Effect of Abstractions and Prompting Strategies on LLM-Guided High-Performance Optimizations

Jiří Klepl[0000-0002-2231-4073], Matyáš Brabec[0009-0008-4470-2748], and Martin Kruliš[0000-0002-0985-8949]

Charles University, Prague, Czech Republic
{klepl,brabec,krulis}@d3s.mff.cuni.cz

arXiv:2608.08085v1 [cs.DC] 8 Aug 2026

Abstract. Code performance optimization is a vital aspect of modern software development, as it enables faster response times and reduced resource usage. These optimizations require a deep understanding of low-level hardware details and the intricacies of parallel processing, making them challenging even for experienced developers. With the advent of Large Language Models (LLMs), which are increasingly capable of generating and understanding code, there is growing interest in incorporating these models into automated code optimization processes. Traditionally, this automation involves transcribing the source code into a domain-specific representation that can be auto-tuned using grid search or machine learning algorithms, while adhering to strict rules and a limited set of feasible transformations to ensure verifiability. LLMs incorporate high-level code semantics and can thus perform transformations that go beyond verifiable automated optimizations. This paper investigates whether the traditional abstractions used in automated code optimization improve the performance and correctness of LLM-guided optimizations of parallel HPC applications. We evaluate this using the PolyBench benchmark suite and demonstrate that, in our evaluated setting, LLMs provided with specific optimization goals achieve better measured performance and validity rates when generating C code compared to creating computation pipelines and optimization schedules with established frameworks, suggesting that future development should explore alternative approaches for verifiable LLM-guided code optimization.

Keywords: Large language model, Code optimization, Domain-specific languages, High-performance computing, Prompt strategies

PREPRINT

1 Introduction

Code performance optimization is an essential part of software development, ensuring quick response times and efficient use of computational resources. This challenge is particularly pronounced in high-performance computing (HPC) applications that often involve complex computations over large datasets. This challenge is addressed at multiple levels, from low-level optimizations performed by compilers like GCC and LLVM and tools based on polyhedral models [6,14] to

algorithm parallelization (typically performed by human experts) or auto-tuning frameworks, such as OpenTuner [3]. However, achieving optimal performance requires a deep understanding of the underlying hardware (e.g., memory hierarchies and parallelism ranging from SIMD to distributed computing) and the specific characteristics of the target application, such as exploitable invariants or optimizable memory access patterns.

Common optimization techniques that target memory hierarchies and parallelism include loop transformations such as tiling, loop interchange or skewing, unrolling, fusion, and parallelization strategies (e.g., OpenMP directives or SIMD vectorization). They are typically applied by the aforementioned polyhedral model-based frameworks or human experts, and then fine-tuned using auto-tuning frameworks. However, without deep expertise in hardware architectures and the mathematical models underlying these optimizations, applying these methods often leads to suboptimal performance [1].

To address this, domain-specific languages (DSLs) and frameworks, such as Halide [30], Exo [15], and Noarr [18], have been developed to simplify the process of specifying and tuning high-performance code for human programmers, replacing mathematical affine models with higher-level abstractions that better capture the semantics of computations and the intent of the optimizations.

In practice, applying these methods still requires human expertise to prepare optimization strategies that can be adapted to specific hardware and application characteristics. Alternative approaches based on polyhedral model optimizations are limited by the strict mathematical rules they must follow to ensure verifiability of the transformations and by speculative cost models.

Large language models (LLMs) have recently shown great promise in understanding and generating code, with models such as Codex [11] demonstrating impressive results. Most of these works focus on web development or general programming tasks, often in languages that are not typically used for performance-critical applications. LLMs have also shown potential in generating HPC-based code [25,33] and optimizing existing code for better performance, usually by proposing directives to lead compiler optimizer or enable parallelization that are easily applicable and verifiable by a formal toolchain [9,10]. However, these approaches have limited expressiveness and may not achieve the optimal performance on tasks that require more complex or non-trivial optimizations, especially if the optimizations involve changes to the algorithm logic.

Recent studies have explored the use of LLMs to generate or optimize high-performance code using specialized prompting [7,8,13,28], but these works typically focus on specific languages or frameworks.

Our work aims to explore the capabilities of LLMs across multiple domain-specific languages for high-performance parallel code development and optimization, as well as across different prompting strategies for generating the optimized code. The goal is to determine whether LLMs can effectively leverage the abstractions provided by these frameworks, or whether generating optimized parallel C code directly yields better performance and correctness.

PREPRINT

As the representative LLM for our study, we chose GPT-5.1 [26], one of the state-of-the-art models for code generation at the time of writing, offering reasonable performance while being priced favorably for our budget.

We formulate our objectives in the following research questions (RQs):

RQ1 Can state-of-the-art frameworks for loop optimizations improve (e.g., add verification) LLM-guided optimization over direct code generation?
RQ2 Do more detailed prompts improve the performance and correctness of LLM-guided loop optimizations?
RQ3 Does following an explicit optimization plan generated prior to code generation improve the performance and correctness of LLM-guided optimizations?
RQ4 How do the approaches discussed in the previous questions compare in terms of performance gain, concurrency, precision, and overall prompt costs?
RQ5 Can LLMs apply optimizations that are not easily expressible in existing DSLs and frameworks reliably without compromising the optimized code?

To study these research questions, we forgo the use of an iterative process in which the LLM refines its outputs based on feedback from the compiler or runtime results. While such an approach is effective in practice [12,21,32], it also introduces additional variables that can obscure the studied factors. Furthermore, the iterative approach is often costly, and we aim to target reasonably priced solutions. Similarly, we consider a workflow that uses the LLM to both introduce a specific optimization framework (or DSL) and generate the optimized code, rather than one based on manual implementation in the target framework.

Answering the presented questions should help establish guidelines for using LLMs in high-performance parallel code optimization and clarify where existing optimization abstractions help or hinder LLM-guided optimization. In addition, we provide a replication package [17] with the entire codebase, prompts, and experiment results to facilitate further research in this area.

The paper is organized as follows. Section 2 outlines the common optimization techniques in the studied DSLs and frameworks. Our proposed approach is formulated in Section 3. Section 4 summarizes the experimental setup and results. Section 5 discusses related work, and Section 6 concludes the paper.

2 Background

Loop transformations and data layout transformations are common techniques, but they can be challenging to apply correctly and effectively. Let us demonstrate this on a well-known example — the naïve matrix multiplication. Mathematically, it can be defined as $C' = C + A \cdot B$. Where C, A, and B are matrices of sizes $M \times N$, $M \times K$, and $K \times N$, respectively. An implementation in the C language may look like the following:

```
void matmul(int M, int N, int K, float *C, float *A, float *B) {
  for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++)
      for (int k = 0; k < K; k++)
        C[i*N + j] += A[i*K + k] * B[k*N + j];
}
```

PREPRINT

Applying loop tiling with tile sizes of 32 for both the i and j loops, changing the B matrix layout to be column-major for better memory access patterns, and unrolling the innermost k loop by a factor of 4 will have the following result:

```
for (int io = 0; io < M; io += 32)              // Tiled i loop
  for (int jo = 0; jo < N; jo += 32)            // Tiled j loop
    for (int i = io; i < min(io+32, M); i++)    // Inner i
      for (int j = jo; j < min(jo+32, N); j++) // Inner j
        for (int k = 0; k < K; k += 4) {        // Unrolled k loop
          C[i*N + j] += A[i*K + (k + 0)] * B[j*K + (k + 0)];
          C[i*N + j] += A[i*K + (k + 1)] * B[j*K + (k + 1)];
          C[i*N + j] += A[i*K + (k + 2)] * B[j*K + (k + 2)];
          C[i*N + j] += A[i*K + (k + 3)] * B[j*K + (k + 3)];
}
```

These optimizations prepare the code for possible parallelization of the outer loops or vectorization of the unrolled innermost loop. Although semantically equivalent, the code is significantly more complex compared to the original; thus, it is more error-prone and harder to maintain. For this reason, various frameworks and domain-specific languages (DSLs) have been developed to express algorithms and their optimizations more easily and safely, without having to manually rewrite the whole code for each transformation, and possibly with automatically added vectorization or parallelization.

Examples of such frameworks that we selected as good representatives of different approaches are Halide [30], Exo [15], and Noarr [18]. All three frameworks aim to facilitate the development of high-performance code through higher-level abstractions and optimizations, and offer ways to separate the definition of the algorithmic logic from the optimization and scheduling of computations. We illustrate the basic usage of each framework using the previously introduced naïve matrix multiplication example.

2.1 Halide

Halide [30] is a domain-specific language (DSL) embedded in C++, primarily designed for image processing, stencil computations, and tensor computations. The following code snippet illustrates an example of how to implement and optimize matrix multiplication using Halide:

```
Buffer<float> C(M, N), A(M, K), B(K, N);

// Algorithm definition
matmul(i, j) = C(i, j);
matmul(i, j) += A(i, k) * B(k, j);

// Schedule definition
matmul.update().tile(i, j, i_i, j_i, 32, 32).fuse(i, j, tile_idx)
               .parallel(tile_idx).vectorize(k, 4);
```

The core concept in Halide is the separation of the algorithm from its execution schedule. The algorithm is defined as a computation pipeline broken down into a series of *functions* that describe how to compute the output values from the input values.

After defining a computation pipeline, Halide allows specifying various optimizations via a user-defined *schedule* that describes how the computation should

PREPRINT

be executed. In the example above, we modify the second stage of the `matmul` function by tiling the iteration space of `i` and `j` into 32×32, parallelizing two outer loops, and explicitly vectorizing the innermost loop. Even with these restrictions, the scheduling language is expressive enough to achieve significant performance improvements for a wide range of applications [22,30].

Since Halide is specifically designed for stencil-like computations and tensor pipelines, it is expected to perform well on such tasks when the computation and schedule are expressed idiomatically. However, the trade-off is that some algorithms are more challenging to express in an idiomatic Halide form, often requiring non-trivial workarounds that break the computation into iteratively applied steps. While such workarounds can help express algorithms with nested loops, they also increase the performance overhead, which has to be mitigated by Halide's optimizations and user-defined schedules.

2.2 Exo

Exo [15] is a Python-based framework for generating high-performance code for target platforms. Matrix multiplication can be implemented in Exo as follows:

```
@proc
def gemm(
  ni: size, nj: size, nk: size,
  C: float[ni,nj]@DRAM, A: float[ni,nk]@DRAM, B: float[nk,nj]@DRAM
):
  for i in seq(0, ni):
    for j in seq(0, nj):
      for k in seq(0, nk):
        C[i, j] += A[i, k] * B[k, j]
```

The core concept in Exo is defining computations as *procedures* (`@proc`) that describe the operations to be performed on the input data. The procedures can include standard control flow constructs such as (`for`) loops and conditionals, allowing for a more general representation of algorithms compared to Halide; however, the language still strictly separates data computation from bounds and memory layout specifications — for example, conditionals based on data values are not allowed. This ensures that the Exo engine can reason about the computation and apply optimizations safely, while sacrificing some flexibility in algorithm expression.

Similar to Halide scheduling, Exo allows specifying various optimizations through externally specified *rewrite rules* [15]. These rules are applied using pattern matching to specific operations or loop nests within the procedure, together with performing type inference, allowing the Exo engine to verify the correctness of the transformations before generating optimized C code. It also supports scheduling operations such as loop parallelization (using `parallelize_loop`).

In contrast to Halide, Exo is designed for general loop-based computations. After performing the specified optimization rewrites and verifications, Exo uses the resulting structure to generate the optimized C code — if no optimizations are applied, the expected generated C code for the above specification would result from simply replacing the specific Python syntax with C syntax.

PREPRINT

The loop-based structure of Exo makes it more flexible than Halide, allowing it to express imperfectly nested loops, which is Halide's bane. However, with flexibility comes complexity. Thus, the optimizations available in Exo are more limited compared to Halide, unless the user provides additional information about the computation semantics to help Exo perform more complex transformations (such as assertions about loop bounds to enable tiling).

2.3 Noarr

Noarr [18] is a C++ header-only library that provides abstractions for defining data layouts (structures) and algorithms (structure traversal). A Noarr implementation of matrix multiplication can be expressed as follows:

```
auto C = bag(scalar<float>() ^ vector<'i'>(M) ^ vector<'j'>(N));
auto A = bag(scalar<float>() ^ vector<'i'>(M) ^ vector<'k'>(K));
auto B = bag(scalar<float>() ^ vector<'k'>(K) ^ vector<'j'>(N));

traverser(C, A, B) | [](auto idx) { C[idx] += A[idx] * B[idx]; };
```

Among the selected frameworks, Noarr uses the most lightweight and straightforward approach. It is a C++ library that provides template-based abstractions for defining and iterating data structures; thus, it allows full usage of C++ features. The core concept in Noarr is defining data structures using composable abstractions (like `vector` or `scalar`) that describe the layout and dimensions of the data, and performing computations using *traversers* that iterate over the iteration space defined by the closure over their dimensions (sequential, OpenMP, TBB, and CUDA-based traversers are available at present). Noarr does not perform any semantic analysis of the computations, but imposes safety via type checking and dimension naming.

Since Noarr is a lightweight abstraction over C++, it is expected to behave similarly to hand-written C/C++ code for the same algorithm, given that the algorithm semantics are identical. Compared to Halide and Exo, Noarr is simpler in terms of setup and integration into existing C++ projects, but it lacks capabilities such as applying more complex transformations or performing verification. It also differs in that it abstracts indexation (e.g., `C[i, j]` in Exo or Halide is represented as `C[idx]` in Noarr, with no explicit indices). This, combined with dimensions of the structures and loop nests being named rather than positional, helps avoid certain classes of bugs related to incorrect indexing. It is thus expected that Noarr implementations will be less error-prone compared to analogs in Halide or Exo. However, since Noarr does not provide verification capabilities, logic errors in Noarr may be caught only at runtime rather than at compile time.

2.4 PolyBench

To test and evaluate our LLM-guided methods, we need to select and implement a set of coding problems that could benefit from HPC optimizations.

PREPRINT

The PolyBench [29] benchmark suite is a collection of 30 numerical computations commonly used in scientific computing and high-performance computing research. It includes a variety of algorithms from different domains, such as linear algebra, data mining, and stencil computations. All benchmarks are implemented in plain C, and the suite defines several problem sizes (`MINI`, `SMALL`, `MEDIUM`, `LARGE`, and `EXTRALARGE`). In total, there are 150 benchmark variants for a given data type (e.g., `float` or `double`). The PolyBench suite is widely used for evaluating the performance of compilers, optimization techniques, and hardware architectures [4,6,14,21,25,34], so we consider it a suitable choice for our experiments.

3 Methodology

In our experiments, we instruct the LLM to automatically optimize the performance of the given problem while preserving the original semantics. Doing so in pure C code is challenging even for human programmers. As illustrated in Section 2, even a relatively simple optimization, such as performing tiling across the input dimensions of a matrix multiplication, changing the layout of the involved arrays, and applying unrolling and vectorization, can considerably increase code complexity.

The PolyBench optimization starts from existing C implementations. For framework-based optimization, our workflow begins by asking the LLM to express the same computation in the target framework or DSL before applying optimization prompts. The LLM produces multiple candidate representations, and we algorithmically select the most suitable one (further described in Section 3.1). Automating this step also avoids introducing human-written optimizations that could bias the subsequent optimization stage.

Each code representation is subjected to two types of LLM-guided optimizations: *direct* code optimizations and optimizations based on an *abstract optimization plan*. In the direct optimizations, we try multiple prompts with different levels of detail. The abstract optimization plan approach is divided into two subsequent prompts: the first is to generate an abstract plan, and the second is to apply (follow) the generated plan to produce the optimized code. The idea is to help the LLM to better structure its thoughts before applying them to the code [19]. Furthermore, the abstract plan may be more readable to users, allowing for a better understanding of the optimizations applied by the LLM (mitigating the *comprehension debt* [35]).

A complete diagram of our methodology is shown in Figure 1. We address the individual parts in the following sections, while the technical details (including the prompts) are provided in the replication package [17].

3.1 Translation

The LLM is requested to translate the original C code (a benchmark code from the PolyBench suite) into the target framework (or DSL), while preserving the

PREPRINT

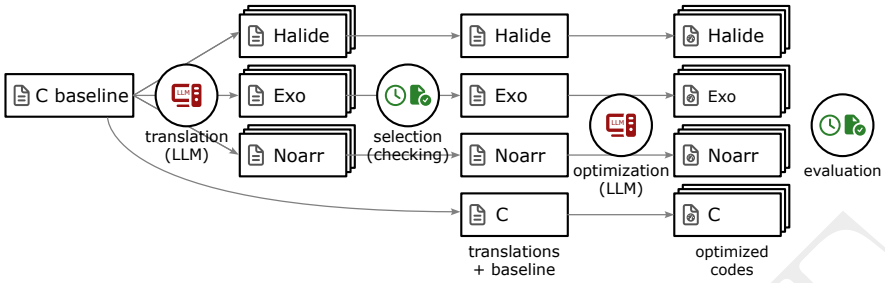

Fig. 1. Methodology overview

original behavior as closely as possible. This stage produces the framework representation that the rest of the workflow will optimize. Ideally, the generated code follows the same structure as the original and performs the same computations in the same order (i.e., having the same memory access patterns), but using the constructs provided by the target framework.

In each request, the LLM is given (in the system message) the programming guide of the target framework, which is short enough to fit within the LLM's context window and specific enough to give the LLM a good idea of how to use the framework. Since the guides do not include complete reference documentation, we also provide the LLM with an example of translating the `GEMM` benchmark from PolyBench into the target framework, along with a list of framework-specific rules that the LLM should follow when generating the code. The goal of these additional resources is to explain further how to use the target framework correctly and to help the LLM avoid common pitfalls. We tailor the rules to each framework individually, based on our prior experience, and evaluate their effectiveness and sufficiency on a small subset of problems before proceeding to the full evaluation. We have used a representative set of four problems to tune the translation prompts: `GEMM`, `2MM`, `Floyd-Warshall`, and `Heat-3D`; these prompt-development benchmarks are excluded from the reported evaluation results.

We make five independent translation requests for each framework and benchmark combination to account for the variability in the LLM's outputs. Each translation candidate is first validated as described in Section 3.4. Among the validated candidates, we select the one whose measured performance most closely matches the original C implementation across the `SMALL`, `MEDIUM`, and `LARGE` input sizes from PolyBench. Only this selected translation is used as the input to the subsequent framework-based optimization experiments, so the reported framework results include both the quality of this translation step and the LLM's ability to optimize the selected representation.

3.2 Direct optimization

The direct optimization approach is similar to the translation step. The LLM is given the current representation produced by the workflow: the original C imple-

PREPRINT

mentation for direct C optimization, or the selected translated representation for framework-based optimization. The prompt also includes the programming guide and specific rules of the respective framework when applicable. We make five independent optimization requests for each combination of benchmark, framework, and direct prompting strategy (including the C code).

The running system is briefly described in general terms as a modern x86-64 machine with a given number of CPU cores, along with the compiler version, the optimization flags, and the target framework specification. The LLM is then requested to optimize the code for performance while preserving its semantics. The prompts are tuned on the same representative set of four benchmarks as the translation prompts, as described in Section 3.1. The optimization goals are specified by prompting the LLM to improve data locality, computation efficiency, and memory access patterns, while preserving the original code semantics and the boilerplate code used to set up the benchmarks and measuring their execution time.

For direct optimization, we use three prompting strategies in our endeavor to answer RQ2:

Naïve approach The LLM is simply prompted to optimize the provided code for performance while preserving its semantics without additional guidance.

All-hints approach The LLM is provided with specific optimization hints (detailed below) to guide the optimization process.

Choose-hints approach Is based on the `all-hints` approach, but the LLM is given an additional instruction to choose the most suitable hints from the provided ones and then apply them to the code.

The optimization hints provided to the LLM are divided into four categories. During prompt development, we also tried these categories as separate exploratory pilot prompts, but these pilot variants are not part of the reported design. In the reported experiments, the categories appear only as the combined hint set used by the `all-hints`, `choose-hints`, and `from-plan` approaches.

Cache Directs the LLM to apply tiling to the loops and reorganize the data layouts of the involved arrays to improve cache locality.

Structure Directs the LLM to define temporary buffers of limited size (up to 50% of the original arrays) to store intermediate results.

Arithmetic Directs the LLM to consider applying algebraic transformations to reduce the number of arithmetic operations performed in the code. Also suggests using accumulator variables to minimize memory accesses within loops.

Parallelism Directs the LLM to identify parallelization opportunities in the code and use appropriate constructs from the target DSL (or framework) to parallelize the computations across multiple CPU cores.

Each of these hints further asserts that the LLM should ensure that the optimizations preserve the original code semantics.

Together with the `from-plan` approach described below, the reported optimization design therefore consists of four prompting approaches crossed with four code representations (C, Exo, Halide, and Noarr).

PREPRINT

3.3 Abstract optimization plan

To address RQ3, we also investigate an alternative approach in which the LLM is first prompted (multiple times) to describe an abstract optimization plan without modifying the code, and then, in a second request, to choose from multiple optimization plans and apply the selected one to the original code. This approach should force the LLM to externalize a plan and reason about the optimizations before applying them to the code, potentially leading to better-structured and more effective optimizations [21].

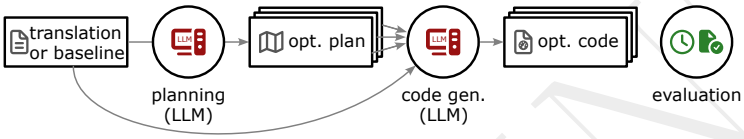

Fig. 2. Abstract plan optimization overview

Figure 2 illustrates the abstract optimization plan approach. The first request is based on the direct `all-hints` approach (the same four optimization hints are provided), but the model is prompted to describe an optimization plan. It differs in that the request contains the framework programming guide without the added framework-specific rules, and the LLM is instructed to describe the optimizations in abstract terms for later application. Similar to the translation step, the prompt includes an example of an abstract optimization plan for the `GEMM` benchmark with the expected format and level of detail.

The optimization plan is based on Halide and Exo principles — namely, separating the data structures, computation pipelines, and optimization schedules. The LLM is expected to describe (in separate sections) the index domain (dimensions), the involved data structures and their layouts, any further algorithm parameters, and the computational steps to be performed (via loop nests representing quantification, and conditional blocks), and any relevant data dependencies that might affect parallelization or vectorization. Then, it is expected to describe the optimizations to be applied to the computation in terms of transformations of the described dimensions, data structures, and computational steps, via descriptive directives similar to those in the studied DSLs (such as tiling, loop interchange, unrolling, vectorization, and parallelization). The LLM is allowed to define its own directives as needed to express new optimizations, which aligns with the extensibility of Exo and Noarr and the intent to allow the LLM to leverage its full capabilities.

The system message for the second request is based on the naïve direct optimization prompts, with additional instructions to select the best plan from the previous request and apply it to the current code representation, instead of giving general optimization instructions. Similarly to the first request, the expected output is showcased on the `GEMM` benchmark in the prompt (using the

PREPRINT

target framework). The LLM is then provided with a user message containing the current code representation and all the previously generated plans.

3.4 Validation

We validate the generated code by comparing its output with the output of the original unoptimized C code for the PolyBench-provided input sizes. We consider the generated code to pass validation if its output matches the original code's output within a small numerical tolerance (absolute tolerance 10^{-2} and relative tolerance 2×10^{-7}, chosen to account for precision errors) for all tested input sizes. If the generated code fails to compile or run for any input size, or produces mismatching output for any input size, it is considered invalid for this evaluation, and the specific failure category is recorded. We also impose an 8-minute execution time limit, which is well beyond the expected execution time of any benchmark.

While this validation approach does not guarantee full semantic equivalence suitable for production use, the risk of false positives is low for the purposes of this study, as the PolyBench benchmarks are intentionally designed with non-symmetric input data and dimension sizes non-divisible by common tiling factors, making it unlikely that semantically incorrect code would pass on all tested input sizes by chance.

3.5 Threats to validity

One of the most common threats to validity when dealing with LLM outputs is their inherent stochasticity. We mitigate this by making multiple independent requests and selecting the output that best matches the original code in the translation step.

A related threat is the LLM's sensitivity to prompt wording and structure, which we mitigate by aligning the structure of the prompts with the provider's prompting best practices [27] and by tuning the prompts on a representative set of benchmarks (Section 3.1) before proceeding to the full evaluation. We put great effort into minimizing unintentional bias in the prompts by constructing them in such a way that they do not favor any particular approach or framework and provide comparable levels of guidance and information across all approaches. Furthermore, we used the Grammarly tool to standardize the language quality and style across all prompts.

Finally, the LLM's familiarity with the benchmarks and their optimizations, as well as with the studied DSLs, may also threaten the validity of the results. We mitigate this by including multiple DSLs differing in design and usage, and by appending their respective programming guides, along with examples and LLM-specific rules, to the prompts. Based on the experiments used to tune the prompts (Section 3.1), we observed no positive correlation between LLM competence and the framework's popularity or age.

PREPRINT

4 Evaluation

The evaluation was performed on a dual-socket system with Intel Xeon Gold 6130 processors, each comprising 16 cores with 2-way hyper-threading (64 threads in total). All benchmarks use double precision and were compiled with GCC 15.2, using `-O3`, `-march=native`, and `-mtune=native` optimization flags. The Halide code was linked against Halide v21.0.0. The performance was measured using a wall-clock timer, like in the PolyBench benchmark suite [29]. In all cases, only the 26 benchmarks not used for prompt design are considered for evaluation (as explained in Section 3.1).

For the LLM requests, we used GPT-5.1 snapshot `gpt-5.1-2025-11-13`, reasoning effort set to `high` and verbosity set to `medium` [26]; request construction is described in Section 3.

For each generated implementation, we averaged the execution time over three measured runs after two warm-up runs, which kept the measurement variance low on our system.

Generated implementations are validated using the procedure described in Section 3.4. Invalid implementations are excluded from performance aggregates.

4.1 Translation results

Figure 3 compares the performance of the PolyBench benchmarks translated by LLM into Exo, Halide, and Noarr frameworks against the original C code without any optimizations applied. The x-axis represents the different benchmarks with horizontal lines separating each benchmark category as defined by the PolyBench suite. The y-axis shows the relative speedup of each translation over the baseline (emphasized by a line at $y = 1$). The y dimension of the displayed points shows the aggregated speedup over all input sizes defined by PolyBench for each benchmark, using the geometric mean, following standard practice [21]. A lower distance from the baseline indicates better translation in terms of fidelity to the original benchmark.

The figure shows all translation attempts before selection. For the optimization experiments, each benchmark/framework pair contributes only the validated translation selected by the procedure in Section 3.1: the candidate closest to C performance on the `SMALL`, `MEDIUM`, and `LARGE` input sizes.

Most translations performing substantially worse than the C baseline belong to the Halide framework. In this workflow, this is mostly due to the LLM-generated Halide representations of cyclic data flows. Intuitive workarounds (e.g., running the Halide pipeline in a loop) often result in inefficient code that the LLM does not optimize until prompted to do so.

The higher variance in Noarr translations can be mostly attributed to Noarr defining array dimensions in reverse order compared to C; sometimes, the LLM prioritizes preserving the syntactic order of dimensions over semantic order, leading to accidental improvements or degradations in memory access patterns.

The LLM consistently fails to successfully translate the `cholesky` and `nussinov` benchmarks to Noarr and Halide due to more complex output formatting re-

PREPRINT

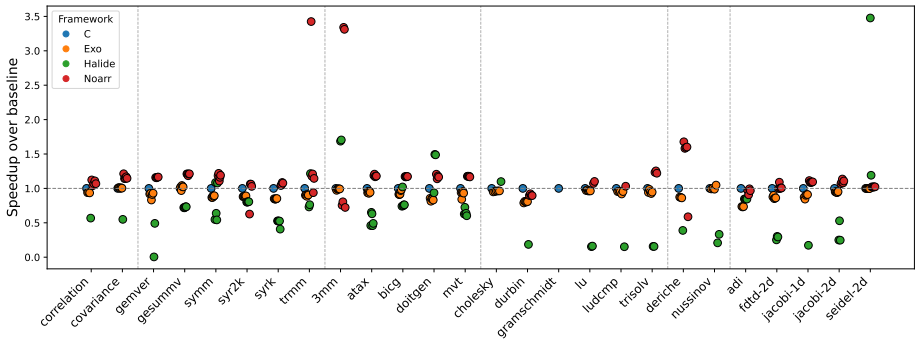

Fig. 3. Speedup of PolyBench benchmark translations to selected frameworks over the original C code. Each point represents one LLM translation of a given benchmark.

quirements, which the LLM fails to meet. The original benchmark outputs only a specific portion of the output array, which is overlooked by the LLM. For the `gramschmidt` benchmark specifically, all LLM implementations fail due to numerical instability and arithmetic errors with all three frameworks.

The LLM achieves the highest success rate with Exo (which represents the most straightforward translation), failing only 12 times out of 130 attempts (including compile-time and runtime errors, and validation failures). Noarr fails in 30 and Halide in 61 attempts (of 130); however, a higher failure rate is expected for Halide in this workflow, as translating certain computations requires expressing them as a chain of function updates or numerous intermediate functions, which can easily lead to code that may pass validation, but is extremely inefficient. Overall, the results roughly correspond to the complexity of transcribing the source computations to each target framework.

4.2 Optimization results

The upper row of Figure 4 shows the geometric mean of the best speedup for a given prompting approach and framework pair for the `EXTRALARGE` input size. The baseline is the corresponding unoptimized C implementation. The aggregation covers the 26 benchmarks not used for prompt design; for each benchmark, invalid generated implementations are ignored for speedup selection, and the C baseline is used as a fallback with speedup 1.0 if no valid implementation performs better. The x-axis represents the budget k of independent attempts, while the y-axis shows the geometric mean of the best speedup achieved within that budget. The lower row shows the corresponding mean probability of obtaining at least one valid optimization within the same budget, so invalid attempts remain in the validity denominator.

We chose `EXTRALARGE` as it models a case where performance typically matters the most. Plots for the other sizes and more detailed data (e.g., individual measurements and validation results) are available in our replication package [17].

PREPRINT

The `all-hints` approach applied to direct C code generation performs the best overall performance, and the `choose-hints` approach performs only slightly worse in terms of performance (for Noarr, the performance degradation is more pronounced, suggesting optimization is less intuitive in this framework). We observed no indication that the LLM would pursue unsuitable optimization paths more frequently for `all-hints` compared to the `choose-hints` approach, despite the former being more aggressive in its optimization requests.

Contrary to initial expectations, the `from-plan` approach does not appear to provide clear performance benefits over the other two approaches, or even over the naïve approach. Moreover, the `from-plan` approach tends to actively hinder the performance compared to `all-hints` it is derived from. This suggests that the LLMs become more proficient in their own reasoning processes, and the users should not coerce them into deep reasoning artificially.

The validity rate is mostly independent of the prompting approach, with the exception of the `from-plan` approach, where the validity rate for the Exo framework is lower than for the other approaches. Direct C has the highest validity rate, independent of approach. For the Noarr framework, the LLM performs consistently slightly worse regardless of the given budget. At five attempts, the Exo framework becomes competitive with the C baseline in terms of validity rate, which suggests Exo code generation failures are more random than systematic in this setting (contrasting Noarr).

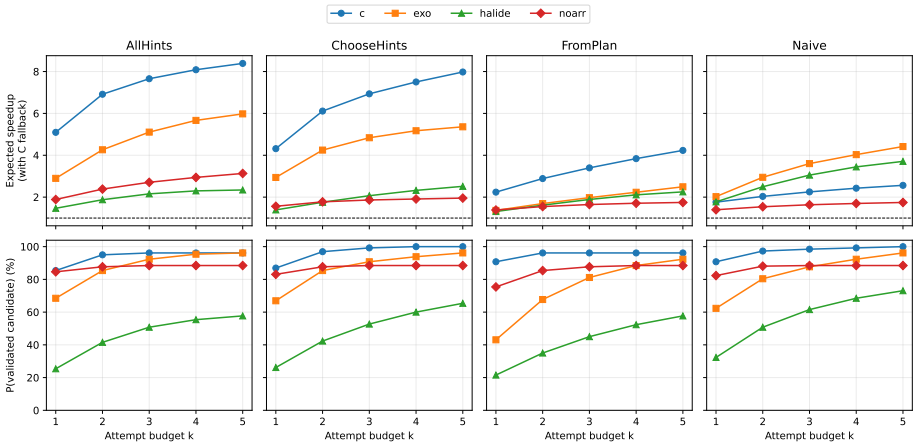

Fig. 4. Geometric mean of the best speedups and mean probability of at least one valid optimization attempt for a given prompting approach and framework over the C baseline for the `EXTRALARGE` input sizes of the 26 benchmarks. The x-axis represents the budget k of independent attempts. Invalid attempts are counted in the validity denominator; speedups use the best valid candidate within the budget, with the C baseline available as a fallback.

All prompting approaches show that the expected geometric mean speedup increases well with the number of independent attempts, with the exception of the Noarr framework. Disregarding Noarr, the budget of 5 independent attempts seems to be still in the steep part of the curve, suggesting little diminishing returns for the best-of-five approach. Achieving a better speedup is possible with more attempts at the cost of further diminishing returns.

Inspection of the outputs suggests that the LLM uses OpenMP parallelization for C and Noarr implementations and the built-in parallelization features of the Exo and Halide frameworks across all approaches. The observed tendency to apply parallelization is highest for the Halide framework, followed by C. This tendency is also influenced by the specificity of the optimization hints — highest for the `all-hints` approach, followed by `choose-hints` and lowest for the `from-plan` approach. This suggests a higher cautiousness in the `from-plan` approach, which, however, is not reflected in the validity rates.

Across the 2,080 optimization attempts, 822 attempts were invalid. The failure-category tags record 454 compilation errors, 276 runtime errors, 91 numerical mismatches (invalid results), and 10 timeouts; these tags are diagnostic rather than disjoint, since one invalid attempt can fail differently across input sizes. Numerical mismatches and timeouts together contribute to only 5% failure rate, making them a minor source of invalid optimizations. Runtime errors were almost entirely concentrated in Halide (273 of 276), as it includes errors from its inner compilation pipeline. Compilation errors were the most common category overall. By framework, compilation-error tags were least frequent for C (33), followed by Noarr (65), Halide (167), and Exo (189).

4.3 Comparison to prior work

We compare our best-performing approach (`all-hints` applied to direct C code generation) to prior work on LLM-guided optimizations, namely the approach of Merouani et al. [21], the most recent study with similar goals. They employ an iterative agentic framework to refine optimizations based on feedback from compiler and runtime measurements, and reported performance improvements of 200× or above on `2MM`, `3MM`, `correlation`, `covariance`, `syr2k`, `syrk`, and `trmm`, with lower improvements on other PolyBench benchmarks at `EXTRALARGE` input sizes. The work used a custom Tiramisu re-implementation for both the unoptimized baseline and the optimization target. Their best-of-five setting also uses 30 optimization iterations per run, for up to 150 explored schedules per benchmark instance.

Our results show that even with a non-iterative approach using only five LLM requests per benchmark and selecting the best optimization among them, it is possible to achieve results of a similar order of magnitude on several of these benchmarks. Against the original package-measured Tiramisu baseline, our best direct-C `all-hints` runs reach 103.27× on `3MM`, 47.42× on `correlation`, and 73.11× on `covariance`. Furthermore, we observe high speedups on some benchmarks where their approach performed poorly, such as `doitgen` (166.42×) and `symm` (53.68×).

PREPRINT

While their work demonstrates the potential of iterative refinement for LLM-guided optimizations, our findings indicate that a more direct approach (using pure C without iterative prompting) achieves strong performance in our setting.

We also observe much higher validity rates in our results compared to their work. While this could be attributed to differences in the strength of the LLM models used, we also note a success rate on Exo and Halide translations that is quite comparable to the validation rates observed by Merouani et al. [21], whose approach uses an abstraction related to Halide. This suggests that the incidence of invalid optimizations may also be attributed to the characteristics of the abstractions used.

4.4 Answering research questions

RQ1*: Do frameworks help?* Section 4.2 shows that using the frameworks in the context of our LLM-guided optimization workflow on the PolyBench suite does not provide clear performance or validity benefits over direct C code generation. It should be noted, however, that identifying an invalid optimization in a specifically designed framework can be easier than in C code due to the involved validation mechanisms, as shown by Merouani et al. [21]. Conversely, these frameworks often impose strict rules that can limit the optimization opportunities the LLM can explore compared to direct C code generation, as we identified in Section 4.3.

RQ2*: Do specific hints help?* Section 4.2 indicates that providing more specific optimization goals to the LLM generally leads to better performance of the optimized code compared to naïve optimization requests. This trend is consistent across both framework-based and direct C code generation approaches.

RQ3*: Does the abstract plan help?* The goal of this research question was to determine if a two-step optimization process (first, generating an abstract optimization plan, then implementing it) improves optimization quality. We hypothesized that requesting the LLM to first propose multiple abstract strategies and subsequently select the most promising one to implement would reduce the likelihood of pursuing invalid or suboptimal paths.

Our results show that introducing the plan externalization phase does not generally improve performance compared to optimizing the code directly. Further investigation needed to fully analyze this effect is out of the scope of this work, but the recorded token accounting indicates that the LLM adapts its reasoning depth based on the given benchmark, which provides a more flexible approach than the tested plan externalization.

RQ4*: Comparison of all approaches.* From the results obtained for RQ1–RQ3, the specificity of optimization goals (RQ2) appears to have the greatest positive impact on the performance of LLM-guided loop optimizations. Specifically, the `all-hints` approach yields the best performance across all frameworks, with a minor exception for Halide.

PREPRINT

Contemporary frameworks and DSLs do not clearly improve validity rates compared to direct C code generation in this workflow; however, the Exo framework comes close to matching the validity rate of direct C code generation after 5 independent attempts, regardless of the prompting approach.

RQ5*: Are LLMs constrained by existing optimization frameworks?* The results discussed in Section 4.3 indicate that LLMs can indeed reliably produce optimizations that are not easily expressible in optimization directives available in existing frameworks. This suggests that future research should focus on developing abstractions better aligned with the capabilities of LLMs, achieving verifiability without relying on prefabricated optimization templates.

5 Related Work

Automated code optimization has historically relied on analytical frameworks, most notably the polyhedral model. Tools such as Pluto [6] and Polly [14] leverage mathematical abstractions to perform aggressive loop reorganizations, including tiling, skewing, and interchange. While effective, these techniques are constrained by strict static analysis requirements, limiting their ability to apply more complex transformations.

For greater flexibility, domain-specific languages (DSLs) like Halide [30], Exo [15], Noarr [18], and Tiramisu [5] decouple algorithms from execution schedules. This separation facilitates auto-tuning, where schedulers search for optimal configurations using beam search or learned cost models [2,22]. However, these approaches are often computationally demanding, especially when targeting diverse hardware backends.

The advent of Large Language Models (LLMs) has transformed code generation [16,31]. Despite their success in general programming, empirical studies indicate that off-the-shelf models struggle to produce efficient parallel code for high-performance computing (HPC) tasks [13,23,33]. To bridge this gap, researchers have explored various enhancement strategies.

One direction is fine-tuning of specialized models. Examples include HPC-Coder [24], which targets parallel code generation, and the LM4HPC framework [9,10] for OpenMP pragmas. Beyond standard fine-tuning, reinforcement learning (RL) has been applied to discover novel algorithms (AlphaDev [20]) or to align models with performance feedback (RLPF [25]). However, these methods are resource-intensive, require access to proprietary weights, and risk obsolescence as foundation models evolve.

Another direction uses prompt engineering. Studies show that expert-crafted hints, source-to-source directives, and zero-shot reasoning strategies can significantly improve performance without retraining [7,8,19,28]. This approach relies on user expertise to formulate effective prompts.

To automate this process, agentic frameworks employ iterative feedback loops using compiler or runtime signals to refine outputs [12,32]. A prominent example is the work by Merouani et al. [21], which integrates LLMs with Tiramisu to

PREPRINT

generate verifiable schedules. While robust, such agentic approaches could be very costly, often requiring dozens of compilation cycles (and prompts) per task.

We leverage generalized optimization hints and generated optimization plans to guide the LLM, achieving competitive speedups via cost-efficient, non-iterative prompting. Compared to prior work, our approach focuses on investigating multiple DSLs and frameworks, as well as direct C code generation, and different prompting strategies to identify the most effective methods for LLM-guided code optimization.

6 Conclusion

In this work, we present a study on using a large language model to optimize code for high-performance parallel applications in conjunction with frameworks that aim to ease the optimization process for both human programmers and automated tuning systems. In our evaluated setting, direct C generation produced stronger measured results than using these frameworks, both in terms of performance and correctness. We also observed that, for the evaluated workflow, the specificity of the prompt plays a larger role than whether the LLM is tasked to optimize code directly in C/C++ or through tools for user-guided optimization, such as Exo, Halide, or Noarr.

Two important implications arise. First, it suggests that prompt engineering is a critical factor in the success of LLM-guided code optimization. Second, it indicates that future research should focus on tailoring DSLs and alternative approaches to better align with LLM capabilities, potentially leading to new tools for code optimization that leverage the strengths of both approaches.

Acknowledgments. This paper was supported by Charles University institutional funding SVV (grant 260821) and by the Johannes Amos Comenius Programme (P JAC, Natural and anthropogenic georisks) project CZ.02.01.01/00/22_008/0004605.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Abdulla, M.F.: Manual and fast C code optimization. arXiv preprint arXiv:1203.0681 (2012)
2. Adams, A., Ma, K., Anderson, L., Baghdadi, R., Li, T.M., Gharbi, M., Steiner, B., Johnson, S., Fatahalian, K., Durand, F., Ragan-Kelley, J.: Learning to optimize Halide with tree search and random programs. ACM Transactions on Graphics **38**(4), 121 (2019)
3. Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U.M., Amarasinghe, S.: OpenTuner: An extensible framework for program autotuning. In: Proceedings of the 23rd international conference on Parallel architectures and compilation. pp. 303–316 (2014)

PREPRINT

4. Ashouri, A.H., Mariani, G., Palermo, G., Park, E., Cavazos, J., Silvano, C.: COBAYN: Compiler autotuning framework using bayesian networks. ACM Transactions on Architecture and Code Optimization **13**(2), 21 (2016)
5. Baghdadi, R., Ray, J., Romdhane, M.B., Del Sozzo, E., Akkas, A., Zhang, Y., Suriana, P., Kamil, S., Amarasinghe, S.: Tiramisu: A polyhedral compiler for expressing fast and portable code. In: IEEE/ACM International Symposium on Code Generation and Optimization. pp. 193–205 (2019)
6. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 101–113 (2008)
7. Brabec, M., Klepl, J., Töpfer, M., Kruliš, M.: Tutoring LLM into a better CUDA optimizer. In: European Conference on Parallel Processing. pp. 250–263 (2025)
8. Chen, B., Mustakin, N., Hoang, A., Fuad, S., Wong, D.: VSCuda: LLM based CUDA extension for Visual Studio Code. In: Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis. pp. 11–17 (2023)
9. Chen, L., Bhattacharjee, A., Ahmed, N., Hasabnis, N., Oren, G., Vo, V., Jannesari, A.: OMPGPT: A generative pre-trained transformer model for OpenMP. In: European Conference on Parallel Processing. pp. 121–134 (2024)
10. Chen, L., Lin, P.H., Vanderbruggen, T., Liao, C., Emani, M., De Supinski, B.: LM4HPC: Towards effective language model application in high-performance computing. In: International Workshop on OpenMP. pp. 18–33 (2023)
11. 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)
12. Chen, X., Lin, M., Schärli, N., Zhou, D.: Teaching large language models to self-debug. In: 12th International Conference on Learning Representations (2024)
13. Godoy, W.F., Valero-Lara, P., Teranishi, K., Balaprakash, P., Vetter, J.S.: Large language model evaluation for high-performance computing software development. Concurrency and Computation: Practice and Experience **36**(26), e8269 (2024)
14. Grosser, T., Groesslinger, A., Lengauer, C.: Polly—performing polyhedral optimizations on a low-level intermediate representation. Parallel Processing Letters **22**(04), 1250010 (2012)
15. Ikarashi, Y., Bernstein, G.L., Reinking, A., Genc, H., Ragan-Kelley, J.: Exocompilation for productive programming of hardware accelerators. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 703–718 (2022)
16. Jiang, J., Wang, F., Shen, J., Kim, S., Kim, S.: A survey on large language models for code generation. ACM Transactions on Software Engineering and Methodology **35**(2), 58 (2026)
17. Klepl, J., Brabec, M., Kruliš, M.: Replication package repository (2026), https://github.com/jiriklepl/llm_code_optimization.git
18. Klepl, J., Šmelko, A., Rozsypal, L., Kruliš, M.: Abstractions for C++ code optimizations in parallel high-performance applications. Parallel Computing p. 103096 (2024)
19. Kojima, T., Gu, S.S., Reid, M., Matsuo, Y., Iwasawa, Y.: Large language models are zero-shot reasoners. Advances in Neural Information Processing Systems **35**, 22199–22213 (2022)

PREPRINT

20. Mankowitz, D.J., Michi, A., Zhernov, A., Gelmi, M., Selvi, M., Paduraru, C., Leurent, E., Iqbal, S., Lespiau, J.B., Ahern, A., et al.: Faster sorting algorithms discovered using deep reinforcement learning. Nature **618**(7964), 257–263 (2023)
21. Merouani, M., Bernou, I.K., Baghdadi, R.: Agentic auto-scheduling: An experimental study of LLM-guided loop optimization. In: 34th International Conference on Parallel Architectures and Compilation Techniques. pp. 186–200 (2025)
22. Mullapudi, R.T., Adams, A., Sharlet, D., Ragan-Kelley, J., Fatahalian, K.: Automatically scheduling Halide image processing pipelines. ACM Transactions on Graphics **35**(4), 83 (2016)
23. 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)
24. Nichols, D., Marathe, A., Menon, H., Gamblin, T., Bhatele, A.: HPC-Coder: Modeling parallel programs using large language models. In: Proceedings of the 39th International Conference on ISC High Performance. pp. 1–12 (2024)
25. Nichols, D., Polasam, P., Menon, H., Marathe, A., Gamblin, T., Bhatele, A.: Performance-aligned LLMs for generating fast code. arXiv preprint arXiv:2404.18864 (2024)
26. OpenAI: GPT-5.1: A smarter, more conversational ChatGPT (2025), https://openai.com/index/gpt-5-1/
27. OpenAI: Prompt engineering (2026), https://developers.openai.com/api/docs/guides/prompt-engineering
28. Palkowski, M., Gruzewski, M.: GPT-driven source-to-source transformation for generating compilable parallel CUDA code for Nussinov's algorithm. Electronics **13**(3), 488 (2024)
29. Pouchet, L.N., Yuki, T.: PolyBench/C 4.2.1 (beta) (2016), https://sourceforge.net/projects/polybench/files/polybench-c-4.2.1-beta.tar.gz/download
30. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.: Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. ACM SIGPLAN Notices **48**(6), 519–530 (2013)
31. Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X.E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., et al.: Code Llama: Open foundation models for code. arXiv preprint arXiv:2308.12950 (2023)
32. Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., Yao, S.: Reflexion: Language agents with verbal reinforcement learning. Advances in Neural Information Processing Systems **36**, 8634–8652 (2023)
33. Valero-Lara, P., Huante, A., Al Lail, M., Godoy, W.F., Teranishi, K., Balaprakash, P., Vetter, J.S.: Comparing Llama-2 and GPT-3 LLMs for HPC kernels generation. In: International Workshop on Languages and Compilers for Parallel Computing. pp. 20–32 (2023)
34. Verdoolaege, S., Juega, J.C., Cohen, A., Gomez, J.I., Tenllado, C., Catthoor, F.: Polyhedral parallel code generation for CUDA. ACM Transactions on Architecture and Code Optimization **9**(4), 54 (2013)
35. Zhang, Y.: Beyond technical debt: How AI-assisted development creates comprehension debt in resource-constrained indie teams. arXiv preprint arXiv:2512.08942 (2025)

PREPRINT