. Similarly, OPTango (Geng et al. 2023) utilizes multi-central representation learning to create a binary diffing tool resilient to the complex effects of compiler optimizations.

A specialized application of this is Compiler Provenance Identification (CPI), which aims to identify the compiler and optimization level used to generate a given binary. Here, researchers have explored various neural architectures. For instance, MuCPI (Gao et al. 2024) enhances a Gated Graph Neural Network (GGNN) with attention mechanisms to learn features from a binary's control flow graph. In a more unconventional approach, ObfuscateCPI (Khan et al. 2024) demonstrates a resilient method by converting binaries into images and applying pre-trained computer vision models to identify the compiler's visual fingerprint, even through obfuscation.

In the end of this section, we could finally answer **RQ2** based on recent LLM compiler studies.

> **RQ2**: What are the primary compiler-related tasks addressed by LLMs?
> **Answer**: **Generative tasks**, such as code generation, code transpilation, code repair, code optimization and decompilation are now the primary tasks addressed by LLM-enabled compilers, which have made significant advancements with more powerful LLMs and more complete system design, and are now surpassing traditional methods in one or more dimensions. **Utility tasks** such as system autoconfiguration, similarity detection and compiler fuzzing are also significantly improved with LLMs.

However, on the narrowly defined compilation tasks, although there are studies on the end-to-end compilation process (Armengol-Estape and O'Boyle 2021; Zhang et al. 2024; Gao et al. 2024; Zhang et al. 2025), part of the compilation process (Jin et al. 2025; Mannarswamy and Das 2022), and the IR optimization process (Deng et al. 2024; Grubisic et al. 2024; Cui et al. 2025; Cummins et al. 2025), they are more or less preliminary and mostly cannot surpass traditional compilers in either performance, cost or scalability. Some works (Zhong et al. 2024, 2025) try to address the compiler construction problem in an "agile-development" view, they still require significant compiler experts intervention and cannot generalize to arbitrary compiler development tasks due to the lack of dataset.

Nevertheless, with increasing number of studies on this field year after year, and the advancements made. We believe LLM-powered compiler system is a promising research direction.

5 Benchmarks & State-of-the-Art Evolution

5.1 Benchmark, Metrics & Scale

A critical aspect of evaluating this field is to systematically analyze the benchmarks themselves. Table 5 provides a broad summary of the benchmarks and

Acronym	Citation	Tasks	Code level
LIBRO	Kang et al. (2023)	CVE test Generation	Defects4J
SecurityTestGen	Zhang et al. (2023)	CVE test Generation	Java CVE test
GeneticImprove	Brownlee et al. (2023)	Compiler Fuzzing	Java
BenchDirect	Tsimpourlas et al. (2023)	Compiler Fuzzing	OpenCL
WhiteFox	Yang et al. (2024)	Compiler Fuzzing	PT-Inductor/XLA/TF-Lite
MetaMut	Ou et al. (2024)	Compiler Fuzzing	C/C++
ClozeMaster	Gao et al. (2025)	Compiler Fuzzing	Rust
FMCSO	Italiano and Cummins (2025)	Compiler Fuzzing	C/C++
CompileAgent	Hu et al. (2025)	Auto Configuration	Cmake/Make
DCC	Taylor et al. (2024)	Error Explanation	Compiler Log
Quantum	Ren et al. (2024)	Qubit Routing	Quantum Program
ComPAT	Cai et al. (2024)	Course Assistant	Compiler Textbook
OSCAR	Peng et al. (2021)	Binary Detection	LLVM IR
JTrans	Wang et al. (2022)	Binary Detection	Assembly
DiEmph	Xu et al. (2023)	Binary Detection	Assembly
OPTango	Geng et al. (2023)	Binary Detection	Assembly
MulCPI	Gao et al. (2024)	Binary Detection	Assembly
ObfuscateCPI	Khan et al. (2024)	Binary Detection	Assembly
HPC-GPT	Ding et al. (2023)	HPCknowledge Pretrain	HPC knowledge
Fair	Niu et al. (2024)	IR Understanding	LLVM IR

Table 4: Summary of LLM for compiler utilities

Benchmark/Dataset	Languages	Primary Task	Size (Units)
CodeNet (Puri et al. 2021)	Multi-lingual (55)	Code Understanding, Transpilation	13.9M
AVATAR (Ahmad et al. 2023)	Java, C++	Code Transpilation	∼9.5k
TransCoder-Test (Roziere et al. 2020)	C++, Java, Py	Code Transpilation	1.4k
PIE (Madaan et al. 2023)	Python, Java	Code Transpilation, Edit	∼77k
MiBench (Guthaus et al. 2001)	C	Embedded Compilation, Optimization	35
KernelBench (Ouyang et al. 2025)	CUDA, PyTorch	GPU Kernel Generation / Optimization	250
TritonBench (Li et al. 2025)	Triton DSL	GPU Kernel Generation / Optimization	184
AnghaBench (Da Silva et al. 2021)	C, ASM	Neural Compilation	∼1M
ExeBench (Armengol-Estapé et al. 2022)	C, ASM	Neural Compilation	0.7M
HumanEval (Chen et al. 2021a)	Python	Code Generation	164
MBPP (Austin et al. 2021)	Python	Code Generation	974
Defects4J (Just et al. 2014)	Java	Code Repair	854

Table 5: Summary of commonly used benchmarks/datasets for LLM-Compiler tasks. The "Size (Units)" column is simplified to show the approximate number of core entities to refelect the evaluation scale, e.g., (M)illions or (k)thousands of the major problems, functions, or code samples.

datasets used across various LLM-Compiler tasks. As shown, we can analyze these benchmarks along three recurring dimensions:

Benchmark construction generally follows three patterns: (1) *adapting large public benchmarks/datasets* (e.g., CodeNet, AVATAR), often with new filters or metadata. We summarize these commonly used benchmarks/datasets in Table 5.

For general understanding and transpilation, **CodeNet** (Puri et al. 2021) provides a large-scale dataset of nearly 14 million code samples spanning more than 55 languages and accompanied by rich metadata. **AVATAR** (Ahmad et al. 2023) builds on CodeNet by curating 9,515 problems and deriving 3,391 parallel function pairs for fine-grained alignment. **TransCoder-Test** offers a standardized parallel set used since Roziere et al. (2020) (948 test instances) and remains a staple for translation accuracy. **PIE (Performance-Improving Edits)** (Madaan et al. 2023) is a performance-optimization benchmark built from roughly 77k submission pairs (mostly sourced from CodeNet) and evaluates runtime improvements under the gem5 simulator, making it a widely adopted testbed for optimization-oriented LLMs.

For compilation and generation tasks, **MiBench** (Guthaus et al. 2001) contributes 35 domain-realistic embedded C programs used to stress end-to-end compilation. **KernelBench** (Ouyang et al.

2025) focuses on GPU kernel generation, packaging 250 tasks with correctness and speedup metrics. **TritonBench** (Li et al. 2025) similarly focuses on Triton-based GPU kernel evaluation. For neural compilation, **AnghaBench** (Da Silva et al. 2021) contains 1M compilable C functions, while **ExeBench** (Armengol-Estapé et al. 2022) scales this to 680K executable C functions with a 40K unittest-based verifiable split. For code generation, **HumanEval** (Chen et al. 2021a) (164 Python problems) is the de facto standard for *pass@k* evaluation, and **MBPP** (Austin et al. 2021) (974 Python tasks) targets everyday programming competence.

The second pattern, (2) *curating real open-source repositories*, aims to reflect end-to-end realism. For example, **AlphaTrans** (Ibrahimzada et al. 2025) decomposes ten real-world repositories to assess runtime behavior at repository scale. Similarly, **Oxidizer** (Zhang et al. 2025) curates several real-world Go projects to evaluate Go-to-Rust transpilation. The well-known **Defects4J** (Just et al. 2014) benchmark, a curated collection of reproducible bugs from real Java projects, also follows this pattern and serves as a foundational benchmark for many program repair studies (Xia et al. 2023; Wei et al. 2023).

The third pattern, (3) *synthesizing test cases*, probes capabilities on tailored downstream tasks. For example, **QiMeng-Xpiler** (Dong et al. 2025) assembles a kernel-level tensor-program suite to test cross-DSL transcompilation. This pattern is also the foundation of **compiler fuzzing**, where works like **WhiteFox** (Yang et al. 2024) and **MetaMut** (Ou et al. 2024) synthesize diverse and non-trivial test programs specifically designed to find compiler bugs.

At the level of **granularity**, most benchmarks remain function-centric, which makes side-by-side comparison tractable. For example, among the 21 transpile-related tasks listed in Table 2, only a small but important slice moves to whole-program/repo contexts, e.g., in **Oxidizer** (Zhang et al. 2025) and **AlphaTrans** (Ibrahimzada et al. 2025), success is defined end-to-end: the code must compile, execute, and exhibit correct behavior, reflecting real deployment conditions rather than snippet-level fidelity.

On **metrics**, evaluation coalesces around two families of metrics: *text-based* similarity and *semantic-based* correctness across these benchmarks. Text-based metrics such as BLEU (Papineni et al. 2002) measure n-gram overlap with references and are inexpensive and reproducible, but they neither account for code syntax/-grammar nor data- and control-flow, which limits their faithfulness for programs; CodeBLEU (Ren et al. 2020) explicitly amends this by combining standard n-gram overlap with keyword-weighted n-grams, AST (syntax) matching, and data-flow (semantics) matching to better correlate with expert judgments on code tasks.

Semantics-centric metrics execute or analyze the artifact: pass@k estimates the probability that at least one of k generated candidates passes all unit tests (now standard for HumanEval-style synthesis); Computational Accuracy (a.k.a. functional-equivalence accuracy) runs reference tests on translated code and reports the fraction that compile and produce correct outputs; and optimization-focused works report SpeedUp and Percent Optimized (share of problems where the new code exceeds a fixed improvement threshold, e.g., 10% faster). Beyond these shared metrics, task-specific measures capture domain goals: **SALLM** (Siddiq et al. 2024) augments pass@k with secure@k and vulnerable@k to quantify the security of generated samples. **CompilerDream** (Deng et al. 2024) evaluates end-to-end code-optimization by reporting code size reduction (e.g., IR instruction count) relative to compiler baselines such as LLVM -Oz, aligning the metric with embedded deployment objectives. **CoTran** (Jana et al. 2024) augments correctness with error-position statistics.

5.2 Evolution

While §§ 5.1 provided a broad overview of the evaluation landscape, this section provides a deeper, chronological analysis of SOTA evolution for four representative tasks. This allows us to track concrete technical progress made in recent years.

We have selected four tasks to illustrate this evolution: **Code Transpilation**, **Neural Compilation**, **GPU Kernel Generation**, and **LLVM IR Optimization**.

We will analyze the key benchmarks and metrics for each, tracking the chronological progress of state-of-the-art results, which are visually summarized in Figure 6. It is crucial to add a caveat: **direct comparisons are not always perfectly fair**. Different studies often use non-uniform experimental setups, with different dataset splits, baseline compiler versions, or testing conditions. Nonetheless, this analysis clearly demonstrates the rapid and significant technical progress in the field.

Code Transpilation: The TransCoder-series of works established a foundational benchmark for source-to-source translation, focusing on programming language transpilations, e.g, Python, Java, and C++. While early metrics included BLEU and Exact Match (EM), Computational Accuracy (CA@1)—whether the translated code passes a set of unit tests—emerged as the most meaningful and enduring metric. As illustrated in Figure 6(a), the original **TransCoder** (Roziere et al. 2020) achieved CA@1 scores such as 29.9% (Py-Java) and 70.6% (Java-C++). Subsequent works, like **DOBF** (anne Lachaux et al. 2021) which focused on deobfuscation, improved on these (e.g., 44.5% on Py-Java). A significant leap came with **TransCoder-ST** (Roziere et al. 2022), which introduced test-based filtering and improved data curation to boost CA@1 to 62.3% (Py-Java) and 75.4% (Java-C++). **TransCoder-IR** (Szafraniec et al. 2023) explored a novel compiler-IR-based approach, achieving a comparable 68.7% (Java-C++), demonstrating robustness across more compiler backends.

Neural Compilation The goal of end-to-end neural compilation (e.g., C-to-x86) is extremely challenging. Progress here is clearly marked by a shift from superficial metrics to rigorous, execution-based

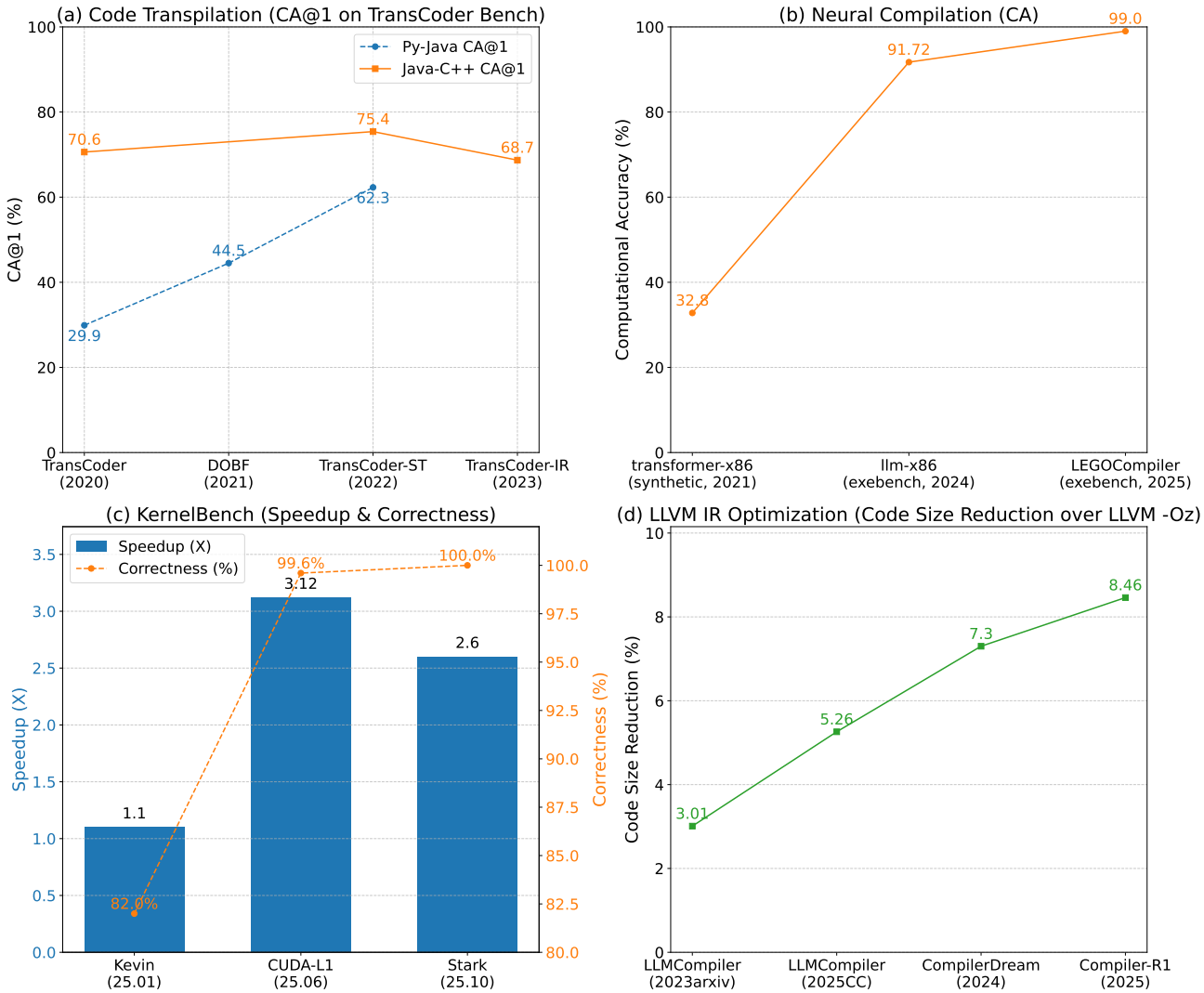

Fig. 6: Visualization of State-of-the-Art (SOTA) progress across four key LLM-Compiler tasks: **(a)** Code Transpilation (CA@1 on TransCoder), **(b)** Neural Compilation (CA on ExeBench), **(c)** GPU Kernel Generation (Speedup & Correctness on KernelBench), and **(d)** LLVM IR Optimization (Code Size Reduction).

benchmarks, as depicted in Figure 6(b). Initial work by Armengol-Estape and O'Boyle (2021) trains a standard transformer on AnghaBench for testing the neural compilation capability, reporting high BLEU (90.2) and Syntax (98.5) scores. However, these metrics failed to capture semantic correctness; on a small 64-item synthetic benchmark, which has similar benchmark difficulty compared to ExeBench (not published yet), they achieved only 32.8% CA. A major advancement came from Zhang et al. (2024), who fine-tuned a large language model CodeLlama-13B with extensive data augmentation. Evaluated on the rigorous ExeBench (developed from AnghaBench, with over 17k verifiable testcases), they achieved a remarkable 91.72% CA. Most recently, LEGO-Compiler (Zhang et al. 2025) employed a training-free, divide-and-conquer methodology, with simplified step-by-step translation and basicblock-level code complexity reduction, it further pushes the CA on ExeBench to over 99% across multiple models.

GPU Kernel Generation For GPU kernel generation, the evaluation standard shifts from mere semantic correctness to achieving **performance speedup** over strong baselines like `torch eager`. The primary benchmark in this area is **KernelBench** (Ouyang et al. 2025), which is used by several key studies, as visualized in Figure 6(c). However, direct comparison remains challenging because **different works may evaluate on different GPU hardware** (e.g., A100 vs. L40), which can significantly affect speedup results. Therefore, the following progression should be viewed as a demonstration of the trend rather than a direct, fair comparison. Early work (**Kevin** (Baronio et al. 2025)) highlighted the correctness challenge, achieving only **82%** correctness with a minimal **1.10x** average speedup over `torch eager`. A significant advancement came with **Stark** (Dong et al. 2025), which first achieved **100% correctness** across all tasks, and delivered substantial speedups ranging from **1.58x** to **3.03x** on different benchmark levels, with an average **2.60x**. Another well-known work **CUDA-L1** (Li et al. 2025) employed both SFT and reinforcement learning (GRPO), achieving near 100% correctness (249/250) while pushing the SOTA average speedup to **3.12x** over

`torch eager`, a result that is further supported by its open-sourced evaluation results across multiple GPUs.

LLVM IR Optimization For LLVM IR optimization, progress is often tracked by code size reduction relative to the compiler's most aggressive size-optimization flag (e.g., -Oz). As shown in Figure 6(d), this area highlights the challenge of non-standardized benchmarks. Different works use different datasets and baseline compiler versions, making direct comparisons difficult. Nonetheless, a clear trend of improvement is visible. The initial Meta-LLMCompiler work (Cummins et al. 2023) achieved a 3.01% reduction over LLVM -Oz via large-scale pre-training. The subsequent Meta-LLMCompiler (Cummins et al. 2025) improved this to 5.26% by adding multi-step post-training to adapt the model for subtasks like compiler behavior emulation and decompilation. Similarly, CompilerDream (Deng et al. 2024) build their own compiler world model with reward smoothing technique, achieves 7.3% over LLVM -Oz with guided search, which leads the CompilerGym (Cummins et al. 2022) leaderboard. A different, agentic approach with LLM-as-Selector philosophy, Compiler-R1 (Pan et al. 2025), used GRPO (RL) for compiler flag tuning and reported an 8.46% reduction over LLVM -Oz.

In summary, this section has critically examined the evaluation methodologies and benchmark-driven progress at the **intersection of LLMs and compilers**. Our analysis, spanning both a broad survey of the landscape (§§ 5.1) and a deep dive into SOTA evolution (§§ 5.2), reveals two key insights. **First**, there is clear and rapid technical progress across diverse tasks, from achieving over 99% correctness in neural compilation to delivering significant speedups in GPU kernel generation. **Second**, this progress is mirrored by a maturation in evaluation, with the community decisively shifting away from superficial text-based metrics (like BLEU) toward rigorous, execution-based metrics (like Computational Accuracy and performance speedup). However, this analysis also highlights significant remaining challenges. As seen in the GPU and IR optimization tasks, there is a **lack of standardized comparison protocols** (e.g., fixed hardware, baselines, and datasets), which hinders fair, direct evaluation. Furthermore, as our analysis in §§ 5.1 and Table 5 indicates, the vast majority of benchmarks remain **function-level**. Large-scale, **project-level benchmarks** with unified standards are still rare, making it difficult to properly evaluate scalability—a critical factor for real-world compiler systems.

6 Discussions

Having categorized the existing body of work according to design philosophy (§ 3) and level of code abstraction (§ 4), we now turn to a broader discussion of the field. This chapter synthesizes our findings to address the final two research questions of this survey. We begin by summarizing the primary advancements offered by LLM-based approaches (**RQ3**). We then delve into the common challenges and corresponding future opportunities that define the research frontier (**RQ4**). Finally, we explore several additional topics that are critical for the healthy evolution of this domain.

6.1 Primary Advancements of LLM-based Approaches (RQ3)

The integration of LLMs into the compilation process has catalyzed several fundamental advancements, moving beyond the capabilities of both traditional handcrafted compilers and earlier machine learning techniques. These advancements primarily stem from the models' ability to learn deep semantic and structural patterns directly from vast corpora of source code.

- **Democratizing Compiler Code Development and Optimization**: LLMs significantly lower the barrier to entry for creating sophisticated code transformation and optimization tools. Instead of requiring years of specialized expertise to design and implement complex compiler/-transpiler heuristics, developers can now achieve impressive results by fine-tuning pre-trained models or applying large foundation models. This capability stems from the models being pre-trained on vast code corpora, much more than any code expert can read in its entire life, allowing them to internalize a wide array of programming patterns and techniques far beyond the scope of any single human expert in its weights. Consequently, this accelerates the development of new optimizers and makes bespoke, high-performance compilation accessible to a wider audience.
- **Discovering Novel Optimization Strategies**: For decades, code optimization has been guided by human-designed heuristics. LLMs, with their ability to learn from enormous datasets of real-world code, can identify complex patterns and discover novel optimization strategies that may be non-obvious to human experts. By exploring vast optimization spaces and generalizing from successful examples seen during pre-training, LLMs have the potential to surpass the performance ceilings of existing heuristic-based systems.
- **Broadening the Scope and Utility of "Compilation"**: The application of LLMs has expanded the traditional definition of a compiler's role. Tasks such as large-scale Code Transpilation and Automated Program Repair are now treated as viable compilation problems. This broadened utility positions the compiler not just as a static tool for translation and optimization, but as a versatile platform for ongoing code maintenance, migration, and modernization, providing enormous value across the entire software lifecycle.

RQ3: What are the primary advancements offered by LLM-based approaches?
Answer: The primary advancements are threefold: (1) They **democratize** compiler code development by lowering the required expertise and accelerating implementation; (2) they can discover **novel optimization strategies** beyond human-designed heuristics by learning from vast codebases; and (3) they **broaden the utility** of compilers, turning them into versatile tools for tasks like code transpilation and repair.

6.2 Common Challenges and Future Opportunities (RQ4)

Despite the rapid progress, the field faces significant challenges that must be addressed to move from academic research to production-grade, reliable tools. These challenges, in turn, highlight promising avenues for future research.

6.2.1 Common Challenges and Task-Specific Approaches

Despite the rapid progress, the field faces significant challenges to bridge the gap to production-grade, reliable tools. Addressing them requires moving beyond isolated, function-level studies. However, as we will discuss, the proposed solutions are **not universal paradigms** but rather **highly task-specific approaches**. The fundamental gap between current LLM capabilities and the robustness of traditional compilers remains significant.

Ensuring Correctness and Verifiability This remains the most critical challenge. Traditional compilers must guarantee semantic equivalence, but the probabilistic nature of LLMs can introduce subtle bugs. Approaches to mitigate this are emerging.

One strategy is **constraining the generation process** itself (Mündler et al. 2025). Methods like grammar-guided constrained decoding (Willard and Louf 2023; Dong et al. 2024) can force the LLM to produce outputs that adhere to the given grammar constraints, which are therefore *syntactically* valid, guaranteeing compilability. However, this only addresses syntax, not semantic correctness, which remains the harder challenge.

A more dominant strategy is the **"Translator + Verifier" hybrid pattern**, where the LLM's generative output is checked by a deterministic component after generation. This pattern primarily manifests in two forms. The most common form is **dynamic functional validation** via test suites. This is often placed within an iterative feedback loop (Ye et al. 2024; Wong et al. 2025), which serves a similar purpose to constrained decoding by iteratively correcting errors, albeit at a higher cost.

The second, more rigorous form employs **formal verification** in a task-specific manner: **LLMLift** (Bhatia et al. 2024) uses Floyd-Hoare Logic, **Qimeng-Xpiler** (Dong et al. 2025) uses SMT-solvers for key transformations, and **LLMVectorizer** (Taneja et al. 2025) formally verifies generated SIMD intrinsics using the Alive2 validator. While combining these strategies is promising, applying rigorous formal verification at scale for all transformations remains a costly and open problem.

Scalability to Large, Real-World Codebases This is a critical barrier where the very definition of "scalability" is task-dependent. For tasks like *GPU kernel optimization*, e.g., CUDA-L1 (Li et al. 2025) and KernelBench (Ouyang et al. 2025), the scope is naturally constrained to a single function/kernel, making repository-level scalability a non-issue.

However, for *legacy code transpilation*, the challenge becomes managing inter-procedural context across the codebase. Here, divide-and-conquer strategies are employed, such as project partitioning in **AlphaTrans** (Ibrahimzada et al. 2025) or type-checking feature mapping in **Oxidizer** (Zhang et al. 2025). For project-level *repair*, **CoCoGen** (Bi et al. 2024) relies on RAG-based context retrieval.

A more fundamental challenge lies in a **gap in design assumptions**. LLMs are pre-trained on vast public codebases that, for the most part, adhere to "Clean Code" principles or the Single Responsibility Principle (SRP). This biases their training data toward short, focused functions. **Real compilers, however, cannot make this assumption**; they must robustly handle any syntactically valid code, including massive, monolithic functions ("God functions") that violate these human-centric principles. This creates a critical gap for any LLM aiming to replace or complement core compiler components.

LEGO-Compiler (Zhang et al. 2025) is a notable work that addresses this *compiler-centric* problem by decomposing functions into finer-grained basic blocks or statements. This semantic-preserving decomposition, as a further supplement to the task-dependent nature of scalability, is itself only effective for non-optimization scenarios or translations with only local, intra-unit optimizations; how to scale this decomposition strategy to global optimization scenarios remains unclear and challenging.

Ultimately, even with these decomposition strategies, the finite context window of LLMs remains the hard bottleneck. Real-world codebases like the LLVM or Linux Kernel projects, with their sheer size and complexity, are orders of magnitude beyond the scalability of current LLM-based approaches.

Interpretability and Debuggability The "black box" nature of LLMs makes their decisions difficult to trust. This challenge is less mature, but specific strategies offer paths forward. One approach is to force the model to "show its work" using Chain-of-Thought (CoT) prompting, as seen in **CodeOptCoT** (Xu et al. 2024) and **SBLLM** (Gao et al. 2025).

Another strategy focuses on debugging and explanation: **CompilerGPT** (Pirkelbauer and Liao 2025) analyzes compiler reports, while **DCC** (Taylor et al. 2024) generates novice-friendly explanations for errors. Perhaps the most robust solution is the **LLM-as-Generator** philosophy (§§ 3.3), exemplified by **CodeTransform** (Cummins et al. 2024), where the generated artifact (a transformation script) is itself human-readable and debuggable.

Performance and Cost-Effectiveness The inference cost of large models can be substantial. For an LLM-based optimizer to be practical, the performance gains it provides must outweigh the computational cost and latency of its own execution. Striking the right balance between model size, inference speed, and optimization quality is an ongoing challenge.

Strategies to manage this cost-benefit trade-off are emerging. One strategy focuses on *reducing the LLM's own cost* by using smaller, specialized models; **Perfcodegen** (Peng et al. 2025) and **PerfRL** (Duan et al. 2025) both show that small models trained with execution feedback can achieve strong performance, avoiding the expense of large-scale models.

A different strategy seeks to *offset the LLM's cost* by using it as an accelerator for existing, expensive processes. In auto-tuning, for example, **Reasoning-Compiler** (Tang et al. 2025) and **TLM** (Zhai et al. 2024) use an LLM to intelligently guide a search. In this context, the LLM's inference cost is negligible compared to the hours of compilation and benchmarking time it saves.

6.2.2 Future Opportunities

- **Hybrid Compiler Systems**: The most promising near-term future lies not in replacing traditional compilers, but in augmenting them. Hybrid systems that combine the creative pattern-matching of LLMs with the rigor and speed of formal, deterministic compiler algorithms could achieve the best of both worlds. This can be realized through the **Selector** or **Generator** philosophies, but it also applies powerfully to the **Translator** model. For instance, a system could delegate the bulk of code compilation to a fast and reliable traditional compiler, while invoking an LLM to handle specific, challenging portions that the compiler cannot. This allows the system to support tasks it was not originally designed for, such as compiling projects with mixed-language codebases or extending support to new and emerging hardware architectures for which a mature backend does not yet exist.
- **Self-Improving and "Learning" Compilers**: A significant opportunity lies in creating compilers that learn and evolve over time by transforming the creative, non-deterministic discoveries of LLMs into permanent, deterministic compiler capabilities. This could be realized through a multi-stage process:
  - First, an **LLM as a Translator** could be used in an exploratory capacity to discover novel, ad-hoc optimizations for specific code snippets that traditional heuristics miss.
  - Next, these successful and verified transformations would be collected into a specialized dataset of high-quality optimization examples.
  - Finally, this dataset would be used to task an **LLM as a Generator** with a more ambitious goal: not just to perform another one-off translation, but to write the source code for a new, deterministic compiler pass or component that systematically implements the discovered optimization strategy.

  This newly generated component can then be validated and integrated into the traditional compiler framework. The result is a compiler that has permanently "learned" a new skill, effectively creating a powerful paradigm for compiler evolution.
- **A New Generation of Interactive Developer Tools**: LLMs can transform how developers interact with compilers. We can imagine future IDEs where an LLM-powered compiler agent not only optimizes code but also explains performance bottlenecks in natural language, suggests complex refactorings, and interactively works with the developer to improve their code.

Based on the detailed discussion of the key obstacles and the corresponding research avenues, we can now synthesize these findings to provide a concise answer to our fourth research question (**RQ4**).

> **RQ4**: What are the common challenges and future opportunities in this emerging field?
> **Answer**:
> **Challenges**: Correctness & Verifiability, Scalability, Interpretability, and Performance Cost.
> **Opportunities**: Hybrid Systems, Self-Improving Compilers, and Interactive Developer Tools.

6.2.3 Further Discussion Points

- **The Need for Standardized Benchmarks:** The field's progress is hampered by the lack of benchmarks designed for the unique challenges of LLM-based compilers. While several benchmarks have emerged for specific downstream tasks, such as ExeBench (Armengol-Estapé et al. 2022), TritonBench (Li et al. 2025), and VerilogEval (Liu et al. 2023), they often fall short in adequately covering the dimensions of complexity and, most critically, scalability. This gap is a significant obstacle for evaluating LLM-based translators and optimizers on realistic, large-scale applications. Conversely, while traditional suites like SPEC (Henning 2006) possess the required scale and complexity, they are not designed for LLM-based workflows and their end-to-end difficulty can be prohibitive for current models. This suggests a crucial need

for a new class of "LLM-friendly" benchmarks designed for hybrid evaluation, where external processes handle boilerplate code, allowing the benchmark to focus specifically on evaluating the LLM's core capability in translating or optimizing the most critical sections of a program.

- **Synergy Between PL/Compiler and ML/LLM Communities:** Meaningful progress requires deep, symbiotic collaboration. The ML community can build more powerful and code-aware models, but the Programming Language (PL) and compiler community is essential for defining the right problems, providing domain-specific knowledge (e.g., program semantics, IR structures), curating high-quality datasets, and developing the rigorous verification techniques necessary to ensure the correctness of the final output.
- **The Evolving Role of the Compiler Engineer:** The rise of LLMs is poised to shift the role of the compiler engineer. The focus may move from manually writing complex, handcrafted heuristic algorithms to a new set of responsibilities. These could include curating massive code datasets for model training, designing effective prompting strategies, developing robust verification systems for LLM outputs, and analyzing the novel optimizations discovered by these models to gain new insights into program performance.

7 Conclusion

In this survey, we presented a systematic overview of the emerging application of Large Language Models to the field of compilation, a domain traditionally governed by handcrafted heuristics. We introduced a multi-dimensional taxonomy to structure this diverse landscape, categorizing existing works by their Design Philosophy, LLM Methodology, Level of Code Abstraction, and specific Task Type. Our analysis highlights that LLMs are making significant advancements by democratizing compiler development, discovering novel optimization strategies, and broadening the compiler's utility to include complex tasks like code transpilation and repair. Despite this progress, the field also faces critical challenges in ensuring the correctness and verifiability of generated code, achieving scalability for large-scale software, and improving model interpretability. By systematically categorizing the state-of-the-art and synthesizing its primary advancements and challenges, this survey serves as a foundational roadmap for researchers and practitioners navigating this exciting and transformative field.

Declarations

Conflict of interest On behalf of all authors, the corresponding author states that there is no Conflict of interest.

Funding This work was partially supported by National R&D Program of China (2024YFB4505603), the Jiangsu Province Key R&D Program (Grant No. BG2024028) and National Natural Science Foundation of China (U23B2020, 62302479, 62232015).

References

Armengol-Estape, J., O'Boyle, M.: Learning c to x86 translation: An experiment in neural compilation. In: Advances in Programming Languages and Neurosymbolic Systems Workshop (2021)

Armengol-Estape, J., Rocha, R.C.O., Woodruff, J., Minervini, P., O'Boyle, M.: Forklift: An extensible neural lifter. In: First Conference on Language Modeling (2024). https://openreview.net/forum?id=LWfDcI6txJ

Armengol-Estapé, J., Woodruff, J., Brauckmann, A., Magalhães, J.W.d.S., O'Boyle, M.F.P.: Exebench: an ml-scale dataset of executable c functions. In: Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming. MAPS 2022, pp. 50–59. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3520312.3534867 . https://doi.org/10.1145/3520312.3534867

Armengol-Estape, J., Woodruff, J., Cummins, C., O'Boyle, M.F.: Slade: A portable small language model decompiler for optimized assembly. In: 2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 67–80 (2024). IEEE

Albuquerque, L., Gheyi, R., Ribeiro, M.: Evaluating the capability of llms in identifying compilation errors in configurable systems. arXiv preprint arXiv:2407.19087 (2024)

Ahmad, I., Luo, L.: Unsupervised binary code translation with application to code clone detection and vulnerability discovery. In: Bouamor, H., Pino, J., Bali, K. (eds.) Findings of the Association for Computational Linguistics: EMNLP 2023, pp. 14581–14592. Association for Computational Linguistics, Singapore (2023). https://doi.org/10.18653/v1/2023.findings-emnlp.971 . https://aclanthology.org/2023.findings-emnlp.971/

Lachaux, M.-a., Roziere, B., Szafraniec, M., Lample, G.: DOBF: A deobfuscation pre-training objective for programming languages. In: Beygelzimer, A., Dauphin, Y., Liang, P., Vaughan, J.W. (eds.) Advances in Neural Information Processing Systems (2021). https://openreview.net/forum?id=3ez9BSHTNT

Anthropic: The claude 3 model family: Opus, sonnet, haiku. Technical report, Anthropic (March 2024). Available at https://www.anthropic.com/news/claude-3-family

anthropics: claude-code: A command line interface for Anthropic's Claude AI. GitHub. Accessed: August 11, 2025 (2025)

Anysphere, Inc.: Cursor: The AI-first Code Editor. https://cursor.sh/. Accessed: August 11, 2025 (2023)

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., Sutton, C.: Program Synthesis with Large Language Models (2021). https://arxiv.org/abs/2108.07732

Ahmad, W.U., Tushar, M.G.R., Chakraborty, S., Chang, K.-W.: AVATAR: A parallel corpus for Java-python program translation. In: Rogers, A., Boyd-Graber, J., Okazaki, N. (eds.) Findings of the Association for Computational Linguistics: ACL 2023, pp. 2268–2281. Association for Computational Linguistics, Toronto, Canada (2023). https://doi.org/10.18653/v1/2023.findings-acl.143 . https://aclanthology.org/2023.findings-acl.143/

Brownlee, A.E., Callan, J., Even-Mendoza, K., Geiger, A., Hanna, C., Petke, J., Sarro, F., Sobania, D.: Enhancing genetic improvement mutations using large language models. In: International Symposium on Search Based Software Engineering, pp. 153–159 (2023). Springer

Berabi, B., He, J., Raychev, V., Vechev, M.: Tfix: Learning to fix coding errors with a text-to-text transformer. In: International Conference on Machine Learning, pp. 780–791 (2021). PMLR

Baronio, C., Marsella, P., Pan, B., Guo, S., Alberti, S.: Kevin: Multi-Turn RL for Generating CUDA Kernels (2025). https://arxiv.org/abs/2507.11948

Bhatia, S., Qiu, J., Hasabnis, N., Seshia, S.A., Cheung, A.: Verified code transpilation with llms. Advances in Neural Information Processing Systems **37**, 41394–41424 (2024)

Bi, Z., Wan, Y., Wang, Z., Zhang, H., Guan, B., Lu, F., Zhang, Z., Sui, Y., Jin, H., Shi, X.: Iterative refinement of project-level code context for precise code generation with compiler feedback. In: Ku, L.-W., Martins, A., Srikumar, V. (eds.) Findings of the Association for Computational Linguistics: ACL 2024, pp. 2336–2353. Association for Computational Linguistics, Bangkok, Thailand (2024). https://doi.org/10.18653/v1/2024.findings-acl.138 . https://aclanthology.org/2024.findings-acl.138/

Chakraborty, S., Ahmed, T., Ding, Y., Devanbu, P.T., Ray, B.: Natgen: generative pre-training by "naturalizing" source code. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 18–30 (2022)

Cai, S., Chen, H., Huang, Y., Ming, Z.: Compat: A compiler principles course assistant. In: KSEM (5), pp. 74–83 (2024)

Cassano, F., Gouwar, J., Lucchetti, F., Schlesinger, C., Freeman, A., Anderson, C.J., Feldman, M.Q., Greenberg, M., Jangda, A., Guha, A.: Knowledge transfer from high-resource to low-resource programming languages for code llms. Proc. ACM Program. Lang. **8**(OOPSLA2) (2024) https://doi.org/10.1145/3689735

Cao, Y., Liang, R., Chen, K., Hu, P.: Boosting neural networks to decompile optimized binaries. In: Proceedings of the 38th Annual Computer Security Applications Conference, pp. 508–518 (2022)

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L., Guestrin, C., Krishnamurthy, A.: Tvm: an automated end-to-end optimizing compiler for deep learning. In: Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation. OSDI'18, pp. 579–594. USENIX Association, USA (2018)

Cummins, C., Seeker, V., Armengol-Estapé, J., Markosyan, A.H., Synnaeve, G., Leather, H.: Don't Transform the Code, Code the Transforms: Towards Precise Code Rewriting using LLMs (2024). https://arxiv.org/abs/2410.08806

Cummins, C., Seeker, V., Grubisic, D., Elhoushi, M., Liang, Y., Roziere, B., Gehring, J., Gloeckle, F., Hazelwood, K., Synnaeve, G., Leather, H.: Large Language Models for Compiler Optimization (2023). https://arxiv.org/abs/2309.07062

Cummins, C., Seeker, V., Grubisic, D., Roziere, B., Gehring, J., Synnaeve, G., Leather, H.: Llm compiler: Foundation language models for compiler optimization. In: Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction. CC '25, pp. 141–153. Association for Computing Machinery, New York, NY, USA (2025). https://doi.org/10.1145/3708493.3712691 . https://doi.org/10.1145/3708493.3712691

Chen, M., Tworek, J., Jun, H., Yuan, Q., Oliveira Pinto, H.P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F.P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W.H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A.N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish,

S., Sutskever, I., Zaremba, W.: Evaluating Large Language Models Trained on Code (2021). https://arxiv.org/abs/2107.03374

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.d.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al.: Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 (2021)

Cummins, C., Wasti, B., Guo, J., Cui, B., Ansel, J., Gomez, S., Jain, S., Liu, J., Teytaud, O., Steiner, B., *et al.*: Compilergym: Robust, performant compiler optimization environments for ai research. In: 2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 92–105 (2022). IEEE

Cui, T., Yew, P.-C., McCamant, S., Zhai, A.: DeCOS: Data-efficient reinforcement learning for compiler optimization selection ignited by LLM. In: Proceedings of the 2025 International Conference on Supercomputing. ICS '25. Association for Computing Machinery, New York, NY, USA (2025). https://doi.org/10.1145/3721145.3725765 . https://doi.org/10.1145/3721145.3725765

Cui, F., Yin, C., Zhou, K., Xiao, Y., Sun, G., Xu, Q., Guo, Q., Liang, Y., Zhang, X., Song, D., *et al.*: Origen: Enhancing rtl code generation with code-to-code augmentation and self-reflection. In: Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design, pp. 1–9 (2024)

Chen, L., Zhang, S., Xu, F., Xing, Z., Wan, L., Zhang, X., Feng, Z.: A test-free semantic mistakes localization framework in neural code translation. arXiv preprint arXiv:2410.22818 (2024)

Ding, X., Chen, L., Emani, M., Liao, C., Lin, P.-H., Vanderbruggen, T., Xie, Z., Cerpa, A., Du, W.: Hpc-gpt: Integrating large language model for high-performance computing. In: Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis. SC-W '23, pp. 951–960. Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3624062.3624172 . https://doi.org/10.1145/3624062.3624172

DeLorenzo, M., Chowdhury, A.B., Gohil, V., Thakur, S., Karri, R., Garg, S., Rajendran, J.: Make every move count: Llm-based high-quality rtl code generation using mcts. arXiv preprint arXiv:2402.03289 (2024)

Duan, S., Kanakaris, N., Xiao, X., Ping, H., Zhou, C., Ahmed, N.K., Ma, G., Capota, M., Willke, T.L., Nazarian, S., Bogdan, P.: PerfRL: A Small Language Model Framework for Efficient Code Optimization (2025). https://arxiv.org/abs/2312.05657

Deligiannis, P., Lal, A., Mehrotra, N., Poddar, R., Rastogi, A.: Rustassistant: Using llms to fix compilation errors in rust code. In: 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE), pp. 267–279 (2024). IEEE Computer Society

De Moura, L., Bjørner, N.: Z3: an efficient smt solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. TACAS'08/ETAPS'08, pp. 337–340. Springer, Berlin, Heidelberg (2008)

Dong, Y., Ruan, C.F., Cai, Y., Lai, R., Xu, Z., Zhao, Y., Chen, T.: XGrammar: Flexible and Efficient Structured Generation Engine for Large Language Models (2024). https://arxiv.org/abs/2411.15100

Da Silva, A.F., Kind, B.C., Souza Magalhães, J.W., Rocha, J.N., Guimaraes, B.C.F., Pereira, F.M.Q.: Anghabench: A suite with one million compilable c benchmarks for code-size reduction. In: 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 378–390 (2021). IEEE

Dong, S., Wen, Y., Bi, J., Huang, D., Guo, J., Xu, J., Xu, R., Song, X., Hao, Y., Zhou, X., *et al.*: Qimengxpiler: Transcompiling tensor programs for deep learning systems with a neural-symbolic approach. In: 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25) (2025)

Deng, C., Wu, J., Feng, N., Wang, J., Long, M.: Compilerdream: Learning a compiler world model for general code optimization. arXiv preprint arXiv:2404.16077 (2024)

Dong, J., Yang, Y., Liu, T., Wang, Y., Qi, F., Tarokh, V., Rangadurai, K., Yang, S.: STARK: Strategic Team of Agents for Refining Kernels (2025). https://arxiv.org/abs/2510.16996

Fang, X., Mukhanov, L.: Towards llm-based optimization compilers. can llms learn how to apply a single peephole optimization? reasoning is all llms need! arXiv preprint arXiv:2412.12163 (2024)

Friedman, D., Wettig, A., Chen, D.: Learning transformer programs. Advances in Neural Information Processing Systems **36**, 49044–49067 (2023)

Gemini Team, Google: Gemini: A family of highly capable multimodal models. arXiv preprint arXiv:2312.11805 (2023)

Gao, S., Gao, C., Gu, W., Lyu, M.R.: Search-based llms for code optimization. In: 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE), pp. 578–590 (2025). IEEE

Gao, Y., Liang, L., Li, Y., Li, R., Wang, Y.: Function-level compilation provenance identification

with multi-faceted neural feature distillation and fusion. Electronics **13**(9) (2024) https://doi.org/10.3390/electronics13091692

Guo, Z.C., Moses, W.S.: Enabling transformers to understand low-level programs. In: 2022 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–9 (2022). IEEE

google-gemini: gemini-cli: A Google Gemini CLI and Python API. GitHub. Accessed: August 11, 2025 (2025)

Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: Mibench: A free, commercially representative embedded benchmark suite. In: Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No. 01EX538), pp. 3–14 (2001). IEEE

Grubisic, D., Seeker, V., Synnaeve, G., Leather, H., Mellor-Crummey, J., Cummins, C.: Priority sampling of large language models for compilers. In: Proceedings of the 4th Workshop on Machine Learning and Systems, pp. 91–97 (2024)

Gao, Z., Wang, H., Wang, Y., Zhang, C.: Virtual compiler is all you need for assembly code search. In: Ku, L.-W., Martins, A., Srikumar, V. (eds.) Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 3040–3051. Association for Computational Linguistics, Bangkok, Thailand (2024). https://doi.org/10.18653/v1/2024.acl-long.167 . https://aclanthology.org/2024.acl-long.167/

Gao, H., Yang, Y., Sun, M., Wu, J., Zhou, Y., Xu, B.: Clozemaster: Fuzzing rust compiler by harnessing llms for infilling masked real programs. In: 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE), pp. 712–712 (2025). IEEE Computer Society

Geng, H., Zhong, M., Zhang, P., Lv, F., Feng, X.: Optango: Multi-central representation learning against innumerable compiler optimization for binary diffing. In: ISSRE, pp. 774–785 (2023). https://doi.org/10.1109/ISSRE59848.2023.00013

Hong, C., Bhatia, S., Cheung, A., Shao, Y.S.: Autocomp: Llm-driven code optimization for tensor accelerators. arXiv preprint arXiv:2505.18574 (2025)

Hu, L., Chen, G., Shang, X., Cheng, S., Wu, B., LiGangyang, L., Zhu, X., Zhang, W., Yu, N.: CompileAgent: Automated real-world repo-level compilation with tool-integrated LLM-based agent system. In: Che, W., Nabende, J., Shutova, E., Pilehvar, M.T. (eds.) Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 2078–2091. Association for Computational Linguistics, Vienna, Austria (2025). https://aclanthology.org/2025.acl-long.103/

Huang, D., Dai, J., Weng, H., Wu, P., Qing, Y., Cui, H., Guo, Z., Zhang, J.: Effilearner: Enhancing efficiency of generated code via self-optimization. Advances in Neural Information Processing Systems **37**, 84482–84522 (2024)

Heckel, K.: Neuroevolutionary compiler control for code optimization. In: Proceedings of the Companion Conference on Genetic and Evolutionary Computation, pp. 2362–2365 (2023)

Henning, J.L.: Spec cpu2006 benchmark descriptions. ACM SIGARCH Computer Architecture News **34**(4), 1–17 (2006)

Hu, P., Liang, R., Chen, K.: Degpt: Optimizing decompiler output with llm. In: Proceedings 2024 Network and Distributed System Security Symposium, vol. 267622140 (2024)

Huang, D., Nan, Z., Hu, X., Jin, P., Peng, S., Wen, Y., Zhang, R., Du, Z., Guo, Q., Pu, Y., Chen, Y.: Anpl: towards natural programming with interactive decomposition. In: Proceedings of the 37th International Conference on Neural Information Processing Systems. NIPS '23. Curran Associates Inc., Red Hook, NY, USA (2023)

He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)

Italiano, D., Cummins, C.: Finding missed code size optimizations in compilers using large language models. In: Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction, pp. 81–91 (2025)

Ishida, S., Corrado, G., Fedoseev, G., Yeo, H., Russell, L., Shotton, J., Henriques, J.F., Hu, A.: Langprop: A code optimization framework using large language models applied to driving. In: ICLR 2024 Workshop on Large Language Model (LLM) Agents (2024). https://openreview.net/forum?id=JQJJ9PkdYC

Ibrahimzada, A.R., Ke, K., Pawagi, M., Abid, M.S., Pan, R., Sinha, S., Jabbarvand, R.: Alphatrans: A neuro-symbolic compositional approach for repository-level code translation and validation. Proceedings of the ACM on Software Engineering **2**(FSE), 2454–2476 (2025)

Just, R., Jalali, D., Ernst, M.D.: Defects4j: A database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis, pp. 437–440 (2014)

Jana, P., Jha, P., Ju, H., Kishore, G., Mahajan, A.,

Ganesh, V.: Cotran: An llm-based code translator using reinforcement learning with feedback from compiler and symbolic execution. In: ECAI (2024)

Jin, X., Larson, J., Yang, W., Lin, Z.: Binary Code Summarization: Benchmarking ChatGPT/GPT-4 and Other Large Language Models (2023). https://arxiv.org/abs/2312.09601

Jin, L., Ruan, Z., Mai, H., Shang, J.: Verilocc: End-to-end cross-architecture register allocation via llm. arXiv preprint arXiv:2506.17506 (2025)

Jiao, M., Yu, T., Li, X., Qiu, G., Gu, X., Shen, B.: On the evaluation of neural code translation: Taxonomy and benchmark. In: Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering, pp. 1529–1541 (2023)

Jiang, H., Zhu, J., Wan, Y., Fang, B., Zhang, H., Jin, R., Guan, Q.: Can large language models understand intermediate representations in compilers? arXiv preprint arXiv:2502.06854 (2025)

Khan, W., Alrabaee, S., Al-kfairy, M., Tang, J., Raymond Choo, K.-K.: Compiler-provenance identification in obfuscated binaries using vision transformers. Forensic Science International: Digital Investigation **49**, 301764 (2024) https://doi.org/10.1016/j.fsidi.2024.301764 . DFRWS USA 2024 - Selected Papers from the 24th Annual Digital Forensics Research Conference USA

Kitchenham, B., Charters, S., et al.: Guidelines for performing systematic literature reviews in software engineering (2007)

Kadosh, T., Hasabnis, N., Soundararajan, P., Vo, V.A., Capota, M., Ahmed, N., Pinter, Y., Oren, G.: OMPar: Automatic Parallelization with AI-Driven Source-to-Source Compilation (2024). https://arxiv.org/abs/2409.14771

Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems **25** (2012)

Kabir, A., Wang, S., Tian, Y., Chen, T.-H., Asaduzzaman, M., Zhang, W.: Zs4c: Zero-shot synthesis of compilable code for incomplete code snippets using llms. ACM Transactions on Software Engineering and Methodology **34**(4), 1–30 (2025)

Kang, S., Yoon, J., Yoo, S.: Large language models are few-shot testers: Exploring llm-based general bug reproduction. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp. 2312–2323 (2023). IEEE

Lv, J., He, X., Liu, Y., Dai, X., Shen, A., Li, Y., Hao, J., Ding, J., Hu, Y., Yin, S.: Hpctranscompile: An ai compiler generated dataset for high-performance cuda transpilation and llm preliminary exploration. arXiv preprint arXiv:2506.10401 (2025)

Lindner, D., Kramár, J., Farquhar, S., Rahtz, M., McGrath, T., Mikulik, V.: Tracr: Compiled transformers as a laboratory for interpretability. Advances in Neural Information Processing Systems **36**, 37876–37899 (2023)

Li, J., Li, S., Gao, Z., Shi, Q., Li, Y., Wang, Z., Huang, J., WangHaojie, W., Wang, J., Han, X., Liu, Z., Sun, M.: TritonBench: Benchmarking large language model capabilities for generating triton operators. In: Che, W., Nabende, J., Shutova, E., Pilehvar, M.T. (eds.) Findings of the Association for Computational Linguistics: ACL 2025, pp. 23053–23066. Association for Computational Linguistics, Vienna, Austria (2025). https://doi.org/10.18653/v1/2025.findings-acl.1183 . https://aclanthology.org/2025.findings-acl.1183/

Lopes, N.P., Lee, J., Hur, C.-K., Liu, Z., Regehr, J.: Alive2: bounded translation validation for llvm. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. PLDI 2021, pp. 65–79. Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3453483.3454030 . https://doi.org/10.1145/3453483.3454030

Luo, T., Lee, H., Johnson, J.: Neural shape compiler: A unified framework for transforming between text, point cloud, and program. Transactions on Machine Learning Research (2023)

Lu, Y., Liu, S., Zhang, Q., Xie, Z.: Rtllm: An open-source benchmark for design rtl generation with large language model. In: 2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 722–727 (2024). IEEE

Lin, H., Maas, M., Roquemore, M., Hasanzadeh, A., Lewis, F., Simonson, Y., Yang, T.-W., Yazdanbakhsh, A., Altinbüken, D., Papa, F., et al.: Eco: An llm-driven efficient code optimizer for warehouse scale computers. arXiv preprint arXiv:2503.15669 (2025)

Liu, M., Pinckney, N., Khailany, B., Ren, H.: Verilogeval: Evaluating large language models for verilog code generation. In: 2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD), pp. 1–8 (2023). IEEE

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., Riedel, S., Kiela, D.: Retrieval-augmented generation for knowledge-intensive nlp tasks. In: Proceedings of the 34th International Conference on Neural Information Processing Systems. NIPS '20. Curran Associates Inc., Red Hook, NY, USA (2020)

Li, X., Sun, X., Wang, A., Li, J., Chris, S.: Cuda-l1: Improving cuda optimization via contrastive reinforcement learning. arXiv preprint arXiv:2507.14111 (2025)

Mannarswamy, S., Das, D.: Learning to Combine Instructions in LLVM Compiler (2022). https://arxiv.org/abs/2202.12379

Mündler, N., He, J., Wang, H., Sen, K., Song, D., Vechev, M.: Type-constrained code generation with language models. Proceedings of the ACM on Programming Languages **9**(PLDI), 601–626 (2025)

Mammadli, R., Jannesari, A., Wolf, F.: Static neural compiler optimization via deep reinforcement learning. In: 2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar), pp. 1–11 (2020). IEEE

Madaan, A., Shypula, A., Alon, U., Hashemi, M., Ranganathan, P., Yang, Y., Neubig, G., Yazdanbakhsh, A.: Learning performance-improving code edits. arXiv preprint arXiv:2302.07867 (2023)

Macedo, M., Tian, Y., Cogo, F., Adams, B.: Exploring the impact of the output format on the evaluation of large language models for code translation. In: Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering, pp. 57–68 (2024)

Nichols, D., Davis, J.H., Xie, Z., Rajaram, A., Bhatele, A.: Can large language models write parallel code? In: Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing, pp. 281–294 (2024)

Niu, C., Li, C., Ng, V., Lo, D., Luo, B.: Fair: Flow type-aware pre-training of compiler intermediate representations. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. ICSE '24. Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3597503.3608136 . https://doi.org/10.1145/3597503.3608136

NVIDIA Corporation: CUTLASS Python Interface Overview. https://docs.nvidia.com/cutlass/media/docs/pythonDSL/overview.html. Accessed: August 11, 2025 (2025)

Nakkab, A., Zhang, S.Q., Karri, R., Garg, S.: Rome was not built in a single step: Hierarchical prompting for llm-based chip design. In: Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD, pp. 1–11 (2024)

Ouyang, A., Guo, S., Arora, S., Zhang, A.L., Hu, W., Re, C., Mirhoseini, A.: Kernelbench: Can LLMs write efficient GPU kernels? In: Forty-second International Conference on Machine Learning (2025). https://openreview.net/forum?id=yeoN1iQT1x

Ou, X., Li, C., Jiang, Y., Xu, C.: The mutators reloaded: Fuzzing compilers with large language model generated mutation operators. In: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4, pp. 298–312 (2024)

OpenAI: GPT-4 Technical Report (2024). https://arxiv.org/abs/2303.08774

Palkowski, M., Gruzewski, M.: Automatic generation of opencl code through polyhedral compilation with llm. In: 2024 19th Conference on Computer Science and Intelligence Systems (FedCSIS), pp. 671–676 (2024). IEEE

Peng, Y., Gotmare, A.D., Lyu, M.R., Xiong, C., Savarese, S., Sahoo, D.: Perfcodegen: Improving performance of llm generated code with execution feedback. In: 2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge), pp. 1–13 (2025). IEEE

Pan, R., Ibrahimzada, A.R., Krishna, R., Sankar, D., Wassi, L.P., Merler, M., Sobolev, B., Pavuluri, R., Sinha, S., Jabbarvand, R.: Lost in translation: A study of bugs introduced by large language models while translating code. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, pp. 1–13 (2024)

Puri, R., Kung, D.S., Janssen, G., Zhang, W., Domeniconi, G., Zolotov, V., Dolby, J., Chen, J., Choudhury, M., Decker, L., Thost, V., Buratti, L., Pujar, S., Ramji, S., Finkler, U., Malaika, S., Reiss, F.: CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks (2021). https://arxiv.org/abs/2105.12655

Purschke, N., Kirchner, S., Knoll, A.: Speedgen: Enhancing code efficiency through large language model-based performance optimization. In: 2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 1–12 (2025). IEEE

Pirkelbauer, P., Liao, C.: Compilergpt: Leveraging large language models for analyzing and acting on compiler optimization reports. arXiv preprint arXiv:2506.06227 (2025)

Pan, H., Lin, H., Luo, H., Liu, Y., Yao, K., Zhang, L., Xing, M., Wu, Y.: Compiler-r1: Towards agentic compiler auto-tuning with reinforcement learning. arXiv preprint arXiv:2506.15701 (2025)

Papineni, K., Roukos, S., Ward, T., Zhu, W.-J.: Bleu:

a method for automatic evaluation of machine translation. In: Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, pp. 311–318 (2002)

Peng, D., Zheng, S., Li, Y., Ke, G., He, D., Liu, T.-Y.: How could neural networks understand programs? In: International Conference on Machine Learning, pp. 8476–8486 (2021). PMLR

Rong, Y., Du, T., Li, R., Bao, W.: Integrating llm-based code optimization with human-like exclusionary reasoning for computational education. Journal of King Saud University Computer and Information Sciences **37**(5), 87 (2025)

Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., Ma, S.: Codebleu: a method for automatic evaluation of code synthesis. arXiv preprint arXiv:2009.10297 (2020)

Roziere, B., Lachaux, M.-A., Chanussot, L., Lample, G.: Unsupervised translation of programming languages. In: Proceedings of the 34th International Conference on Neural Information Processing Systems. NIPS '20. Curran Associates Inc., Red Hook, NY, USA (2020)

Romero Rosas, M.A., Torres Sanchez, M.A., Eigenmann, R.: Should ai optimize your code? a comparative study of classical optimizing compilers versus current large language models. In: Proceedings of the 2025 Supercomputing Asia Conference, pp. 22–29 (2025)

Roziere, B., Zhang, J., Charton, F., Harman, M., Synnaeve, G., Lample, G.: Leveraging automated unit tests for unsupervised code translation. In: International Conference on Learning Representations (2022). https://openreview.net/forum?id=cmt-6KtR4c4

Ren, X., Zhang, T., Xu, X., Zheng, Y.-C., Zhang, S.: Leveraging machine learning for quantum compilation optimization. In: Proceedings of the 61st ACM/IEEE Design Automation Conference, pp. 1–4 (2024)

Sajjadinasab, R., Arora, S., Drepper, U., Sanaullah, A., Herbordt, M.: A graph-based algorithm for optimizing gcc compiler flag settings. In: 2024 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–8 (2024). IEEE

Shaw, P., Cohan, J., Eisenstein, J., Lee, K., Berant, J., Toutanova, K.: ALTA: Compiler-based analysis of transformers. Transactions on Machine Learning Research (2025)

Sun, T., Chai, L., Yang, J., Yin, Y., Guo, H., Liu, J., Wang, B., Yang, L., Li, Z.: UniCoder: Scaling code large language model via universal code. In: Ku, L.-W., Martins, A., Srikumar, V. (eds.) Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 1812–1824. Association for Computational Linguistics, Bangkok, Thailand (2024). https://doi.org/10.18653/v1/2024.acl-long.100 . https://aclanthology.org/2024.acl-long.100/

Siddiq, M.L., Silva Santos, J.C., Devareddy, S., Muller, A.: Sallm: Security assessment of generated code. In: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops. ASEW '24, pp. 54–65. Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3691621.3694934 . https://doi.org/10.1145/3691621.3694934

Saldyt, L., Kambhampati, S.: Algorithmic Language Models with Neurally Compiled Libraries (2025). https://arxiv.org/abs/2407.04899

Sibaee, S., Najar, O., Ghouti, L., Koubaa, A.: Llms as compiler for arabic programming language. arXiv preprint arXiv:2403.16087 (2024)

Szafraniec, M., Roziere, B., Leather, H.J., Labatut, P., Charton, F., Synnaeve, G.: Code translation with compiler representations. In: The Eleventh International Conference on Learning Representations (2023). https://openreview.net/forum?id=XomEU3eNeSQ

Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Bi, X., Zhang, H., Zhang, M., Li, Y.K., Wu, Y., Guo, D.: DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models (2024). https://arxiv.org/abs/2402.03300

Shen, L., Yang, Q., Zheng, Y., Li, M.: Autoiot: Llm-driven automated natural language programming for aiot applications. In: Mobicom 2025 (2025)

Tabnine: Tabnine: AI Code Completion Tool. https://www.tabnine.com/. Accessed: August 11, 2025 (2022)

Thakur, S., Ahmad, B., Pearce, H., Tan, B., Dolan-Gavitt, B., Karri, R., Garg, S.: Verigen: A large language model for verilog code generation. ACM Trans. Des. Autom. Electron. Syst. **29**(3) (2024) https://doi.org/10.1145/3643681

TehraniJamsaz, A., Bhattacharjee, A., Chen, L., Ahmed, N.K., Yazdanbakhsh, A., Jannesari, A.: Coderosetta: Pushing the boundaries of unsupervised code translation for parallel programming. In: The Thirty-eighth Annual Conference on Neural Information Processing Systems (2024). https://openreview.net/forum?id=V6hrg4O9gg

Thakur, S., Blocklove, J., Pearce, H., Tan, B., Garg, S., Karri, R.: Autochip: Automating hdl generation

using llm feedback. arXiv preprint arXiv:2311.04887 (2023)

Tavarageri, S., Goyal, G., Avancha, S., Kaul, B., Upadrasta, R.: Ai powered compiler techniques for dl code optimization. arXiv preprint arXiv:2104.05573 (2021)

Tillet, P., Kung, H.T., Cox, D.: Triton: an intermediate language and compiler for tiled neural network computations. In: Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages. MAPL 2019, pp. 10–19. Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3315508.3329973 . https://doi.org/10.1145/3315508.3329973

Tan, H., Luo, Q., Li, J., Zhang, Y.: LLM4Decompile: Decompiling binary code with large language models. In: Al-Onaizan, Y., Bansal, M., Chen, Y.-N. (eds.) Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, pp. 3473–3487. Association for Computational Linguistics, Miami, Florida, USA (2024). https://doi.org/10.18653/v1/2024.emnlp-main.203 . https://aclanthology.org/2024.emnlp-main.203/

Taneja, J., Laird, A., Yan, C., Musuvathi, M., Lahiri, S.K.: Llm-vectorizer: Llm-based verified loop vectorizer. In: Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization. CGO '25, pp. 137–149. Association for Computing Machinery, New York, NY, USA (2025). https://doi.org/10.1145/3696443.3708929 . https://doi.org/10.1145/3696443.3708929

Tang, S., Priebe, C., Mahapatra, R., Qin, L., Esmaeilzadeh, H.: Compiler optimization via llm reasoning for efficient model serving. arXiv preprint arXiv:2506.01374 (2025)

Tsimpourlas, F., Petoumenos, P., Xu, M., Cummins, C., Hazelwood, K., Rajan, A., Leather, H.: BenchDirect: A Directed Language Model for Compiler Benchmarks (2023). https://arxiv.org/abs/2303.01557

Taylor, A., Vassar, A., Renzella, J., Pearce, H.: dcc –help: Transforming the role of the compiler by generating context-aware error explanations with large language models. In: Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1. SIGCSE 2024, pp. 1314–1320. Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3626252.3630822 . https://doi.org/10.1145/3626252.3630822

Wen, Y., Guo, Q., Fu, Q., Li, X., Xu, J., Tang, Y., Zhao, Y., Hu, X., Du, Z., Li, L., *et al.*: Babeltower: Learning to auto-parallelized program translation. In: International Conference on Machine Learning, pp. 23685–23700 (2022). PMLR

Weiss, G., Goldberg, Y., Yahav, E.: Thinking like transformers. In: International Conference on Machine Learning, pp. 11080–11090 (2021). PMLR

Wang, X., Hui, X., Liao, C., Shen, X.: Reductive analysis with compiler-guided large language models for input-centric code optimizations. Proc. ACM Program. Lang. **9**(PLDI) (2025) https://doi.org/10.1145/3729282

Willard, B.T., Louf, R.: Efficient Guided Generation for Large Language Models (2023). https://arxiv.org/abs/2307.09702

Wang, Z., O'Boyle, M.: Machine learning in compiler optimization. Proceedings of the IEEE **106**(11), 1879–1901 (2018) https://doi.org/10.1109/JPROC.2018.2817118

Wang, H., Qu, W., Katz, G., Zhu, W., Gao, Z., Qiu, H., Zhuge, J., Zhang, C.: jtrans: jump-aware transformer for binary code similarity detection. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2022, pp. 1–13. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3533767.3534367 . https://doi.org/10.1145/3533767.3534367

Wang, T., Wang, R., Chen, Y., Yu, L., Pan, Z., Zhang, M., Ma, H., Zheng, J.: Enhancing black-box compiler option fuzzing with llm through command feedback. In: 2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE), pp. 319–330 (2024). IEEE

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E.H., Le, Q.V., Zhou, D.: Chain-of-thought prompting elicits reasoning in large language models. In: Proceedings of the 36th International Conference on Neural Information Processing Systems. NIPS '22. Curran Associates Inc., Red Hook, NY, USA (2022)

Wong, W.K., Wu, D., Wang, H., Li, Z., Liu, Z., Wang, S., Tang, Q., Nie, S., Wu, S.: Decllm: Llm-augmented recompilable decompilation for enabling programmatic use of decompiled code. Proceedings of the ACM on Software Engineering **2**(ISSTA), 1841–1864 (2025)

Wei, Y., Xia, C.S., Zhang, L.: Copiloting the copilots: Fusing large language models with completion engines for automated program repair. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2023, pp. 172–184. Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3611643.3616271 . https://doi.org/10.1145/3611643.3616271

Wang, Y., Ye, W., Guo, P., He, Y., Wang, Z., Tian,

B., He, S., Sun, G., Shen, Z., Chen, S., et al.: Symrtlo: Enhancing rtl code optimization with llms and neuron-inspired symbolic reasoning. arXiv preprint arXiv:2504.10369 (2025)

Wang, N., Yao, B., Zhou, J., Hu, Y., Wang, X., Guan, N., Jiang, Z.: Insights from verification: Training a verilog generation llm with reinforcement learning with testbench feedback. arXiv preprint arXiv:2504.15804 (2025)

Xu, X., Feng, S., Ye, Y., Shen, G., Su, Z., Cheng, S., Tao, G., Shi, Q., Zhang, Z., Zhang, X.: Improving binary code similarity transformer models by semantics-driven instruction deemphasis. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2023, pp. 1106–1118. Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3597926.3598121 . https://doi.org/10.1145/3597926.3598121

Xu, C., Guo, H., Cen, C., Chen, M., Tao, X., He, J.: Efficient program optimization through knowledge-enhanced lora fine-tuning of large language models. The Journal of Supercomputing **81**(8), 1006 (2025)

Xiong, C., Liu, C., Li, H., Li, X.: Hlspilot: Llm-based high-level synthesis. In: Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design, pp. 1–9 (2024)

Xu, S., Li, Z., Mei, K., Zhang, Y.: Aios compiler: Llm as interpreter for natural language programming and flow programming of ai agents. arXiv preprint arXiv:2405.06907 (2024)

Xia, C.S., Wei, Y., Zhang, L.: Automated program repair in the era of large pre-trained language models. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp. 1482–1494 (2023). https://doi.org/10.1109/ICSE48619.2023.00129

Xu, Q., Yang, D., Zhang, L.: Code optimization chain-of-thought: Structured understanding and self-checking. In: Proceedings of the 2024 4th International Conference on Artificial Intelligence, Big Data and Algorithms, pp. 425–430 (2024)

Xu, X., Zhang, Z., Feng, S., Ye, Y., Su, Z., Jiang, N., Cheng, S., Tan, L., Zhang, X.: Lmpa: Improving decompilation by synergy of large language model and program analysis. arXiv preprint arXiv:2306.02546 (2023)

Yang, C., Deng, Y., Lu, R., Yao, J., Liu, J., Jabbarvand, R., Zhang, L.: Whitefox: White-box compiler fuzzing empowered by large language models. Proceedings of the ACM on Programming Languages **8**(OOPSLA2), 709–735 (2024)

Ye, T., Ma, T., Zhang, X., Yu, H., Yin, J., Wang, W.: A problem-oriented perspective and anchor verification for code optimization. arXiv preprint arXiv:2406.11935 (2024)

Yin, X., Ni, C., Nguyen, T.N., Wang, S., Yang, X.: Rectifier: Code translation with corrector via llms. arXiv preprint arXiv:2407.07472 (2024)

Zhang, H., David, C., Wang, M., Paulsen, B., Kroening, D.: Scalable, validated code translation of entire projects using large language models. Proceedings of the ACM on Programming Languages **9**(PLDI), 1616–1641 (2025)

Zhong, M., LYU, F., Wang, L., Geng, H., Qiu, L., Cui, H., Feng, X.: Comback: A versatile dataset for enhancing compiler backend development efficiency. In: The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track (2024). https://openreview.net/forum?id=vfju5hjrJw

Zhong, M., Lv, F., Wang, L., Qiu, L., Wang, Y., Liu, Y., Cui, H., Feng, X., Xue, J.: Vega: Automatically generating compiler backends using a pre-trained transformer model. In: Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization. CGO '25, pp. 90–106. Association for Computing Machinery, New York, NY, USA (2025). https://doi.org/10.1145/3696443.3708931 . https://doi.org/10.1145/3696443.3708931

Zhang, Y., Song, W., Ji, Z., Danfeng, Yao, Meng, N.: How well does LLM generate security tests? (2023). https://arxiv.org/abs/2310.00710

Zhai, Y., Yang, S., Pan, K., Zhang, R., Liu, S., Liu, C., Ye, Z., Ji, J., Zhao, J., Zhang, Y., *et al.*: Enabling tensor language model to assist in generating {High-Performance} tensor programs for deep learning. In: 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), pp. 289–305 (2024)

Zhang, S., Zhao, J., Xia, C., Wang, Z., Chen, Y., Cui, H.: Introducing compiler semantics into large language models as programming language translators: A case study of C to x86 assembly. In: Al-Onaizan, Y., Bansal, M., Chen, Y.-N. (eds.) Findings of the Association for Computational Linguistics: EMNLP 2024, pp. 996–1011. Association for Computational Linguistics, Miami, Florida, USA (2024). https://doi.org/10.18653/v1/2024.findings-emnlp.55 . https://aclanthology.org/2024.findings-emnlp.55/

Zhang, S., Zhao, J., Xia, C., Wang, Z., Chen, Y., Feng, X., Cui, H.: LEGO-Compiler: Enhancing Neural Compilation Through Translation Composability (2025). https://arxiv.org/abs/2505.20356

Zhang, Q., Zhang, T., Zhai, J., Fang, C., Yu, B.,

Sun, W., Chen, Z.: A critical review of large language model on software engineering: An example from chatgpt and automated program repair. arXiv preprint arXiv:2310.08879 (2023)