

Compiler.next: A Search-Based Compiler to Power the AI-Native Future of Software Engineering

FILIPE R. COGO, Centre for Software Excellence - Huawei Canada, Canada

GUSTAVO A. OLIVA, Centre for Software Excellence - Huawei Canada, Canada

AHMED E. HASSAN, School of Computing, Queen's University, Canada

The rapid advancement of AI-assisted software engineering has brought transformative potential to the field of software engineering, but existing tools and paradigms remain limited by cognitive overload, inefficient tool integration, and the narrow capabilities of AI copilots. In response, we propose *Compiler.next*, a novel search-based compiler designed to enable the seamless evolution of AI-native software systems as part of the emerging Software Engineering 3.0 era. Unlike traditional static compilers, *Compiler.next* takes human-written intents and automatically generates working software by searching for an optimal solution. This process involves dynamic optimization of cognitive architectures and their constituents (e.g., prompts, foundation model configurations, and system parameters) while finding the optimal trade-off between several objectives, such as accuracy, cost, and latency. This paper outlines the architecture of *Compiler.next* and positions it as a cornerstone in democratizing software development by lowering the technical barrier for non-experts, enabling scalable, adaptable, and reliable AI-powered software. We present a roadmap to address the core challenges in intent compilation, including developing quality programming constructs, effective search heuristics, reproducibility, and interoperability between compilers. Our vision lays the groundwork for fully automated, search-driven software development, fostering faster innovation and more efficient AI-driven systems.

CCS Concepts: • Software and its engineering → Software notations and tools.

Additional Key Words and Phrases: Search-based software engineering, search-based compilation, automation, code synthesis, foundation models, FMware, prompt engineering, cognitive architecture

ACM Reference Format:

Filipe R. Cogo, Gustavo A. Oliva, and Ahmed E. Hassan. 2025. Compiler.next: A Search-Based Compiler to Power the AI-Native Future of Software Engineering. 1, 1 (October 2025), 31 pages. <https://doi.org/XXXXXX.XXXXXXX>

1 Introduction

The rapid rise of artificial intelligence (AI)-assisted software engineering (SE) has significantly transformed the development process, but it has also revealed critical limitations. Developers may face cognitive overload [1, 2], inefficiencies in tool integration [3, 4], and the narrow capabilities of AI copilots [5–7]. To overcome these challenges, we have previously proposed a vision of an AI-native future as a representation of a paradigm shift towards Software Engineering 3.0 (SE 3.0), where human developers and AI collaborate seamlessly and their strengths are combined to realize SE tasks [8]. SE 3.0 redefines software development by moving beyond task-based AI assistance towards a paradigm where

Authors' Contact Information: Filipe R. Cogo, filipe.cogo@acm.org, Centre for Software Excellence - Huawei Canada, Kingston, ON, Canada; Gustavo A. Oliva, gustavo.oliva@huawei.com, Centre for Software Excellence - Huawei Canada, Kingston, ON, Canada; Ahmed E. Hassan, ahmed@cs.queensu.ca, School of Computing, Queen's University, Kingston, ON, Canada.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.

Manuscript submitted to ACM

AI systems become intelligent collaborators that are capable of **autonomously synthesizing human intents into working software** (e.g., an idea into a mobile app, or functional requirements into a system that fulfills a customer’s need). We refer to the component responsible for performing this synthesis as *Compiler.next*. Autonomous coding agents, despite its current limitations, are now actively initiating, reviewing, and evolving code at scale throughout open-source ecosystems. Recent empirical work provides concrete evidence that AI teammates are already integrated into modern software development workflows [9]. Our core contribution in this paper is to provide a systematic framework and research roadmap for this emerging paradigm, particularly the compilation infrastructure needed to transform intents into optimized FMware, positioning the scattered advances in agent-based development within a coherent vision of AI-native software engineering.

Due to the reasoning and generative capabilities of Foundation Models (FMs) like Large Language Models (LLMs), FM-powered systems (FMware) will grow even more in popularity and become a fundamental type of software in the SE 3.0 era. However, synthesizing intents into FMware poses several challenges. FMware encompasses an entire ecosystem of interdependent components that interact dynamically in real time. This includes retrieval-augmented generation (RAG) to fetch relevant information, continuous data flows powered by a data flywheel approach that provides up-to-date context, and a runtime environment where models are constantly evolving in response to changing inputs, feedback loops, and tasks. As these systems evolve, they require an orchestrated effort to manage these components, ensuring that the system’s performance remains optimal despite the constantly shifting landscape. Therefore, synthesizing FMware requires more than optimizing prompts with prompt engineering techniques. It requires a whole new, dynamic, integrated approach to generate every component of a complex cognitive architecture (CA) while ensuring that those components are aligned and can evolve in harmony.

To synthesize intents into complex FMware, *Compiler.next* employs a search-based approach. By leveraging FMs, the solution is iteratively developed at hyper speed through code mutations and self-reflection mechanisms that iteratively evaluate how well the resulting software matches the intent. Another key property of *Compiler.next* is its ability to find the best trade-off between various competing objectives, such as accuracy (how well it fulfills the intent), latency (e.g., number of API requests sent to one or more FMs), and cost (e.g., number of input tokens in the prompt). That is, *Compiler.next* performs a multi-objective optimization.

Ultimately, *Compiler.next* is the realization that FMs serve as probabilistic CPUs and that prompts act as the “binaries” that are executed by these FMs. By allowing developers to focus on intents (instead of creating and hacking prompts), *Compiler.next* powers and accelerates the development of FMware in the SE 3.0 era by **searching** for the optional CA (e.g., prompts, FM selection and configuration, and retrieval-augmented generation (RAG) components) to fulfill the human’s intents while ensuring that multiple criteria are satisfied with the best trade-off possible.

In this paper, we lay out a *research and development roadmap* for *Compiler.next*. This roadmap results in 10 *call for action* items. We focus specifically on the compilation challenge, i.e. transforming human intents into optimized FMware through search-based synthesis, as this process directly addresses critical software maintenance challenges. For example, recent research demonstrates that prompts are highly fragile and sensitive to minor variations, with performance varying significantly based on formatting choices [10], instruction phrasing [11], and even the order of few-shot examples [12]. This fragility creates a substantial maintenance burden as FMs evolve and requirements change. By compiling from stable intents, *Compiler.next* enables FMware to adapt to new models and requirements through recompilation, similar to how traditional software can be recompiled for new target architectures. This separation of intent (what to achieve) from implementation (optimized prompts and configurations) is fundamental to sustainable FMware engineering. Therefore, intent compilation represents a foundational capability that enables many downstream

SE activities, but it is not the entirety of software engineering in the SE 3.0 era. A complete SE 3.0 ecosystem requires complementary advances in requirements validation, comprehensive testing strategies, FMware deployment and operations, maintenance and evolution processes, and collaborative development workflows. We position *Compiler.next* as an enabling technology that must integrate with these broader practices to support complete software development lifecycles. We hope that our set of calls for action will inspire the software engineering community and foster deeper academia-industry discussions and collaborations.

Calls for action

1. Establish quality programming constructs for representing FMware programs.
2. Consolidate the forms of compilation that involves not only prompt templates but all free parameters of an FMware, including better support for agent-based applications [13].
3. Identify the set of most effective heuristics to search the space of FMware parameters.
4. Construct sets of gold labels to evaluate candidate solutions and guide the search procedure during compilation.
5. Assure the quality of an FMware application, failing compilation when quality thresholds are not met.
6. Reduce the cost and improve the efficiency of compilers.
7. Make intent compilation reproducible.
8. Enable user-defined, multiple concurrent objectives to be optimized during compilation.
9. Improve the interoperability between compilers.
10. Build community-sharing platforms of compilation traces such that this information can be used as a feedback signal to improve compilation.

This paper is structured as follows. Section 2 discusses background and related work. Section 4 describes *Compiler.next*, which reflects our vision of a compiler that synthesizes intents into running software via a search-based approach. Section 5 presents a research and development roadmap for *Compiler.next*, which includes the 10 “calls for action” described above. Finally, Section 6 states our final remarks and concludes the paper.

2 Background and related work

Compilers are fundamental to SE as they transform human-readable code into machine-executable instructions, bridging the gap between developers and computing infrastructure and allowing developers to focus on application logic rather than low-level optimizations. In this section, we describe the role and contributions of traditional (Section 2.1), deep learning (DL) (Section 2.2), and prompt (Section 2.3) compilers to SE.

2.1 Traditional compilers

Traditional compilers have played a pivotal role in SE by transforming high-level human-readable code into machine-executable binaries. The primary function of a traditional compiler is to map the abstract syntax of programming languages (PLs) into the concrete syntax of a target machine, optimizing code at various stages, such as lexical analysis, parsing, and code generation. Over decades, compilers have evolved to include sophisticated optimization techniques like constant folding, loop unrolling, and register allocation, aimed at maximizing performance while minimizing resource utilization [14]. Despite their enduring importance, traditional compilers operate in a static context: once the code is compiled and optimized, there is no continuous refinement based on runtime feedback.

The rigid architecture of traditional compilers limits their applicability in dynamic, AI-powered environments, where systems need to adapt to evolving requirements in real-time. The rise of AI-native systems, particularly those powered by FMs, presents challenges that traditional compilers are ill-equipped to handle. Static compilers, such as GCC [15] or LLVM [16], are designed with deterministic and well-defined source code in mind, whereas AI-driven software demands continuous optimization and adaptation based on probabilistic reasoning, user feedback, and real-world data streams. This gap in adaptability is one of the core reasons why a new generation of compilers, such as *Compiler.next*, is required to handle the fluid and iterative nature of modern AI-powered systems.

2.2 Deep learning compilers

DL compilers have emerged as a response to the need for specialized optimization techniques that are tailored to the unique characteristics of DL models. These compilers, such as TVM [17], XLA [18], and Glow [19], focus on optimizing tensor operations, data flow, and memory usage to accelerate model inference and training. DL compilers take advantage of hardware-specific instructions, including GPUs, TPUs, and other accelerators, to maximize throughput and reduce latency. Unlike traditional compilers, which focus on optimizing symbolic logic, DL compilers are designed to optimize the execution of computational graphs [20], handling the complexity of parallelization, operator fusion, and data movement with high efficiency.

However, despite their significant contributions to performance optimization in DL workloads, DL compilers are still relatively limited in scope when it comes to broader AI-native software systems. They are primarily designed for the optimization of pre-defined neural network architectures, which is inadequate for the continuous, real-time optimization required in dynamic FMware systems. Moreover, DL compilers typically lack integration with higher-level abstractions like prompt engineering, reasoning modules, and multi-agent architectures that are integral to AI-native systems. *Compiler.next* extends beyond these limitations by incorporating search-based techniques to optimize not only the deep learning components but the entire FMware stack, including prompts, CAs, and agent-based frameworks, enabling real-time adaptation and evolution of AI-native software.

2.3 Prompt compilers

Automatic Prompt Engineer (APE) [21] frames the problem of automatically generating and selecting prompts as a “natural language program synthesis”, drawing parallels between prompt engineering and traditional programming. APE uses an FM to generate candidate prompt solutions, and a set of input-output demonstrations $\mathcal{D}_{train} = (Q, A)$ for solution evaluation (a.k.a. gold labels, see Section 4.1). APE evaluates each candidate solution ρ by prompting the FM with the concatenation of ρ and Q and comparing the generated result with A (see error estimator in Section 4.1). APE can also be configured to run an Iterative Monte Carlo Search algorithm by applying a paraphrasing prompt (see heuristic approximator in Section 4.1) to the candidate solutions and filtering out candidates with low scores.

Promptbreeder [22] uses an FM to drive a “self-improvement” search process that iteratively mutates a set of candidate task-prompts and evaluates the fitness of the candidate task-prompts based on a “training set” (or gold labels). One of the main innovations behind Promptbreeder is the idea of using “self-referential” mutation-prompts that are used to instruct the FM to perform mutations over the task-prompts and also undergo mutation themselves. In a similar vein, EvoPrompt [23] uses evolutionary algorithms to optimize a population of prompts by iteratively applying evolutionary operators (i.e., crossover and mutation) followed by the evaluation and selection of best-fit prompts to generate new offspring.

ProTeGi [24] simulates a “gradient-descent” approach to optimize prompt templates. In a “forward” step, it uses a “mini-batch” of input data and a reflection prompt to generate “gradients”, i.e., a summary, in natural language, of the associated “error” with the prompt under optimization for each of the instances of the mini-batch. Afterwards, on the “backpropagation” step, a delta-prompt is used to edit the prompt under optimization towards the direction of the “gradients” (i.e., by observing the “error” summary generated in the “forward” step). A beam search is then used to search over the space of candidate prompts.

SAMMO [25] represents FMware programs as “symbolic prompt programs”, which are direct acyclic graphs (DAGs) with each node indicating an arbitrary function and an edge indicating a function call. SAMMO extends the search for optimal FMware program configurations beyond prompt templates to include parameters of Promptware components. While optimizing the FMware program, SAMMO uses metaprogramming to mutate the associated DAG (e.g., to change the format of a prompt template or remove a node from the computation graph). Like the other approaches, SAMMO uses labelled samples to evaluate the candidate solutions during the search procedure. Search in SAMMO can be either enumerative, where the search space is explicitly defined (e.g., for parameters of Promptware components), or iterative, where the search space is described implicitly by an initial state and a set of mutation operators (e.g., for prompt templates) The mutation operators offered by SAMMO can modify the prompt template text or the prompt template structure.

3 Synthesizing Intents into FMware

Synthesizing human intents into fully functional FMware involves more than just generating prompts. It requires integrating various FMware components within a cohesive CA. In the following, we introduce the concepts of FMware (Section 3.1) and CAs (Section 3.2). Next, we discuss the role of compilers in the SE 3.0 era (Section 3.3) and the challenges of synthesizing intents into FMware (Section 3.4). Finally, we discuss the state of the practice (Section 3.5) in FMware compilation.

3.1 FMware and related concepts

FMware is a software system that uses FMs like LLMs as fundamental building blocks [26]. An FMware is further subdivided into two other categories: Promptware and Agentware. A Promptware is characterized by using prompts (typically written in natural language) to interact with the FMs, whereas Agentware is characterized by the usage of FMs to implement autonomous agents [27, 28] that are capable of decision-making and interaction with the environment or with other agents. An *FMware program* is composed of one or more *FMware modules*, with each module sharing dependency relationships with Codeware (the software paradigm driven by logic-based source code), Neuralware (the software paradigm driven by feature-based and deep learning models), and other FMware modules. Each FMware module comprises one or more *FMware components*, and each FMware component comprises smaller, self-contained Promptware and Agentware components. Figure 1 depicts an FMware module composed of a single FMware component, which, in turn, comprises a Promptware and an Agentware component.

3.2 FMware and cognitive architectures

A CA defines the coordination mechanism that dictates *what* software systems (possibly from different paradigms) will be invoked, *how*, and *in which order*. As we discuss in our prior work [26], the choice of coordination mechanism is what differentiates Promptware from Agentware. In Promptware, coordination follows a static *workflow* (or simply, *flow* [29]) that outlines a specific set of tasks and the sequence in which they must be performed. A task within this

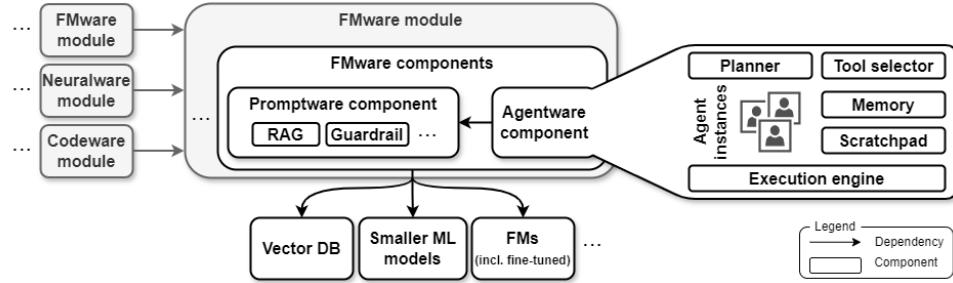


Fig. 1. Example of an FMware module with one FMware component comprised of a Promptware and an Agentware components.

workflow can also trigger another workflow. On the other hand, Agentware employs a coordination mechanism driven by autonomous agents. These agents determine both the steps to be taken and the order in which they occur. In CAs that use multiple agents, coordination results from interactions between the agents themselves, such as in Microsoft Autogen [30]. Patterns for CAs are starting to emerge [26].

3.3 The role of compilers in the SE 3.0 era

SE 3.0 represents a significant shift towards an intent-first approach, where development is no longer driven by code but by *intents* conveyed through interactive conversations between human developers and their AI counterparts [8]. We call this approach *conversation-oriented development*. In SE 3.0, AI takes the lead in automating the code creation process by **synthesizing** intents into executable software. We note that the resulting software system might be Codeware, Neuralware, Promptware, Agentware, or any combination thereof. Due to the new use cases and business opportunities unlocked by FMs and generative AI, this paper focuses on the synthesis of intents into FMware. As such, the compiler must search for the most suitable CA for the FMware based on a set of competing criteria (e.g., accuracy, cost, and latency).

3.4 The challenging task of synthesizing intents into FMware

Synthesizing intents into FMware is a very challenging task due to the complexity of FMware. Gone were the days when FMware was simply a thin wrapper around an FM (e.g., GPT-4o). Analogously to how machine learning (ML) models constitute only a minuscule part of AI systems from the SE 2.0 era [31], FMs also constitute an equally small part of the stack in the SE 3.0 era. Much of the code in an FMware lies in (i) configuration (e.g., setting up FM output parameters, such as temperature and top-k), (ii) data collection (e.g., curriculum engineering), (iii) RAG systems and embedding models, (iv) data verification (e.g., guardrails), (v) analysis tools (e.g., evals and benchmarks), (vi) process management (e.g., AIOps), (vii) machine resource management (e.g., Ray clusters), (viii) serving infrastructure (e.g., Ray serve), and (ix) monitoring (e.g., semantic observability). Due to the non-deterministic behavior of FMs, these components tend to be significantly more complex than those from Neuralware or Codeware. As an illustrative example, RAG itself further expands into a set of several interconnected components (see Figure 2).

We also emphasize that the many pieces of FMware must evolve in response to feedback data, especially in the context of *data flywheel* approaches [33]. The data flywheel is a self-reinforcing cycle in which data collection, analysis, and insights lead to ongoing improvements and growth within a system. As more data is accumulated (such as field, telemetry, or human feedback data), it enhances the accuracy and efficiency of algorithms and processes (e.g., model

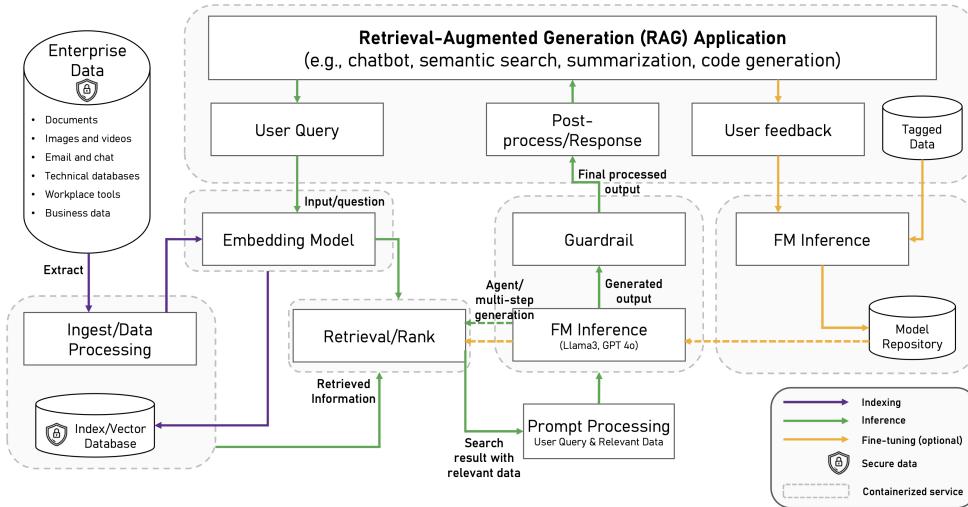


Fig. 2. RAG pipeline (adapted from OPEA’s reference RAG pipeline [32])

refinement and agent enhancements), which in turn improves user experiences and operational effectiveness. This attracts more users and generates additional activities, resulting in even more data, which drives further optimization. Consequently, FMware remains in a continuous state of evolution.

3.5 State of the practice

DSPy [34] is one of the first steps towards an end-to-end infrastructure to optimize an FMware program, contributing programming constructs to express a computation graph and an optimization method for the whole program. DSPy’s programming model [34] represents an FMware as a computation graph of constructs that involve two abstractions: modules (to perform common prompt operations like reasoning procedures) and signatures (to define a module’s input and expected output). DSPy’s compiler optimizes a DSPy program (written with the aforementioned programming model) by taking as input a “training set” (a.k.a. gold labels) and an optimization metric. The compiler first generates candidate solutions (e.g., few-shot examples for a prompt template or specific control flows for the program) and then uses a search algorithm (e.g., random search) to find the best candidate solutions based on the given metric.

TextGrad [35] adopts the same “gradient-descent” metaphor as ProTeGi (Section 2.3) to optimize an FMware program described as a computation graph with a syntax that resembles that of the popular DL PyTorch library. In TextGrad, the “gradients” generated by the reflection prompt propagate through the computation graph, describing how each component of the FMware program should be modified to improve the overall performance. More specifically, each node v of the computational graph represents unstructured data, such as natural language or an image. During compilation, these nodes perform a transformation on their associated data, following the aggregated “gradient” (e.g., a set of summary of “errors”) from all successor nodes of v . Objective functions in TextGrad are customizable and can be defined, for example, as a natural language text that is evaluated by an FM or the output of running an FM-generated piece of code against a set of test cases. TextGrad also differentiates between instance optimization and prompt optimization. While the former refers to optimizing the solution to a problem, the latter refers to optimizing the prompt templates themselves.

Despite the importance of prior advances in the representation of FMware programs and prompt optimization, our vision for *Compiler.next* in the era of SE 3.0 extends beyond these foundations. Tools like DSPy represent significant and mature advances in FMware program optimization, providing programming abstractions and optimizers that realize key aspects of our vision. DSPy, in particular, supports program-level optimization of prompts, demonstrations, and module configurations, with pluggable optimizers and systematic approaches to improving FMware programs. Our vision for *Compiler.next* builds upon and extends these foundations in four complementary directions.

First, we envision integration with the SE lifecycle (detailed in Section 5.3). While existing tools focus on optimizing FMware programs within their frameworks, *Compiler.next* addresses how intent compilation connects to requirements engineering, testing practices, CI/CD pipelines, maintenance and evolution, and collaborative development. This broader scope positions FMware compilation within complete software systems rather than as an isolated optimization problem.

Second, we propose a compiler infrastructure for cross-tool interoperability. This includes IRs that enable FMware programs written in different frameworks to share optimization advances, front-end/back-end separation allowing diverse FMware representations to use common optimizers, and community platforms for sharing compilation traces and gold-label datasets. While existing tools provide pluggable optimizers within their ecosystem, our vision extends toward a standardized compilation infrastructure that is agnostic to individual frameworks.

Third, *Compiler.next* explores multi-objective Pareto optimization rather than single-metric or weighted-combination approaches. Our prototype implements NSGA-II to simultaneously optimize competing objectives (e.g., accuracy, latency, cost) and present trade-off frontiers to users. While existing tools could optimize weighted combinations of metrics, true multi-objective optimization explores the Pareto frontier without requiring users to specify weights a priori.

Fourth, *Compiler.next* addresses compilation performance through systematic infrastructure (see Section 4.1). This includes semantic caching at multiple levels, compilation trace reuse that leverages prior compilations to accelerate future ones, distributed synthesis capabilities for parallelizing the search process, and transparent model serving optimization (including model layering and swapping). These performance optimizations are essential for making FMware compilation practical at scale, particularly in continuous integration environments where compilation time directly impacts development velocity.

We now turn to the architecture of *Compiler.next*, detailing the key components and design choices that realize these capabilities and support the research directions outlined in Section 5.

3.6 Positioning SE 3.0 Compilers within Broader SE Paradigms

While we frame our vision through the lens of compilation, we recognize that SE 3.0 compilers share conceptual similarities with other established top-down software engineering approaches. In particular, Model-Based Software Engineering (MBSE) has long explored the automatic generation of implementations from high-level specifications and models, and self-adaptive systems research has investigated mechanisms for runtime optimization based on quality objectives. Moreover, software product lines provide systematic approaches to managing variability through feature models and configuration.

We adopt the compiler framing for different reasons. First, compilation emphasizes systematic, repeatable transformation processes operating on well-defined representations, enabling rigorous optimization techniques. Second, the compiler analogy captures the distinction between compilation-time decisions (i.e., prompt synthesis and configuration selection) and runtime execution (e.g., FM invocation and agent coordination) of the FMware program. Third, compiler infrastructure naturally supports modular front-ends and back-ends, enabling interoperability across different FMware representations. Finally, the compiler abstraction provides a principled framework for reasoning about correctness,

efficiency, and optimization in ways that complement but differ from model-driven or adaptive system perspectives. Yet, these are complementary rather than competing viewpoints. As we discuss in Section 5.2, SE 3.0 compiler research can benefit from techniques developed in MBSE, self-adaptive systems, and software product lines research.

4 Compiler.next

Compiler.next represents a step forward in offering a solution to some of these challenges presented in our prior work discussion ten challenges for the engineering of trustworthy FMware [26]. In particular, our prior work outlines FMArts, a low-code, full lifecycle platform for engineering FMware comprised of three layers:

- **FMware hub (top layer):** Contains reusable tools and infrastructure to support the FMware engineering lifecycle,
- **FMware framework:** Encompasses a set of conceptual primitives to define and represent FMware programs,
- **Graph compiler and fusion runtime (bottom layer):** Provides infrastructure for running the FMware program optimized for the underlying FM deployment.

Within FMArts, *Compiler.next* plays the role of the higher-level program synthesizer that produces or refactors assets in the FMware Hub (e.g., prompts, agents, workflows) and provides a compiled plan to the Graph Compiler and Fusion Runtime for performance optimization and execution. Figure 3 depicts the technology stack of *Compiler.next*. In the following, we discuss each of its components.

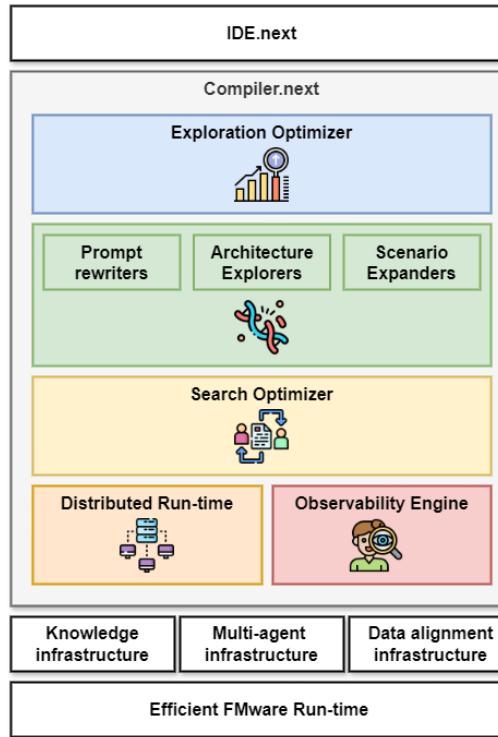


Fig. 3. The technology stack of *Compiler.next*.

Cognition Exploration Optimizers. Drives the search process efficiently and intelligently using techniques like self-reflection. It also allows the specification of different optimization objectives to improve FMware attributes such as correctness, latency, and cost.

Prompt rewriters. Focuses on enhancing and refining prompts, using advanced techniques to modify prompt structures for better outcomes. A prompt rewriter can leverage the several prompt engineering techniques and prompt patterns that have been published in the literature (e.g., in the form of a curated library [36]) to increase its chances of bootstrapping examples that will improve results.

Architecture explorers. Searches for optimal configurations of RAG parameters (see Section 3.3) and patterns of CAs, such as FM Call (a basic architecture focusing on single function model calls), FM Chain (a sequential model chain, where outputs of one model feed into the next), Router Agent (an architecture where an agent routes tasks based on the context or input type), State Machine Agent (a structured architecture that operates based on predefined states and transitions), and Autonomous Agents (fully autonomous agents that make decisions and act independently within the system).

Scenario expanders. Expands the scenario description and cases used to drive the search by synthesizing new scenarios from existing ones that include new dimensions. This ensures that, during the search, the FMware was optimized for different target domains with a diverse set of scenarios. In the context of *Compiler.next*, a scenario is a structured representation of the task that the compiler seeks to solve during intent compilation. A scenario defines the problem space by specifying the user intent, the associated gold labels, and the evaluation criteria used to assess candidate solutions. Scenarios thus act as the optimization environment within which *Compiler.next* searches for the best configuration of prompts, cognitive architectures, and system parameters. *Compiler.next*’s scenario expander mutates these task descriptions by synthesizing new variations of a scenario from the existing ones. For example, given a code generation task with a particular docstring, the scenario expander may generate paraphrased versions of the docstring, alternative formulations of the input-output pairs, or additional test cases. This expansion increases the diversity of the optimization environment, encouraging *Compiler.next* to search for solutions that are robust across variations of the original task, reducing the risk of overfitting to narrowly defined task instances.

Search optimizer. Leverages prior local and crowd-runs for more efficient driving of the search process. In particular, uses the compilation traces of prior compilations as feedback information and reuses it to make the next compilations more efficient (e.g., caching common compilation steps). Past search data can help in tuning the parameters of search algorithms (like genetic algorithms, simulated annealing, etc.) to better fit the problem space based on previous outcomes. Insights from past searches can guide the development of new heuristics that are more adept at solving specific types of problems, thereby improving search efficacy. Historical searches can also be used to understand user preferences, allowing for personalization. By continuously incorporating insights gained from historical search data, systems can incrementally improve.

Distributed synthesizer run-time. Uses a distributed platform to speed up the synthesizer. In the context of resource-limited devices (e.g., mobile), it is important that model layering or swapping can be transparently handled [37, 38].

Synthesizer observability engine. Enables debugging and traceability of the whole synthesizer, such that developers can understand the program states that caused issues and take action if needed. Debugging and traceability are needed at several levels of abstraction (e.g., CA selection, optimizations within a given CA, prompt optimizations).

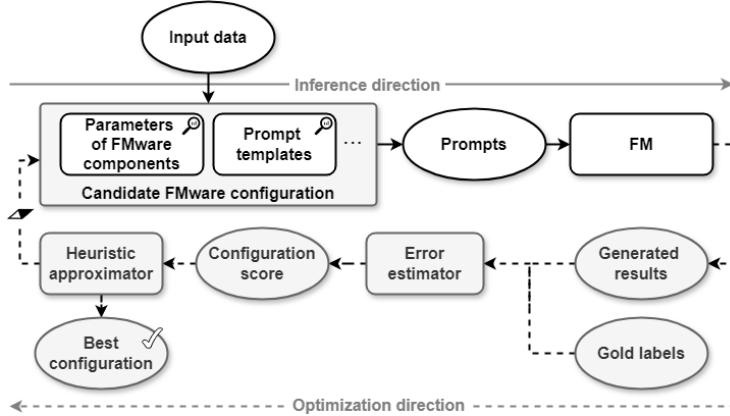
Other layers. On top of the stack is the *IDE.next* component. *IDE.next* is the new AI-native IDE that powers software development in the SE 3.0 era [8]. In turn, *Compiler.next* sits on top of different infrastructures that support FMware implementation, such as *knowledge infrastructure* to support RAG, *multi-agent infrastructure* to drive autonomous agents, and *data alignment infrastructure* to support FM alignment (e.g., fine-tuning processes). Finally, the bottom layer involves serving FMs efficiently.

4.1 The search mechanism

Both Promptware and Agentware components have a combination of input parameters for which they perform optimally for a certain task. For example, for a Promptware component, these parameters might be associated with the embedding models used in the RAG component or the parameters of the guardrail mechanism. Similarly, for the Agentware components, parameters associated with the agents' CA (such as the number of agents and the roles of each agent) and memory components (such as the memory buffer size) need to be determined. At the same time, the prompt templates used to interface the FMware components with the FMs also impact the general performance of the FMware application. Therefore, there is a need to find the set of components' parameters and prompt templates that lead to the best performance of the FMware application, while the search space of such parameters is too large to be explored manually.

Figure 4 depicts the general search mechanism of *Compiler.next*, highlighting the continuous optimization of components' parameters and prompt templates. The optimization process starts by instantiating the optimizable components of an FMware and inputting task-dependent data to these components, generating a specific *FMware configuration* based on the instantiated components. In the example of Figure 4, we focus on the set of components' parameters and prompt templates of FMware applications, but other components can also be subject to optimization. Afterwards, an inference step is performed by passing prompts to the target FM, generating results for the specific combination of prompts and FM. The *error estimator* then compares the generated results against a set of target *gold labels* that describe expected results for the specific task, generating a *configuration score* that estimates how close the generated results are to the optimal solutions (based on the gold labels as a reference of optimality). The error estimator can adopt different approaches to calculate the configuration score as long as they are suitable for the task performed by the FMware component under optimization. For example, to compare the generated results against the gold labels, a code generation task can use either text-based metrics, such as BLEU [39], or test-based metrics, such as computational accuracy [40]. In the current design of *Compiler.next*, we assume that gold labels are provided as static inputs to the search process. However, we recognize that in practice, specifications often evolve during compilation. For instance, requirements may be refined as developers gain a better understanding of the task, or new evaluation cases may be generated as the system exposes unanticipated behaviours. Finally, a *heuristic approximator* records the best configuration found during the optimization process and applies operations to modify the components' parameters and prompt templates to generate new candidates of FMware configuration and steer the optimization process towards better candidate solutions. The heuristic approximator can use different methods to modify the components' parameters (e.g., random search [41]) and prompt templates (e.g., using a FM to rewrite the template or using differentiable prompt formats such as soft prompts [42]). The optimization process is repeated until a certain stop criterion is achieved, such as the number of iterations, compilation time, or a threshold configuration score.

Conceptual model. In *Compiler.next*, each component of an FMware program is represented by an *Operation* that implements an arbitrary functionality and contains both static and dynamic parameters, with some of the dynamic

Fig. 4. Search iteration steps in *Compiler.next*.

parameters being optimizable. For example, an Operation can represent a Promptware component that performs a FM inference, for which a prompt template can be optimized, or retrieval from a vector database (RAG), for which the parameter associated with the top- k documents can be optimized. All Operation elements have a common interface that can be used to specify optimizable parameters and an Optimizer. An Operation is a stateful entity that can exchange messages with another Operation, being suitable to different FIRMWARE program representations, such as prompt chains [43], computation DAGs [25, 35], or CAs [13].

An Optimizer in *Compiler.next* is a pluggable component that specifies how the parameters of an Operation are optimized. For instance, one of the optimizers implemented in *Compiler.next* uses an NSGA-II multi-objective genetic algorithm (GA) [44, 45] to optimize prompt templates, allowing the customization of all genetic operators such as crossover, and mutation [44] in addition to the objective functions. This prompt template optimizer uses a FM to drive the search (e.g., to mutate a prompt candidate) and receives as parameters the associated intent with the Operation (e.g., “generate source code from documentation”), the input data (e.g., the function signature and documentation), the gold labels (e.g., unit tests), and references to both the *evaluator*- and *release*-FMs used to drive the search and evaluate the candidate results, respectively. An Optimizer can be extended or customized according to the task performed by the associated Operation. For instance, for the optimization of enumerable component parameters, our GA-based optimizer can use some genetic representation of the parameters (e.g., an array of binaries for integers and floats) [46], avoiding the high costs of using a FM to drive the search.

An Optimizer also has an aggregation relationship with another customizable element of the *Compiler.next* framework called EvaluationBench, which defines the format and evaluation logic of the gold labels used during the optimization process. This is an important aspect of the framework because gold labels highly depend on the task performed by the associated Operation with an Optimizer. In *Compiler.next*, we also separate the FMs used for application deployment (*release*-FM) from the FM used to drive the search (*evaluator*-FM) because the former can be a smaller FM that does not have the same capabilities of the latter to generate diverse and quality candidate solutions during the optimization process.

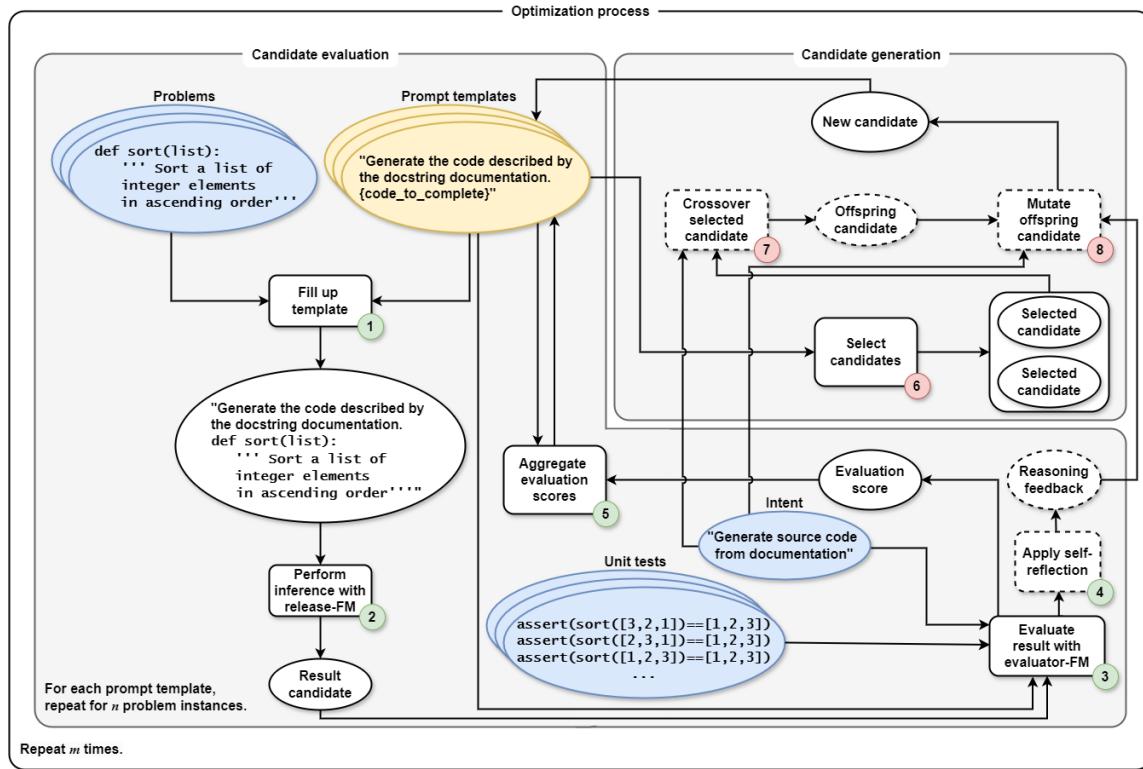


Fig. 5. The operation of one of *Compiler.next*'s optimizers in our running example of an FMware for code generation. The dashed box indicates optional steps.

4.2 Illustrative Example: Optimizing Prompts for Code Generation

To illustrate the working mechanism of *Compiler.next*, we use a running example of a code generation FMware containing a single Promptware component. The FMware receives as input a function signature with documentation describing the desired functionality and generates the implementation. The compilation objective is to synthesize an optimized system prompt template that maximizes accuracy and performance on code generation tasks (i.e., generating semantically correct code that runs fast). The software maker provides *Compiler.next* with an intent (“Generate source code from documentation”), a collection of problems (function signatures with documentation), and gold labels consisting of unit tests for evaluation.

Figure 5 illustrates the optimization process, which operates through iterative cycles of candidate evaluation and candidate generation. The process begins with an initial population of prompt template candidates, either user-provided or automatically generated by an FM based on the intent, and refines this population over multiple iterations to identify high-performing configurations.

1. Fill up template: The first step of the optimization occurs during the candidate evaluation process (left side of Figure 5). Each prompt template contains placeholders that are instantiated with information from a problem instance. In our example, the placeholder code_to_complete is replaced with a function signature and its documentation string. During optimization, the placeholder content remains static, while other template components (e.g., few-shot examples,

instructions) are optimized. Each component can be optimized separately. The results of template instantiation are cached, as the same problem-template pairs frequently recur across iterations.

2. Perform inference with release-FM: The instantiated prompt is executed using the release-FM to generate a result candidate (in this case, completed code that concatenates the function signature and documentation with the FM-generated implementation). The prompt template is optimized specifically for the FM that will be deployed in the production FMware application. All FM inference calls are cached based on their input prompts, substantially reducing costs when identical or similar prompts are evaluated in subsequent iterations.

3. Evaluate result with evaluator-FM: The result candidate is evaluated using two objective functions: (1) computational accuracy, measured as the proportion of unit tests that pass when executed against the generated code, and (2) execution latency, measured as the time required to execute the generated code. The evaluation approach is customizable and can incorporate additional quality attributes (e.g., code complexity) depending on user preferences and downstream requirements. Evaluation results are likewise cached, enabling rapid re-assessment of previously encountered candidates.

4. Apply self-reflection: Optionally, when unit tests fail, self-reflection prompts [47] can generate reasoning feedback about how the prompt template should be modified to improve performance. A key challenge with this technique is that feedback often becomes overly specific to individual problem instances. Since the optimization objective is to maximize performance across all problem instances, including out-of-sample instances at runtime, measures must ensure the feedback remains sufficiently general. The evaluator-FM, which may differ from the release-FM, produces this self-reflection feedback to leverage superior emergent abilities in generating actionable guidance.

5. Aggregate evaluation scores: Each prompt template is evaluated against n different problem instances, and the individual scores are aggregated to compute an overall fitness score for the template. This aggregated score quantifies how effectively the template enables the release-FM to generate correct code across diverse scenarios.

6. Select candidates: During the candidate generation process (right side of Figure 5), the optimizer selects prompt templates based on their aggregated evaluation scores. The selection algorithm employed in our NSGA-II implementation uses crowding distance to select dispersed individuals on the Pareto front [44], though alternative approaches such as roulette-based selection can be configured to foster diversity during search space exploration [48].

7. Crossover selected candidates: Selected templates undergo crossover, which generates offspring by instructing an FM to combine elements from parent templates while considering the provided intent. As a customizable operator, crossover can adopt lower-cost approaches that avoid FM inference or can be skipped entirely. The results of crossover operations are cached to avoid regenerating identical offspring in subsequent iterations.

8. Mutate offspring candidate: Offspring templates are mutated by instructing an FM to modify the prompt based on the intent. To reduce costs, mutation can combine FM inference with simpler token replacement strategies [49]. Mutation results are also cached, as identical mutations may be proposed multiple times as the population converges. The newly generated candidates enter the evaluation phase in the subsequent iteration, completing the optimization cycle.

The flexibility of *Compiler.next* enables implementation of diverse search strategies. For instance, a gradient descent-like approach can be implemented using a crossover operator that forwards selected candidates to a mutator that modifies templates based on self-reflection feedback. Alternatively, *Compiler.next* can evaluate a set of prompt template alternatives and select the best performer, generating reasoning traces about potential improvements. *Compiler.next* also leverages parallelization by evaluating multiple candidate results concurrently, further reducing overall compilation time.

As iterations progress, the population gravitates toward configurations that consistently achieve high evaluation scores on the gold labels. Upon convergence, *Compiler.next* selects the highest-performing prompt template from the final population as the optimized FMware artifact, ready for deployment in the code generation assistant. This example demonstrates how *Compiler.next* applies systematic, search-based optimization to synthesize effective FMware configurations, transforming an abstract intent into a concrete, empirically validated prompt template through automated exploration of the design space.

4.2.1 Cost optimization through semantic caching. The iterative nature of the optimization process creates substantial opportunities for cost reduction through semantic caching. Unlike traditional exact-match caching, *Compiler.next* employs semantic caching that stores embeddings of prompts and retrieves cached results for semantically similar inputs. For FM inference calls, both to the release-FM during code generation and to the evaluator-FM during assessment, and for genetic operator outputs (crossover and mutation), the cache compares the semantic similarity of the current input against previously cached entries using a configurable similarity threshold.

This similarity threshold introduces a critical trade-off between compilation speed and the exploration of the search space. A higher threshold (e.g., 0.95) requires near-exact semantic matches, resulting in fewer cache hits but ensuring diverse candidate evaluation across the search space. Conversely, a lower threshold (e.g., 0.80) increases cache hit rates and dramatically reduces compilation time and FM invocation costs, but risks limiting exploration by treating semantically similar yet potentially distinct candidates as equivalent. The optimal threshold depends on the compilation scenario: early iterations may benefit from lower thresholds to quickly eliminate poor regions of the search space, while later iterations may require higher thresholds to distinguish between high-quality candidates with subtle differences.

The semantic caching mechanisms prove particularly effective as the population converges toward high-quality regions of the search space, where candidates increasingly share structural components and generate similar prompts. Additionally, each iteration of the optimization loop is checkpointed, enabling recovery from failures without repeating expensive computations. The cumulative effect of semantic caching, when properly tuned, substantially reduces both compilation time and the number of costly FM invocations required while maintaining sufficient search diversity to discover high-performing configurations.

4.2.2 Co-optimizing composed FMware components. The example presented above focuses on optimizing a single Promptware component in isolation. However, FMware applications often comprise multiple composed components with dependencies between them. For instance, an Agentware component may contain multiple Promptware components, each with its own prompt template, or a RAG component may precede a Promptware component in a processing pipeline. Our current implementation of *Compiler.next* addresses this compositional challenge through joint optimization, where the search space encompasses configurations across multiple components simultaneously, evaluating candidates based on the end-to-end performance of the composed FMware. In this approach, the optimizer explores parameter combinations across all components (e.g., RAG retrieval settings and prompt template structures) to identify configurations that maximize overall system performance.

An alternative approach for SE 3.0 compilers is hierarchical optimization, in which components are optimized in stages, with information flowing between phases. For example, in a code generation system where a RAG component retrieves relevant code examples before the Promptware component generates the completion, after optimizing the prompt template, the compiler could leverage this information (such as the structure of the optimized prompt, performance signals indicating how retrieval quality affects downstream generation, or self-reflection feedback) to guide the optimization of the RAG component's parameters (e.g., the number of examples retrieved, the similarity threshold, or

the embedding model used for retrieval). A subsequent optimization pass of the Promptware component could then refine the prompt given the updated RAG configuration. This sequential strategy offers potential advantages, including controlled search space dimensionality and opportunities for pipelining (e.g., optimizing one component while another is being evaluated), though it may miss synergies between component configurations that joint optimization is better at discovering. The compositional nature of FMware architectures and the trade-offs between these optimization strategies represent important directions for future SE 3.0 compiler research.

4.3 Initial validation

To provide a concrete proof of concept of *Compiler.next*, we conducted a controlled case study using the HumanEval-Plus benchmark, a widely used dataset for evaluating code generation systems. HumanEval-Plus consists of Python programming tasks where the goal is to generate the body of a function given its signature and docstring. This case study focuses on two central features of *Compiler.next*. First, we demonstrate the automated synthesis of system prompts from user-provided intents. Starting from the user-provided intent (“Generate source code for a function given a problem description found in the function’s documentation”), *Compiler.next* generates candidate prompts without human intervention. The search mechanism used by the system prompt synthesizer leverages the NSGA-II multi-objective evolutionary algorithm to optimize the candidates across three objectives: accuracy, latency, and execution cost (see Figure 4). Accuracy is measured as the proportion of generated solutions that pass all the unit tests associated with each HumanEval-Plus entry. Latency corresponds to the runtime required to evaluate a candidate solution. Execution cost is measured by the number of tokens consumed per run (input and output). This setup demonstrates how intent-level specifications can be automatically compiled into optimized system prompts while balancing trade-offs between competing objectives.

Second, we evaluate the caching mechanism that accelerates the compilation process by reusing results of previously executed operations. Specifically, we implemented a tracer that manages a two-level semantic cache (we use an Euclidean similarity threshold of 0.85). At the first level (L1), cache entries are indexed by a hashable representation of the combination of the intent and the description of the training data, with each entry pointing to an associated second level (L2) cache. At L2, cache entries are indexed by a hash of the optimizable parameter (in this case, the prompt under evaluation). Each L2 entry stores the result of a time-consuming operation, such as a fitness calculation, crossover, or mutation in our NSGA-II implementation of the optimizer. When the optimizer requires one of these operations, the tracer intercepts the request and then attempts to locate a matching entry in the L1 cache. On a hit, it proceeds to the corresponding L2 cache, where a successful match immediately returns the stored result, such as the fitness score of a previously evaluated prompt. If a miss occurs at either level, the operation is executed and its result is stored in the relevant L1 and L2 cache entries. This design reduces redundant computations while ensuring that the optimization process remains faithful to the intended search dynamics.

In our evaluation, we allowed five generations of ten candidate solutions per task, resulting in a total of fifty candidates per optimization run. Seventy percent of the HumanEval-Plus tasks were held out as gold labels to guide the optimization process, while the remaining thirty percent were used for evaluation. We tested two foundation models, Qwen2.5-7B-Instruct and GPT-4o-mini, and measured performance for the initial synthesized prompts as well as the best optimized prompts after five generations. We further compared runs using GPT-4o-mini with caching enabled against runs without caching to highlight the trade-off between compilation efficiency and diversity of search space exploration.

The results, summarized in Table 1, show that *Compiler.next* improves accuracy while also reducing latency and execution cost across both models. Table 2 shows that the caching mechanism provides substantial speed-ups of 22.1% by avoiding redundant evaluations, though at the cost of a reduction in exploration, as previously seen solutions are favoured over newly generated ones. Overall, this controlled case study demonstrates the feasibility of *Compiler.next* and some research opportunities for optimizing intent compilation for AI-native software engineering.

Table 1. Evaluation of *Compiler.next* on the HumanEval-Plus benchmark. We report accuracy (%), average latency (s), and average number of tokens used per run for the best initial synthesized prompt and the best optimized prompt.

Model	Metric	Initial	Optimized	Improvement (%)
Qwen2.5-7B-Instruct	Accuracy	0.26	0.56	46.4
	Avg. latency (s)	14.2	10.8	76.6
	Avg. # tokens	537.1	369.3	68.7
GPT-4o-mini	Accuracy (%)	0.68	1.00	47.0
	Avg. latency (s)	8.7	5.0	42.5
	Avg. tokens (#)	500.0	417.1	16.5

Table 2. Comparison of *Compiler.next* performance with and without caching on the HumanEval-Plus benchmark using GPT-4o-mini. We report accuracy (%), average latency (s), and average number of tokens used per run for the best optimized prompt. Total time is the end-to-end compilation runtime.

Metric	Without Cache	With Cache	Difference
Accuracy	1.00	0.70	-30%
Avg. latency	5.0	5.9	-18%
Avg. # tokens	417.1	467.0	12%
Total Running Time	8m:15s	10m:27s	22.1%

The design of *Compiler.next* not only defines the architectural principles of FMware compilation but also directly surfaces a set of open research challenges. In the next section, we examine these challenges in detail, showing how each derives from the framework’s features and illustrating how *Compiler.next* can guide and structure future research on FMware.

5 Research & Development Roadmap for Compiler.next

In this section, we discuss research challenges and opportunities for advancing intent compilation in SE 3.0. In Section 5.1, we discuss 10 calls for action that express our vision of the core SE challenges to be solved around this topic. In Section 5.2, we discuss how *Compiler.next* and our calls for action intersect with “top-down” SE methods like model-based software engineering (MBSE) and self-adaptive systems, inspiring researchers to explore these intersections to solve SE3.0 compilation challenges. Finally, in Section 5.3, we discuss how our calls for action connect to well-established SE practices and the core research questions around these connections.

5.1 Calls for action

This section describes a research and development (R&D) roadmap that culminates in 10 calls for action that express our view on the SE challenges around the implementation of SE 3.0 compilers. These challenges were derived based on a

combination of our experience developing a prototype of *Compiler.next*, in-depth literature reviews [50], and discussions with top academics and industry leaders during several events (e.g., FM+SE Vision 2030, FM+SE Summit 2024). Building on the *Compiler.next* framework described in Sections 3 and 4, we organize the 10 calls for action of FMware research into four dimensions and associate each call for action with concrete *Compiler.next* features. Table 3 summarizes these dimensions, showing how the framework defines the architecture of FMware compilation and structures the R&D agenda we propose. The next subsections present each of the calls for action. We map each call for action to the associated component of the technology stack shown in Figure 3 by matching the icons of the figure and the section titles. Subtitles with the  icon indicate cross-cutting calls for action.

5.1.1 Quality constructs for FMware representation . The development of SE 3.0 compilers would benefit from establishing constructs with semantics that can represent all common operations and control logic of an FMware program and can be translated to an intermediate representation (IR) that different back-end optimizers can manipulate. Intermediate representations are a foundational concept in compiler design, serving as the bridge between source languages and target architectures. A well-designed IR must be accurate—capable of representing source code without loss of information—and independent of any particular source or target language. The use of IRs has enabled compiler systems like LLVM to support many different source languages, generating code for many different target architectures [16], demonstrating the power of this abstraction for compiler modularity and reusability.

Modern compiler infrastructures have evolved toward increasingly flexible IR designs. LLVM’s intermediate representation provides a language-independent, typed assembly language that serves as a portable format for optimization across multiple passes. More recently, MLIR (Multi-Level Intermediate Representation) has extended this concept by allowing multiple levels of abstraction to coexist within the same compiler system, with dialects enabling domain-specific operations while maintaining interoperability. This multi-level approach has proven particularly valuable for heterogeneous computing scenarios and domain-specific optimization.

For SE 3.0 compilers, such constructs should allow the definition of intents [51] while avoiding eventual ambiguities, which might be particularly challenging if the intents are expressed using natural language. We also argue that compilers should be capable of compiling existing FMware programs written in any representation (e.g., LangChain and AutoGen) by providing a front-end component that can process programs written in such existing representations, transform them into this IR, and send them to a back-end compilation process whenever needed.

As an analogy with DL compilers that use tensor-oriented constructs (tensor in, tensor out) as building blocks of the computations, SE 3.0 compilers can adopt prompt-oriented constructs (prompt in, generation out) as such building blocks. Such constructs should be attentive to common PL quality criteria such as readability, orthogonality, and simplicity. In addition, the definition of proper constructs to represent the control logic of Agentware, in particular, is still an open problem, as it is in a continuum spectrum between fully manual – where software makers specify all the execution paths based on the evaluation of inputs and outputs – and fully automated – where the FM drives all decisions based on reasoning techniques such as chain-of-thought (CoT) [52].

Call for Action 1: FMware representation needs constructs with semantics that allow software makers to express complex FMware programs through intentions. In particular, such constructs should be able to represent every component of an FMware and their interactions, with different levels of controllability, abstraction, and programmability. Such constructs can then be transformed to a common IR that can interoperate with different backend optimizers.

Table 3. Dimensions of *Compiler.next* and their relationship to the Calls for Action.

Dimension	<i>Compiler.next's</i> design feature	Calls for Action	Rationale
Firmware program representation	<ul style="list-style-type: none"> - Representation of Firmware modules as modular compositions of parameterized heterogeneous artifacts (Sec. 3.1). - Architecture explorers search for optimal configurations of parameters of heterogeneous artifacts (Sec. 4). 	Calls 1, 2, and 9	By composing Firmware from modular components and supporting the optimization of multiple architectural patterns, SE 3.0 compilers motivate the need for programming constructs, heterogeneous compilation, and interoperability to be useful for a range of applications written with different technology stacks.
Result Validation	<ul style="list-style-type: none"> - Synthesizer observability engine enables debugging and traceability of the whole synthesizer and its outputs (Sec. 4), including how quality attributes of the generated program change during compilation. - Compilation traces involving successful trajectories are re-used in future compilations, while failed trajectories can be used as feedback information to understand common error patterns (Sec. 4). 	Calls 5 and 7	Observability and trace reuse support both runtime validation and reproducible builds, ensuring Firmware trustworthiness.
Computational Performance	<ul style="list-style-type: none"> - Caching common compilation steps to make the next compilations more efficient, with results from past searches guiding the development of new heuristics (Sec. 4). - Distributed synthesizer run-time uses a distributed platform to speed up the synthesizer, including transparent handling of model layering or swapping (Sec. 4). 	Calls 3, 6, and 10	SE 3.0 compilers achieve performance by combining heuristics, distributed execution, and trace reuse. These characteristics motivate research into efficient heuristics and community trace sharing.
User Priorities & Goals	<ul style="list-style-type: none"> - Cognition exploration optimizers allow the specification of different user-defined and context-dependent optimization objectives (Sec. 4). - Gold labels nudge the optimization to improve the Firmware in specific instances and use cases that are of interest to the users, and can automatically evolve during compilation. 	Calls 4 and 8	SE 3.0 compilers must adapt to user goals, balancing multiple optimization objectives with a broad evaluation of user-prioritized scenarios.

5.1.2 End-to-end FMware optimization  An FMware is built with several interacting components connected to different FMs. To interface with the FMs, these components generate dynamic outputs that are passed as arguments to prompt templates, which in turn has a significant impact on the outputs generated by the FMs [53]. As such, fully automating the search for the optimal FMware configuration goes beyond the simple definition of prompt templates and also requires optimizing the parameters of all system components that interact with a FM. For example, an FMware that applies a RAG pipeline must determine the best combination of encoder, retriever, and generator [54], as the retrieved information will be part of the prompt given to a FM. Similarly, a multi-agent component of an FMware requires the determination of the agents' data and control flow, the CA [55, 56] and the different reasoning strategies [57] that the agents carry out to perform sophisticated computational processes with the FM [13]. Therefore, if the compilation process aims at optimizing the overall performance of the FMware, then all the system parameters deserve the attention of the optimization process.

Call for Action 2: The compilation of FMware needs to move beyond the optimization of prompt templates in isolation to include the co-optimization of the different components of the FMware. Despite the recent developments towards this direction, all FMware components must be considered, particularly with better support for Agentware-based applications.

5.1.3 Effective search heuristics  Compiling an FMware program involves the joint optimization of prompt templates and several other system parameters. Therefore, the search space for compiling an FMware considerably increases as the number of components and their parameters increases, making the compilation an intractable problem requiring a heuristic search. Search-Based Software Engineering (SBSE) has established a rich repertoire of optimization techniques for SE problems [58, 59], including genetic algorithms, simulated annealing, and constraint-solving approaches. These techniques have been successfully applied to diverse problems such as test case generation, requirements selection, and refactoring [60]. More recently, program synthesis research has explored both classical search methods and neural approaches for generating code from specifications [61, 62], demonstrating that hybrid techniques combining search with learned heuristics can be particularly effective.

However, synthesizing FMware presents distinct challenges that differentiate it from traditional program synthesis. While classical program synthesis operates on well-defined formal specifications and deterministic execution semantics, FMware compilation must navigate the probabilistic nature of FMs, the semantic rather than syntactic correctness criteria, and the need to co-optimize heterogeneous components (prompts, RAG parameters, agent configurations) simultaneously. Moreover, searching in the FMware domain often involves the repeated and systematic application of FM-driven heuristics (e.g., prompt mutations) followed by (potentially) expensive evaluations of the intermediate solutions, which calls for an evidence-based determination of cost-effective heuristics that guide the search to quickly converge to a near-optimal solution.

Despite the extensive literature on search heuristics, it is not always clear which properties of a prompt or which parameters of each FMware component contribute the most to the overall performance of an FMware application. For example, the large majority of the empirical research that examined the impact of changing a prompt in the FM's response focuses on the *structure* of the prompt [10–12, 63], but other properties of the prompt (and of an FMware parametrization, more generally) still need research. Understanding these properties would enable the adaptation of established SBSE techniques (such as informed mutation operators [64] and fitness landscape analysis [65]) to the unique characteristics of FMware optimization.

Call for Action 3: Empirical research is needed to demonstrate the prompt features and FMware parameters that influence FM's output the most, such that search heuristics are defined based on such features and the search budget can be spent more effectively.

5.1.4 Gold labels construction  To compile an FMware, the search procedure requires gold labels to determine the quality of each intermediate solution and guide the search toward prompt templates and component parameters that lead to the best solutions [66]. The software engineering community has developed extensive benchmarking frameworks to evaluate FMs and FM-powered systems across diverse tasks. Prominent benchmarks include HumanEval and MBPP for code generation [67, 68], SWE-bench for realistic software engineering tasks [5], MMLU for multitask language understanding [69], and task-specific datasets for question-answering, summarization, and reasoning. These benchmarks provide standardized evaluation datasets with gold-standard examples that have enabled systematic comparison of different models and approaches.

In the context of FMware compilation, gold labels are associated with the tasks performed by the FMware component under optimization (see Section 5.1.1), being materialized as a set of demonstrations along with their ground truth for evaluating the optimized parameters by the compiler. Such gold labels can be used at the prompt template level to evaluate a specific template and the parameters of the components associated with that template, or at the FMware level to jointly evaluate the configuration of all parameters and prompt templates simultaneously. However, such gold labels are not highly available for the diversity of tasks that software makers can want to perform with an FMware. Despite the ability of FMs to generate such gold labels, research is still required to determine whether (and to which extent) this synthetic generation approach would introduce any negative side effects in the compilation, particularly given concerns about data contamination where models are evaluated on data similar to their training sets [70].

Call for Action 4: The different communities investing in FMware R&D should create gold labels with clearly stated assumptions about the data. In particular, such gold labels should be of high quality, with independent data points, and representative of true populations.

5.1.5 Quality range estimation  Compilation of intents into FMware is not done with the assumption of the deterministic behaviour of the compiled components. Hence, intent compilers should include mechanisms to guarantee that the compiled FMware executes at pre-defined quality standard levels within a confidence interval. For example, an FMware that performs sentiment classification [71] might wish to execute at a specified accuracy range (i.e., the proportion of correctly classified intentions). Similarly, an FMware might want to execute under a range of other quality attribute metrics related to safety or fairness, for example. Such capability can be achieved by sampling these quality attributes during the compilation process (see Section 5.1.4) and calculating, within a provided confidence level, a range of quality attribute metrics for which the compiled FMware operates under many different configurations. The compilation process would then search under the constraint of the pre-specified confidence level and quality attribute metrics. If the FMware program cannot be compiled within a specified threshold of quality levels, the compilation should fail, such that measures can be taken to "fix" the program (e.g., by adding an extra component, changing the static parameters of some existing component, or refining the intents expressed with the FMware programming formalism).

Call for Action 5: SE 3.0 compilers should provide mechanisms to calculate the probability that a program will execute under certain quality thresholds and inform the software maker when such thresholds are not achieved by any of the searched FMware configurations.

5.1.6 Efficiency improvement and cost reduction  Compiling an FMware is computationally costly [49], as it requires multiple evaluations of a performance metric given a candidate configuration of the system, with each evaluation typically requiring one or more inferences with the FM. The high computational (and potentially financial) costs associated with repeated FM calls present a substantial bottleneck for compilation efficiency. Recent advances in LLM inference optimization have demonstrated promising approaches to address these challenges. Prompt compression methods such as LLMLingua achieve up to 20x compression with minimal performance loss by identifying and removing less important tokens [72], directly reducing both latency and cost per inference. Additionally, optimizations to the inference process itself, including techniques such as PagedAttention [73] and speculative decoding [74], can significantly reduce the per-call overhead of FM interactions.

Hence, the compilation of an FMware requires measures to improve the compilation efficiency and allow the search space to be efficiently and effectively explored. While these optimization techniques have been developed primarily for improving LLM application performance, they are directly applicable to the compilation context. General methods such as concurrency, semantic caching [75], and check-pointing have the potential to improve efficiency and reduce costs of compilation, but research still needs to quantify the improvements of using such methods in the specific context of FMware compilation where compilation traces, candidate solutions, and fitness evaluations present unique caching opportunities. In addition, specific compile optimization methods such as operator fusion [17] and cost modelling [76] can be adapted from traditional and DL compilers to SE 3.0 compilers. However, the iterative and exploratory nature of search-based compilation introduces additional considerations: caching must balance between storing expensive computation results and avoiding over-reliance on previously seen solutions, which could limit the diversity of the search process (as discussed in Section 4.1).

Call for Action 6: Techniques to reduce the latency and increase the throughput of compilation should be further developed, such that the parameter search space can be explored more thoroughly for a given computing budget. In particular, research should investigate how semantic caching, prompt compression, and inference optimization techniques can be effectively integrated into search-based compilation workflows while maintaining search diversity. Similarly, techniques to reduce the cost of performing an end-to-end compilation should be provided and evaluated, with careful attention to the trade-offs between compilation efficiency and solution quality.

5.1.7 Reproducible compilations  Reproducible compilation is an important feature to allow software makers to build trustworthy FMware, as it allows the delivered FMware to be externally verified (e.g. to ensure that the compiler is not tampered with and no adversarial prompts are injected during the compilation process). The SE community has developed robust practices around reproducible builds as a set of development practices that create an independently-verifiable path from source to binary code. Reproducible builds enable external parties to verify that distributed binaries match their source code exactly, strengthening software supply chain security and allowing for independent audits [77, 78]. These practices have been successfully adopted by major open-source projects (e.g., Debian, NixOS, Bitcoin Core) and rely on deterministic build systems, stable inputs, pinned dependencies, and well-defined build environments.

Ideally, the compilation process for FMware should be reproducible, with the same FMware program and configuration compiling to the same output set of parameters and prompt templates [79]. However, achieving reproducibility in intent compilation presents unique challenges compared to traditional reproducible builds. Because FMs are typically used to guide the compilation process (e.g., to implement prompt mutation heuristics [22]), the search process is

invariably non-deterministic. In this case, FM decoder parameters (e.g., temperature [80]) can be adjusted to mitigate non-deterministic outputs from the FM-generated content, making the process more reproducible in practice. However, non-deterministic behavior can still arise from multiple sources (e.g., sampling strategies, runtime environment, or hardware dependencies). This highlights that reproducibility in intent compilation is not only a matter of tuning FM parameters, but rather a broader systems challenge requiring systematic control of all sources of non-determinism. Similar issues are well known in other software engineering domains, such as reproducible builds and machine learning training, where exact replication of outcomes is difficult. Therefore, SE 3.0 compilers should adopt and extend established techniques (e.g., record-and-replay, systematic logging of seeds and runtime states [77, 78]) to mitigate non-deterministic behaviour and make compiled FMware more trustworthy.

Moreover, search algorithms typically require randomness to ensure that various solutions are verified for performance and that the algorithm can escape local optima, posing challenges to the reproducibility of compilation. Therefore, compilers should balance the need for reproducibility with the need for effective search, potentially through controlled randomness via documented seed values or by ensuring that the compilation trace itself (including all intermediate states and decisions) can be deterministically replayed even if the original search was non-deterministic [81]. Similar to build [82] and DL [83] training reproducibility, we foresee a window of research challenges and opportunities related to the reproducibility of compilers.

Finally, the SE 3.0 vision and the compiler infrastructure we propose around it suggest reconsidering how reproducibility is achieved for FMware. Rather than pursuing bit-for-bit identical outputs, which may be unattainable with FMs, *Compiler.next* enables quality-level reproducibility in FMware by systematically guaranteeing that recompiled systems meet the same performance thresholds even if exact artifacts differ (see Section 5.1.5). The compiler provides (a) systematic visibility into quality metrics, making it easy to detect when recompilation fails to meet requirements, (b) automated validation against gold labels to verify functional equivalence, and (c) automated recompilation when models or requirements change, guaranteeing quality restoration without manual prompt adjustments. This shifts reproducibility from “regenerate exact artifacts” to “guarantee quality properties” which is both more achievable and more valuable for maintaining FMware components over time.

Call for Action 7: Despite the inherent challenges to guarantee determinism, SE 3.0 compilers should implement techniques to make the compilation process reproducible such that the executable FMware can be verified by external parties. In particular, reproducibility in intent compilation should adopt and extend well-known approaches from reproducible builds (such as deterministic build systems, stable inputs, and systematic documentation) and ML training (such as record-and-replay and systematic logging). Research should investigate how to balance the tension between search diversity and reproducibility, potentially through mechanisms such as deterministic replay of compilation traces or controlled randomness with documented seeds.

5.1.8 User-defined optimization objectives  During the compilation of a FMware, software makers might want to focus on different co-existing optimization objectives, such as the cost of executing the FMware (e.g., measured in number of tokens), the overall application latency, or the application’s performance in a task. Similarly to traditional compilers where users can define different optimization preferences, compilers should treat optimization objectives as user-defined parameters, with software makers possibly assigning different weights to each of such objectives. This feature would require compilers to implement mechanisms to allow users to define the optimization objectives with different levels of flexibility, either with pre-defined objectives directly implemented by the compiler or with a plug-in-based approach where any objective can be defined by the user. Such a feature also requires the implementation

of multi-objective search algorithms by compilers and the ability to select a single solution from the potential set of competing solutions that dominate each other in different objective dimensions.

Call for Action 8: A flexible and reusable approach to define optimization objectives should be defined by SE 3.0 compilers, which would support the alignment with specific users' needs and use cases.

5.1.9 Interoperability between compilers . The number of SE 3.0 compiler offerings will eventually increase, posing emerging interoperability requirements for FMware programs, so they can be composed using multiple compiler technologies and the same FMware implementation can be easily ported from one compiler to another, avoiding vendor lock-in. Cross-language and cross-compiler interoperability has been a longstanding challenge in SE. Traditional approaches have ranged from defining common calling conventions and application binary interfaces [84], to building multi-language runtimes that share a common intermediate representation [85]. The success of LLVM demonstrates how a well-designed, language-independent IR can enable front-ends for many programming languages to interoperate with back-ends for many target architectures [16].

For SE 3.0 compilers, possible approaches to solve the interoperability problem include directly transforming a program representation that is compatible with one compiler to another representation compatible with another compiler [86], or proposing a unified IR to which all FMware programs can be converted [87] back-and-forth. The latter approach, inspired by LLVM's success in the traditional compiler domain and MLIR's extensible dialect system, appears particularly promising. A standardized FMware IR could define common abstractions for prompts, cognitive architectures, agent coordination, and RAG components while allowing compiler-specific dialects for domain-specific optimizations.

Call for Action 9: It is important that software makers benefit from the strengths of each SE 3.0 compiler offer and can compose programs that are compatible with different compilers. Therefore, efforts should be made to ensure that the different compilers can interoperate, such as establishing a standardized IR to which FMware programs can be transformed.

5.1.10 Community-sharing of compilation traces . The compilation process generates traces containing valuable information that can be used to continuously improve the compilation process. In particular, each compilation involves different problem definitions (e.g., through the expression of intentions, as proposed in Section 5.1.1) and objective functions calculations (e.g., by evaluating task performance using gold labels, as proposed in Section 5.1.4). If this information is recorded along with the intermediate solutions (i.e., values of the optimized parameters and prompt templates) during the compilation process, such information can be reused in future compilations that have the same problem definitions and objective function or used to build cost models [17], ultimately avoiding re-computation of some of the intermediate solutions and the associated values of the objective function. The traces can also include final prompts for a certain task, and they can be used to bootstrap the prompts of other similar FMware and FMs. Therefore, compilers will benefit from community sharing of such compilation traces, as this information can be reused to benefit all users of the same compiler.

Call for Action 10: Compilation traces contain valuable information that can be used to avoid re-computations during compilation. As such, compilers should provide resources to allow users to opt-in for sharing of compilation traces and mechanisms to feedback this information into the compiler to accelerate and improve the compilation process.

5.2 Research Opportunities at the Intersection with Top-Down SE Paradigms

The challenges outlined in our Calls for Action have parallels with longstanding research in several “top-down” SE approaches that transform a higher-level representation of a software system into executable artifacts. While we frame our vision through compilation, these paradigms offer complementary perspectives and proven techniques that can inform SE 3.0 compiler research.

Model-Based Software Engineering. MBSE emphasizes the systematic generation of implementations from high-level models, using model transformations and well-defined modelling languages to support the entire system lifecycle [88]. SE 3.0 compilers share this top-down philosophy: both transform abstract specifications (models or intents) into executable implementations. However, the approaches differ in fundamental ways. MBSE typically operates with deterministic, formal models and transformations, while SE 3.0 compilers must navigate the non-deterministic behaviour of FMs and the ambiguities of natural language. Call for Action 1’s emphasis on quality constructs for FMware representation parallels MBSE’s decades of work on metamodels and modelling language design, research that established principles of abstraction, compositionality, and semantic precision. Similarly, Call for Action 9’s focus on interoperability aligns with MBSE’s extensive work on model interchange formats and transformation standards. SE 3.0 compilers can adopt proven metamodeling techniques to define FMware IRs and validate FMware programs during compilation.

Self-Adaptive Systems. Self-adaptive systems employ feedback loops to enable runtime adaptation under changing conditions and uncertainty [89, 90]. SE 3.0 compilers face analogous challenges at compilation-time: monitoring FMware behaviour through gold labels, analyzing quality metrics, planning optimization strategies, and executing transformations, all guided by accumulated knowledge about effective approaches. Call for Action 3’s focus on effective search heuristics aligns with self-adaptive systems’ extensive research on adaptation strategies and optimization under uncertainty. Call for Action 6 on efficiency and cost reduction parallels self-adaptive systems’ work on resource-aware adaptation and performance optimization. Call for Action 8 on quality assurance shares challenges with verifying self-adaptive systems under non-deterministic conditions, as both domains must ensure correctness despite inherent uncertainty. Research opportunities include adapting MAPE-K [91] and AWARE [92] patterns for compilation-time use, applying self-adaptive verification techniques to FMware testing, and exploring how compilation-time optimization strategies might inform runtime adaptation decisions in deployed FMware systems (e.g., route runtime calls to different FMs based on their performance on the gold labels during compilation).

Software Product Lines (SPLs). SPL engineering provides systematic methods for managing variability in software families through feature models and automated configuration [93]. The parallel to SE 3.0 compilers is substantial, as both select from a space of possible configurations to optimize desired properties. In SPL, engineers configure features, while in SE 3.0 compilation, compilers select prompts, models, architectural patterns, and other FMware components. Call for Action 3 on search heuristics directly relates to SPL’s extensive literature on feature selection optimization, including genetic algorithms, constraint solving, and multi-objective optimization techniques. Call for Action 8 on quality assurance resonates with SPL testing strategies, particularly sampling techniques for efficiently testing product variants, analogous to testing multiple FMware configurations [94]. Research opportunities include adapting SPL configuration algorithms to FMware optimization and applying variability testing strategies to assess FMware configurations.

These established SE paradigms provide formalized methods and techniques developed over decades that can inform SE 3.0 compiler research. We encourage researchers from these communities to explore these intersections, bringing their expertise to bear on the emerging challenges of compiling FMware.

5.3 Bridging intent compilation and SE practices

While this roadmap focuses on challenges related to intent compilation, *Compiler.next* integrates with established software engineering practices to support complete development workflows. Below, we present a set of RQs that bridge the compilation infrastructure we envision with broader SE practices to operate effectively in the SE 3.0 era.

Requirements Engineering. Call for Action 1's quality constructs for intent representation directly address requirements specification in the SE 3.0 context. Intents serve as a form of high-level requirements that must be validated against stakeholder needs, traced through the compilation process to generate FMware artifacts, and evolved as requirements change. Future research should investigate: (i) how intent specifications can be systematically elicited from stakeholders, (ii) how conflicts between multiple intents can be detected and resolved, and (iii) how requirements traceability can be maintained from intents through compilation to deployed FMware.

Software Testing and Quality Assurance. Call for Action 4's gold labels serve dual purposes: guiding compilation-time optimization and seeding evaluations for deployed FMware. However, compilation-time evaluation represents only a subset of comprehensive testing. Research should investigate: (i) how gold labels used during compilation relate to production test suites, (ii) how test coverage can be systematically expanded beyond the scenarios used during compilation, and (iii) how regression testing can leverage compilation traces to identify when recompilation is necessary.

Continuous Integration and DevOps. Call for Action 6 (efficiency) and Call for Action 10 (compilation trace sharing) enable integration with CI/CD pipelines. Future work should develop practices for: (i) incremental recompilation triggered by changes to intents or system parameters, (ii) caching strategies that work across development teams and build servers, (iii) quality gates that fail builds when compilation cannot meet specified quality thresholds (Call for Action 5), and (iv) automated deployment pipelines that package compiled FMware artifacts with necessary runtime infrastructure.

Software Maintenance and Evolution. SE 3.0 systems will evolve as requirements change, new foundation models become available, and feedback accumulates from production deployments. Call for Action 7 (reproducibility) ensures predictable recompilation when systems must be rebuilt, while the search-based approach naturally supports iterative refinement. Research should investigate: (i) how FMware artifacts are maintained over time as dependencies, (ii) how compilation strategies can adapt to model improvements without requiring complete re-optimization, (iii) how technical debt manifests in compiled FMware (e.g., prompts optimized for outdated models), and (iv) how version control and configuration management apply to FMware artifacts.

Collaborative Development. Modern software development is inherently collaborative, involving multiple developers, designers, domain experts, and stakeholders. Research should investigate how: (i) multiple developers can collaborate on FMware development with shared compilation caches and reusable gold labels, (ii) compilation traces can document design decisions and optimization rationale for knowledge sharing.

Project Management and Process. SE 3.0 projects require planning, resource allocation, and process management adapted to intent-driven development. Research should investigate: (i) how compilation time and cost can be estimated for project planning, (ii) how compilation-time metrics (convergence rates, quality improvements) can inform project status and risk assessment, and (iii) how agile development practices (sprints, iterative refinement, continuous delivery) adapt to compilation-based workflows.

Finally, in our previous paper [26], we focus on challenges for building trustworthy FMware in practice, highlighting issues such as debugging difficulties, prompt fragility, high operational costs, non-deterministic testing, lack of collaboration support, and siloed tooling, in this paper we discuss a set of calls for action, focusing on the challenges around

the research and development of an infrastructure to synthesize FMware programs based on developers' intents. These challenges include developing quality constructs for representing FMware, effective search heuristics, reproducible and cost-efficient compilation, interoperability between compilers, and sharing of compilation traces.

6 Conclusion

AI-native systems, especially those powered by FMs, require continuous adaptation and optimization due to their non-deterministic and evolving nature. Traditional compilers, designed for static environments, cannot handle these real-time adjustments. In this context, *Compiler.next*'s search-based approach allows it to optimize various aspects of cognitive architectures, such as prompts, FM configurations, and system parameters, ensuring that these AI systems remain efficient and effective even as they evolve. By leveraging advanced search-based techniques, *Compiler.next* ensures optimal trade-offs between various objectives such as accuracy, latency, and cost, enhancing the efficiency of AI-powered systems.

Our outlined research and development roadmap presents actionable steps for the SE community to further develop and refine intent compilers, focusing on interoperability, reproducibility, and cost reduction. Ultimately, intent compilers not only pushes the boundaries of AI in SE but also contributes to a future where software creation is more intuitive, accessible, and aligned with human intent.

Disclaimer

Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not reflect the views of Huawei. Also, ChatGPT with GPT-4o was used for copy-editing and table formatting. All experiments, analysis, and results were performed by the authors, who also thoroughly reviewed the final written content for accuracy. This complies with IEEE and ACM policies on AI use in publications.

References

- [1] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*, ser. CHI EA'22. Association for Computing Machinery, 2022.
- [2] S. Jalil, "The transformative influence of large language models on software development," 2023. [Online]. Available: <https://arxiv.org/abs/2311.16429>
- [3] B. Zhang, P. Liang, X. Zhou, A. Ahmad, and M. Waseem, "Practices and challenges of using github copilot: An empirical study," in *Proceedings of the 35th International Conference on Software Engineering and Knowledge Engineering*, ser. SEKE2023, vol. 2023. KSI Research Inc., 2023, p. 124–129.
- [4] X. Zhou, P. Liang, B. Zhang, Z. Li, A. Ahmad, M. Shahin, and M. Waseem, "Exploring the problems, their causes and solutions of ai pair programming: A study on github and stack overflow," *J. Syst. Softw.*, vol. 219, no. C, 2025.
- [5] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?" 2024. [Online]. Available: <https://arxiv.org/abs/2310.06770>
- [6] Z. Yuan, M. Liu, S. Ding, K. Wang, Y. Chen, X. Peng, and Y. Lou, "Evaluating and improving chatgpt for unit test generation," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024.
- [7] Q. Guo, J. Cao, X. Xie, S. Liu, X. Li, B. Chen, and X. Peng, "Exploring the potential of chatgpt in automated code refinement: An empirical study," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024.
- [8] A. E. Hassan, G. A. Oliva, D. Lin, B. Chen, Z. Ming, and Jiang, "Towards ai-native software engineering (se 3.0): A vision and a challenge roadmap," 2024. [Online]. Available: <https://arxiv.org/abs/2410.06107>
- [9] H. Li, H. Zhang, and A. E. Hassan, "The rise of ai teammates in software engineering (se) 3.0: How autonomous coding agents are reshaping software engineering," 2025. [Online]. Available: <https://arxiv.org/abs/2507.15003>
- [10] M. Sclar, Y. Choi, Y. Tsvetkov, and A. Suhr, "Quantifying language models' sensitivity to spurious features in prompt design or: How i learned to start worrying about prompt formatting," in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: <https://openreview.net/forum?id=RJu5lyNXjT>
- [11] B. Cao, D. Cai, Z. Zhang, Y. Zou, and W. Lam, "On the worst prompt performance of large language models," 2024. [Online]. Available: <https://arxiv.org/abs/2406.10248>

- [12] Y. Lu, M. Bartolo, A. Moore, S. Riedel, and P. Stenetorp, “Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity,” in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, S. Muresan, P. Nakov, and A. Villavicencio, Eds. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 8086–8098. [Online]. Available: <https://aclanthology.org/2022.acl-long.556>
- [13] S. Hu, C. Lu, and J. Clune, “Automated design of agentic systems,” 2024. [Online]. Available: <https://arxiv.org/abs/2408.08435>
- [14] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*. USA: Addison-Wesley Longman Publishing Co., Inc., 1986.
- [15] Free Software Foundation, *GCC Online Documentation, Version 14.2.0*, 2024, accessed: 2024-10-10. [Online]. Available: <https://gcc.gnu.org/onlinedocs/gcc-14.2.0/gcc/>
- [16] C. Lattner and V. Adve, “Llvm: A compilation framework for lifelong program analysis & transformation,” in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, 2004, p. 75.
- [17] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, “Tvm: an automated end-to-end optimizing compiler for deep learning,” in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’18. USA: USENIX Association, 2018, p. 579–594.
- [18] OpenXLA Community, “Openxla: Accelerated machine learning compilation,” 2024, accessed: 2024-10-10. [Online]. Available: <https://openxla.org/>
- [19] N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein, J. Montgomery, B. Maher, S. Nadathur, J. Olesen, J. Park, A. Rakhov, M. Smelyanskiy, and M. Wang, “Glow: Graph lowering compiler techniques for neural networks,” 2019. [Online]. Available: <https://arxiv.org/abs/1805.00907>
- [20] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, “Large scale distributed deep networks,” in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS’12. Red Hook, NY, USA: Curran Associates Inc., 2012, p. 1223–1231.
- [21] Y. Zhou, A. I. Muresanu, Z. Han, K. Paster, S. Pitis, H. Chan, and J. Ba, “Large language models are human-level prompt engineers,” in *The Eleventh International Conference on Learning Representations*, 2023. [Online]. Available: <https://openreview.net/forum?id=92gvk82DE>
- [22] C. Fernando, D. S. Banarse, H. Michalewski, S. Osindero, and T. Rocktäschel, “Promptbreeder: Self-referential self-improvement via prompt evolution,” in *Forty-first International Conference on Machine Learning*, 2024. [Online]. Available: <https://openreview.net/forum?id=9ZxnPZGmPU>
- [23] Q. Guo, R. Wang, J. Guo, B. Li, K. Song, X. Tan, G. Liu, J. Bian, and Y. Yang, “Connecting large language models with evolutionary algorithms yields powerful prompt optimizers,” in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: <https://openreview.net/forum?id=ZG3RaNisQ8>
- [24] R. Pryzant, D. Iter, J. Li, Y. Lee, C. Zhu, and M. Zeng, “Automatic prompt optimization with “gradient descent” and beam search,” in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, H. Bouamor, J. Pino, and K. Bali, Eds. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 7957–7968. [Online]. Available: <https://aclanthology.org/2023.emnlp-main.494>
- [25] T. Schnabel and J. Neville, “Prompts as programs: A structure-aware approach to efficient compile-time prompt optimization,” April 2024. [Online]. Available: <https://www.microsoft.com/en-us/research/publication/prompts-as-programs-a-structure-aware-approach-to-efficient-compile-time-prompt-optimization/>
- [26] A. E. Hassan, D. Lin, G. K. Rajbahadur, K. Gallaba, F. R. Cogo, B. Chen, H. Zhang, K. Thangarajah, G. Oliva, J. J. Lin, W. M. Abdullah, and Z. M. J. Jiang, “Rethinking software engineering in the era of foundation models: A curated catalogue of challenges in the development of trustworthy fmware,” in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, ser. FSE 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 294–305. [Online]. Available: <https://doi.org/10.1145/3663529.3663849>
- [27] J. S. Park, J. O’Brien, C. J. Cai, M. R. Morris, P. Liang, and M. S. Bernstein, “Generative agents: Interactive simulacra of human behavior,” in *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST ’23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: <https://doi.org/10.1145/3586183.3606763>
- [28] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, and Y. Cao, “React: Synergizing reasoning and acting in language models,” in *The Eleventh International Conference on Learning Representations*, 2023. [Online]. Available: https://openreview.net/forum?id=WE_vluYUL-X
- [29] Microsoft, “Prompt flow documentation,” <https://microsoft.github.io/promptflow>, 2023, accessed 02-06-2024.
- [30] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, A. H. Awadallah, R. W. White, D. Burger, and C. Wang, “Autogen: Enabling next-gen llm applications via multi-agent conversation,” 2023. [Online]. Available: <https://arxiv.org/abs/2308.08155>
- [31] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, “Hidden technical debt in machine learning systems,” in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS’15. Cambridge, MA, USA: MIT Press, 2015, p. 2503–2511.
- [32] “Open platform for enterprise ai (opea).” <https://opea.dev/>, 2024, accessed: 2024-10-11.
- [33] Amazon Web Services (AWS), “The data flywheel,” accessed: 2024-09-29. [Online]. Available: <https://pages.awscloud.com/EMEA-Data-Flywheel.html>
- [34] O. Khattab, A. Singhvi, P. Maheshwari, Z. Zhang, K. Santhanam, S. Vardhamanan, S. Haq, A. Sharma, T. T. Joshi, H. Moazam, H. Miller, M. Zaharia, and C. Potts, “Dspy: Compiling declarative language model calls into self-improving pipelines,” 2023. [Online]. Available: <https://arxiv.org/abs/2310.03714>
- [35] M. Yuksekgonul, F. Bianchi, J. Boen, S. Liu, Z. Huang, C. Guestrin, and J. Zou, “Optimizing generative ai by backpropagating language model feedback,” *Nature*, vol. 639, pp. 609–616, 2025. [Online]. Available: <https://doi.org/10.1038/s41586-025-08661-4>

- [36] D. C. Schmidt, J. Spencer-Smith, Q. Fu, and J. White, "Towards a catalog of prompt patterns to enhance the discipline of prompt engineering," *Ada Lett.*, vol. 43, no. 2, p. 43–51, jun 2024. [Online]. Available: <https://doi.org/10.1145/3672359.3672364>
- [37] I. Ong, A. Almahairi, V. Wu, W.-L. Chiang, T. Wu, J. E. Gonzalez, M. W. Kadous, and I. Stoica, "Routellm: Learning to route llms with preference data," 2024. [Online]. Available: <https://arxiv.org/abs/2406.18665>
- [38] Q. J. Hu, J. Bieker, X. Li, N. Jiang, B. Keigwin, G. Ranganath, K. Keutzer, and S. K. Upadhyay, "Routerbench: A benchmark for multi-lm routing system," 2024. [Online]. Available: <https://arxiv.org/abs/2403.12031>
- [39] A. Eghbali and M. Pradel, "Crystableu: Precisely and efficiently measuring the similarity of code," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: <https://doi.org/10.1145/3551349.3556903>
- [40] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand, "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*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: <https://doi.org/10.1145/3597503.3639226>
- [41] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *J. Mach. Learn. Res.*, vol. 13, no. null, p. 281–305, Feb. 2012.
- [42] J. Wu, T. Yu, R. Wang, Z. Song, R. Zhang, H. Zhao, C. Lu, S. Li, and R. Henao, "Infoprompt: Information-theoretic soft prompt tuning for natural language understanding," in *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., vol. 36. Curran Associates, Inc., 2023, pp. 61 060–61 084. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2023/file/c01c0da4fe2ef2df98e3f5f5261e2e924-Paper-Conference.pdf
- [43] Langchain-Ai, "Langchain-ai/langchain: Build context-aware reasoning applications." [Online]. Available: <https://github.com/langchain-ai/langchain>
- [44] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.
- [45] J. Blank and K. Deb, "Pymoo: Multi-objective optimization in python," *IEEE Access*, vol. 8, pp. 89 497–89 509, 2020.
- [46] A. Eiben and J. Smith, *Introduction to Evolutionary Computing*, ser. Natural Computing. Springer-Verlag, Berlin, 2003.
- [47] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: language agents with verbal reinforcement learning," in *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., vol. 36. Curran Associates, Inc., 2023, pp. 8634–8652. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2023/file/1b44b878bb782e6954cd888628510e90-Paper-Conference.pdf
- [48] K. Jebari, M. Madiafi *et al.*, "Selection methods for genetic algorithms," *International Journal of Emerging Sciences*, vol. 3, no. 4, pp. 333–344, 2013.
- [49] H. Taherkhani, M. Sepindband, H. V. Pham, S. Wang, and H. Hemmati, "Epic: Cost-effective search-based prompt engineering of llms for code generation," 2024. [Online]. Available: <https://arxiv.org/abs/2408.11198>
- [50] A. E. H. Hao Li, Cor-Paul Bezemer, "Software engineering and foundation models: Insights from industry blogs using a jury of foundation models," 2024.
- [51] E. Levi, E. Brosh, and M. Friedmann, "Intent-based prompt calibration: Enhancing prompt optimization with synthetic boundary cases," 2024. [Online]. Available: <https://arxiv.org/abs/2402.03099>
- [52] X. Wang, J. Wei, D. Schurmans, Q. V. Le, E. H. Chi, S. Narang, A. Chowdhery, and D. Zhou, "Self-consistency improves chain of thought reasoning in language models," in *The Eleventh International Conference on Learning Representations*, 2023. [Online]. Available: <https://openreview.net/forum?id=1PL1NIMMrw>
- [53] U. Anwar, A. Saparov, J. Rando, D. Paleka, M. Turpin, P. Hase, E. S. Lubana, E. Jenner, S. Casper, O. Sourbut, B. L. Edelman, Z. Zhang, M. Günther, A. Korinek, J. Hernandez-Orallo, L. Hammond, E. J. Bigelow, A. Pan, L. Langosco, T. Korbak, H. C. Zhang, R. Zhong, S. O. hEigeartaigh, G. Recchia, G. Corsi, A. Chan, M. Anderljung, L. Edwards, A. Petrov, C. S. de Witt, S. R. Motwani, Y. Bengio, D. Chen, P. Torr, S. Albanie, T. Maharaj, J. N. Foerster, F. Tramèr, H. He, A. Kasirzadeh, Y. Choi, and D. Krueger, "Foundational challenges in assuring alignment and safety of large language models," *Transactions on Machine Learning Research*, 2024. [Online]. Available: <https://openreview.net/forum?id=oVTKOs8Pk>
- [54] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive nlp tasks," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS '20. Red Hook, NY, USA: Curran Associates Inc., 2020.
- [55] O. J. Romero, J. Zimmerman, A. Steinfeld, and A. Tomasic, "Synergistic integration of large language models and cognitive architectures for robust ai: An exploratory analysis," in *Proceedings of the AAAI Symposium Series*, vol. 2, no. 1. AAAI Press, 2023, pp. 396–405.
- [56] T. Sumers, S. Yao, K. Narasimhan, and T. Griffiths, "Cognitive architectures for language agents," *Transactions on Machine Learning Research*, 2024, survey Certification. [Online]. Available: <https://openreview.net/forum?id=1i6ZCvflQJ>
- [57] M. Josifoski, L. Klein, M. Peyrard, N. Baldwin, Y. Li, S. Geng, J. P. Schnitzler, Y. Yao, J. Wei, D. Paul, and R. West, "Flows: Building blocks of reasoning and collaborating ai," 2024. [Online]. Available: <https://arxiv.org/abs/2308.01285>
- [58] M. Harman, A. Mansouri, and Y. Zhang, "Search based software engineering: Trends, techniques and applications," *ACM Computing Surveys*, vol. 45, no. 1, p. Article 11, 2012. [Online]. Available: <https://dl.acm.org/doi/10.1145/2379776.2379787>
- [59] W. Afzal, R. Torkar, and R. Feldt, "A systematic review of search-based testing for non-functional system properties," *Information and Software Technology*, vol. 51, no. 6, pp. 957–976, 2009.
- [60] T. Mariani and S. R. Vergilio, "A systematic review on search-based refactoring," *Information and Software Technology*, vol. 83, pp. 14–34, 2017.

- [61] S. Gulwani, O. Polozov, and R. Singh, "Program synthesis," *Foundations and Trends in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017.
- [62] N. Kant, "Recent advances in neural program synthesis," arXiv preprint arXiv:1802.02353, 2018. [Online]. Available: <https://arxiv.org/abs/1802.02353>
- [63] F. Errica, G. Siracusano, D. Sanvito, and R. Bifulco, "What did i do wrong? quantifying llms' sensitivity and consistency to prompt engineering," 2024. [Online]. Available: <https://arxiv.org/abs/2406.12334>
- [64] C. Hanna, A. Blot, and J. Petke, "Reinforcement learning for mutation operator selection in automated program repair," *Automated Software Engineering*, vol. 32, 2025. [Online]. Available: <https://link.springer.com/article/10.1007/s10515-025-00501-z>
- [65] O. Sahin, M. Zhang, and A. Arcuri, "Causes and effects of fitness landscapes in system test generation: A replication study," *Automated Software Engineering*, vol. 33, no. 8, 2026. [Online]. Available: <https://link.springer.com/article/10.1007/s10515-025-00539-z>
- [66] K. Opsahl-Ong, M. J. Ryan, J. Purtell, D. Broman, C. Potts, M. Zaharia, and O. Khattab, "Optimizing instructions and demonstrations for multi-stage language model programs," 2024. [Online]. Available: <https://arxiv.org/abs/2406.11695>
- [67] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," arXiv preprint arXiv:2107.03374, 2021. [Online]. Available: <https://arxiv.org/abs/2107.03374>
- [68] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, "Program synthesis with large language models," arXiv preprint arXiv:2108.07732, 2021. [Online]. Available: <https://arxiv.org/abs/2108.07732>
- [69] D. Hendrycks, C. Burns, S. Basart, A. Zou, M. Mazeika, D. Song, and J. Steinhardt, "Measuring massive multitask language understanding," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021. [Online]. Available: <https://arxiv.org/abs/2009.03300>
- [70] C. Deng, Y. Zhao, X. Tang, M. Gerstein, and A. Cohan, "Investigating data contamination in modern benchmarks for large language models," in *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT), Long Papers*, 2024, pp. 8706–8719. [Online]. Available: <https://aclanthology.org/2024.naacl-long.482/>
- [71] G. Sahu, P. Rodriguez, I. Laradji, P. Atighehchian, D. Vazquez, and D. Bahdanau, "Data augmentation for intent classification with off-the-shelf large language models," in *Proceedings of the 4th Workshop on NLP for Conversational AI*, B. Liu, A. Papangelis, S. Ultes, A. Rastogi, Y.-N. Chen, G. Spithourakis, E. Nouri, and W. Shi, Eds. Dublin, Ireland: Association for Computational Linguistics, 2022, pp. 47–57. [Online]. Available: <https://aclanthology.org/2022.nlp4convai-1.5>
- [72] H. Jiang, Q. Wu, C.-Y. Lin, Y. Yang, and L. Qiu, "Llmlingua: Compressing prompts for accelerated inference of large language models," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 13 358–13 376.
- [73] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*, 2023, pp. 611–626.
- [74] Y. Leviathan, M. Kalman, and Y. Matias, "Fast inference from transformers via speculative decoding," in *Proceedings of the 40th International Conference on Machine Learning*, vol. 202, 2023, pp. 19 274–19 286.
- [75] F. Bang, "GPTCache: An open-source semantic cache for LLM applications enabling faster answers and cost savings," in *Proceedings of the 3rd Workshop for Natural Language Processing Open Source Software (NLP-OSS 2023)*, L. Tan, D. Milajevs, G. Chauhan, J. Gwinnup, and E. Rippeth, Eds. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 212–218. [Online]. Available: <https://aclanthology.org/2023.nlposs-1.24>
- [76] R. Baghdadi, M. Merouani, M.-H. Leghettas, K. Abdous, T. Arbaoui, K. BENATCHBA, and S. amarasinghe, "A deep learning based cost model for automatic code optimization," in *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica, Eds., vol. 3, 2021, pp. 181–193. [Online]. Available: https://proceedings.mlsys.org/paper_files/paper/2021/file/d9387b6d643efb25132be36f7b908d96-Paper.pdf
- [77] C. Lamb and S. Zacchiroli, "Reproducible builds: Increasing the integrity of software supply chains," *IEEE Software*, vol. 39, no. 2, pp. 62–70, 2022. [Online]. Available: <https://doi.org/10.1109/MS.2021.3073045>
- [78] X. de Carné de Carnavalet and M. Mannan, "Challenges and implications of verifiable builds for security-critical open-source software," in *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC '14)*, 2014, pp. 16–25. [Online]. Available: <https://dl.acm.org/doi/10.1145/2664243.2664288>
- [79] X. de Carné de Carnavalet and M. Mannan, "Challenges and implications of verifiable builds for security-critical open-source software," in *Proceedings of the 30th Annual Computer Security Applications Conference*, ser. ACSAC '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 16–25. [Online]. Available: <https://doi.org/10.1145/2664243.2664288>
- [80] M. Renze and E. Guven, "The effect of sampling temperature on problem solving in large language models," 2024. [Online]. Available: <https://arxiv.org/abs/2402.05201>
- [81] Y. Shi, M. Wen, F. R. Cogo, B. Chen, and Z. M. Jiang, "An experience report on producing verifiable builds for large-scale commercial systems," *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3361–3377, 2022.
- [82] C. Lamb and S. Zacchiroli, "Reproducible builds: Increasing the integrity of software supply chains," *IEEE Software*, vol. 39, no. 2, pp. 62–70, 2022.
- [83] B. Chen, M. Wen, Y. Shi, D. Lin, G. K. Rajbahadur, and Z. M. J. Jiang, "Towards training reproducible deep learning models," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 2202–2214. [Online]. Available: <https://doi.org/10.1145/3510003.3510163>

- [84] D. Chisnall, “The challenge of cross-language interoperability,” *Communications of the ACM*, vol. 56, no. 12, pp. 50–56, 2013.
- [85] M. Grimmer, R. Schatz, C. Seaton, T. Würthinger, and M. Luján, “Cross-language interoperability in a multi-language runtime,” *ACM Transactions on Programming Languages and Systems*, vol. 40, no. 2, p. 8, 2018.
- [86] Y. Liu, C. Chen, R. Zhang, T. Qin, X. Ji, H. Lin, and M. Yang, “Enhancing the interoperability between deep learning frameworks by model conversion,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1320–1330.
- [87] T. Jin, G.-T. Bercea, T. D. Le, T. Chen, G. Su, H. Imai, Y. Negishi, A. Leu, K. O’Brien, K. Kawachiya, and A. E. Eichenberger, “Compiling onnx neural network models using mlii,” 2020. [Online]. Available: <https://arxiv.org/abs/2008.08272>
- [88] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*, 2nd ed., ser. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2017.
- [89] D. Garlan, S.-W. Cheng, and B. Schmerl, “Software architecture-based self-adaptation,” in *Autonomic Computing and Networking*. Springer, 2009, pp. 31–55.
- [90] R. de Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel, D. Weyns, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cukic, R. Desmarais, S. Dustdar, G. Engels, K. Geihs, K. M. Göschka, A. Gorla, V. Grassi, P. Inverardi, G. Karsai, J. Kramer, A. Lopes, J. Magee, S. Malek, S. Mankovskii, R. Mirandola, J. Mylopoulos, O. Nierstrasz, M. Pezzè, C. Prehofer, W. Schäfer, R. Schlichting, D. B. Smith, J. P. Sousa, L. Tahvildari, K. Wong, and J. Wuttke, “Software engineering for self-adaptive systems: A second research roadmap,” in *Software Engineering for Self-Adaptive Systems II*, ser. Lecture Notes in Computer Science. Springer, 2013, vol. 7475, pp. 1–32.
- [91] J. O. Kephart and D. M. Chess, “The vision of autonomic computing,” *Computer*, vol. 36, no. 1, pp. 41–50, 2003.
- [92] B. P. Sanwouo Chekam, C. Quinton, and P. Temple, *Breaking the Loop: AWARE is the New MAPE-K*. New York, NY, USA: Association for Computing Machinery, 2025, p. 626–630. [Online]. Available: <https://doi.org/10.1145/3696630.3728512>
- [93] D. Benavides, S. Segura, and A. Ruiz-Cortés, “Automated analysis of feature models 20 years later: A literature review,” *Information Systems*, vol. 35, no. 6, pp. 615–636, 2010.
- [94] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich, “FeatureIDE: An extensible framework for feature-oriented software development,” *Science of Computer Programming*, vol. 79, pp. 70–85, 2014.

Received 10 January 2025