Context-Aware CodeLLM Eviction for AI-assisted Coding

Kishanthan Thangarajah*, Boyuan Chen*, Shi Chang†, Ahmed E. Hassan‡

*Centre for Software Excellence, Huawei Canada
†Western University, Canada, ‡Queen's University, Canada
cse@huawei.com

Abstract—**AI-assisted coding tools powered by Code Large Language Models (CodeLLMs) are increasingly integrated into modern software development workflows. To address concerns around privacy, latency, and model customization, many enterprises opt to self-host these models. However, the diversity and growing number of CodeLLMs, coupled with limited accelerator memory, introduce practical challenges in model management and serving efficiency. This paper presents CACE, a novel context-aware model eviction strategy designed specifically to optimize self-hosted CodeLLM serving under resource constraints. Unlike traditional eviction strategies based solely on recency (e.g., Least Recently Used), CACE leverages multiple context-aware factors, including model load time, task-specific latency sensitivity, expected output length, and recent usage and future demand tracked through a sliding window. We evaluate CACE using realistic workloads that include both latency-sensitive code completion and throughput-intensive code reasoning tasks. Our experiments show that CACE reduces Time-to-First-Token (TTFT) by 70% and end-to-end (E2E) latency by 37%, while significantly lowering the number of model evictions by 55% compared to state-of-the-art systems. Ablation studies further demonstrate the importance of multi-factor eviction in balancing responsiveness and resource efficiency. This work contributes practical strategies for deploying scalable, low-latency AI coding assistants in real-world software engineering environments.**

Index Terms—**CodeLLM, Model Eviction, Developer Productivity, Coding Assistants, Latency, Performance**

I. INTRODUCTION

CodeLLMs are a distinct category of Large Language Models (LLMs) specifically designed for coding tasks. They are trained on a vast number of repositories of code artifacts and related documentation [1], [2]. AI-assisted coding with CodeLLMs has significantly reshaped software development, boosting productivity by reducing both coding time and errors [3]–[8]. AI-assisted coding enables developers to concentrate on higher-level design and problem-solving tasks rather than boilerplate implementation. With the rapid release of new models, the number and variety of available CodeLLMs continue to grow. However, not all models perform equally across different coding scenarios. Some CodeLLMs excel at generating code, while others are more effective at understanding or reasoning through complex logic, especially when tailored to specific programming languages [9]–[14]. Likewise, performance can vary across application domains such as web development, mobile, blockchain, game, distributed systems, or networking, where certain CodeLLMs demonstrate domain-specific strengths [15]–[18]. As a result, developers now have the flexibility to choose from a wide range of models optimized for their particular use cases, making model selection increasingly critical to achieving high performance and relevance in AI-assisted coding workflows.

To boost developer productivity, many organizations are integrating AI-assisted coding tools powered by CodeLLMs [4], [6], [19], [20]. Enterprise software across diverse domains often rely on developers working with multiple programming languages [21]–[26]. For instance, a typical full-stack application may involve front-end developers using JavaScript or TypeScript, back-end developers writing services in Java or Python, DevOps engineers authoring infrastructure code in Go, and automation tasks scripted in Bash. This diversity is driven by the specialization of tools and languages for different layers of the software stack. Consequently, an effective CodeLLM-based assistant must support multiple languages and provide fast, high-quality responses across a broad spectrum of coding tasks and codebases.

In addition, due to growing privacy and security concerns [27]–[29], these organizations are increasingly opting to self-host CodeLLMs, including fine-tuned ones on proprietary data, rather than relying on external model APIs. While self-hosting specialized CodeLLMs enhances privacy control and allows customization to meet enterprise-specific requirements, it introduces a new challenge: managing limited in-house accelerator resources in environments where the total number of CodeLLMs needed in use far exceeds available resources. As developers expect near-instantaneous feedback, especially for latency-critical tasks such as code completion, reducing the cold-start latency of LLM requests becomes essential. One common approach for existing LLM serving systems is to enable model multiplexing.

Model multiplexing refers to concurrently hosting and dynamically managing multiple models on shared accelerator resources. This approach optimizes resource usage by loading, unloading, and routing inference requests among various model instances according to real-time demands. Frameworks such as Ray Serve [30], Ollama [31], ModelMesh [32], and other state-of-the-art systems [33] exemplify this approach by dynamically loading, unloading, and routing inference requests among multiple model instances within constrained accelerator memory. These systems aim to maximize resource utilization and throughput, typically employing simple heuristics, such as Least Recently Used (LRU), for model eviction deci-

sions. However, without context-aware strategies, they often encounter unnecessary data movement and cold-start latencies that negatively impact developer experience in latency-sensitive tasks. This makes efficient model eviction strategies a key requirement to ensure the most relevant models are loaded dynamically, cold starts are minimized, and both latency and resource costs are optimized.

In this paper, we introduce **C**ontext-**A**ware **C**odeLLM **E**viction (CACE), a dynamic and context-aware model eviction policy designed specifically for efficient and responsive CodeLLM serving in enterprise environments. Unlike traditional eviction methods, such as LRU, CACE integrates multiple context-aware factors, including model loading times, recent usage patterns, task-specific latency requirements, and predicted future developer requests. Through a comprehensive evaluation using realistic developer workloads comprising both latency-sensitive code completion tasks and throughput-intensive code reasoning tasks, we demonstrate that CACE significantly reduces Time-To-First-Token (TTFT) and end-to-end (E2E) latency, while minimizing unnecessary data movement (i.e., model reload). Additionally, we conduct extensive ablation studies to isolate and analyze the relative impact of each individual factor, providing clear insights into the effectiveness of our multi-factor eviction strategy.

Specifically, we explore the following research questions (RQs):

- **RQ1:** Can CACE reduce the overall number of model evictions compared to conventional eviction strategies in a self-hosted CodeLLM environment?
- **RQ2:** How effectively does CACE reduce TTFT and E2E latency in a self-hosted CodeLLM environment?
- **RQ3:** What is the individual impact of each context-aware factors within the CACE eviction policy?

To ensure transparency and reproducibility, we intend to make our solution publicly available [34]. The open-source release will allow reviewers and researchers to evaluate and build upon our work.

The remainder of this paper is structured as follows: Section II presents the background and motivation for efficient, latency-sensitive CodeLLM serving in enterprise environments. Section III highlights key operational challenges and industry trends. Section IV describes the detailed design of our proposed CACE approach, including the eviction policy and algorithmic factors. Section V outlines the experimental methodology, datasets, workload patterns, and evaluation metrics. Section VI discusses about insights from the results potential future research directions. Section VII focus on related work, positioning our contributions within the landscape of existing solutions, including model multiplexing and memory management strategies. Section VIII explores potential threats to validity and limitations of our approach. Finally, Section IX concludes the paper and outlines promising directions for future research.

II. BACKGROUND AND MOTIVATION

A. Latency in AI-Assisted Coding

AI-assisted coding powered by CodeLLMs significantly enhances developer productivity through intelligent suggestions for tasks such as code completion, test generation, and codebase understanding. In particular, within interactive development environments (IDEs), developers have stringent expectations regarding responsiveness and latency. Even minor delays in tasks such as code completion can disrupt workflow continuity, making latency a critical barrier to the effective adoption of coding assistants [20]. Systems such as Code-Compose at Meta [6] address this challenge by using a fine-tuned version of CodeLlama, delivering multi-line suggestions within milliseconds, aligning closely with developers' fast-paced workflow. Similarly, DeepVulGuard [35] employs a small finetuned CodeBERT model to rapidly identify vulnerabilities. The Cursor team has used a Mixture-of-Expert based model to handle large codebases and long input context length while reducing inference latency [36] and lastly, Jet-Brains IDEs integrate lightweight, language-specific models optimized for low-latency, line-level code completions [19]. These industry efforts collectively underscore the importance of minimizing inference latency to meet developer expectations for real-time assistance.

B. Motivations for Self-Hosting CodeLLMs

Enterprises deploying AI-assisted coding tools face additional considerations beyond latency, particularly about data privacy [28], [37], security [29], [38], and operational control. Due to stringent privacy regulations such as GDPR [39] and the necessity of safeguarding proprietary source code [27], many organizations prefer self-hosted CodeLLMs rather than relying on external API services. Self-hosting allows enterprises to control data exposure risks and reduces latency by eliminating network calls to external servers, thus ensuring more responsive and secure interactions for developer tasks.

C. Selecting the Right CodeLLM for the Task

An inherent complexity arises from the diverse range of coding tasks and the availability of numerous CodeLLM variants optimized for specific use cases. Small, specialized CodeLLMs, fine-tuned per programming language (e.g., Python, Java, C++), typically support latency-critical tasks like code completion. Conversely, larger models are utilized for computationally intensive reasoning tasks, which, although less latency-sensitive, require greater computational resources. Model effectiveness varies significantly across languages and domains. For example, Qwen2.5-Coder-32B excels in Python and C#, yet underperforms in Java, C++ and PHP [40], while Codestral 20B surpasses larger models in specific languages such as SQL and Swift [41]. For code reasoning tasks, such as code output reasoning, WizardCoder 15B performs well for Java and Scala, Starcoder2 15B performs well for Swift, Codegen 6B for Shell, and CodeLlama 34B performs well for Rust, C++, and Julia languages [11]. Such variability

necessitates providing developers with a choice of models tailored explicitly to their tasks and development contexts.

D. Necessity of Dynamic CodeLLM Eviction

Accommodating all these models simultaneously on limited accelerator resources (GPUs/TPUs/NPUs) is impractical due to both cost and accelerator memory constraints. Traditional model serving approaches with model multiplexing, often rely on simple eviction policies, such as Least Recently Used (LRU), which do not adequately account for the unique characteristics of coding tasks, including latency sensitivity, model load time, and anticipated developer usage patterns. Consequently, naive eviction strategies can frequently trigger unnecessary model reloads, introducing disruptive cold-start latencies and diminishing the efficiency and responsiveness of the development workflow.

Addressing these gaps requires a novel, context-aware approach capable of dynamically prioritizing model evictions by considering multiple factors: recent usage patterns, task-specific latency requirements, model load times, and anticipated developer requests.

III. CHALLENGES

Current production-grade LLM serving systems, such as Ray Serve [30], Ollama [31], ModelMesh [32], and state-of-the-art research frameworks like FaaSwap [33] primarily rely on simplistic LRU policies for model eviction between host and accelerator memory. While LRU is straightforward and easy to implement, it often triggers unnecessary model evictions, as it fails to deal with the following challenges. As a result, it leads to increased data movement (i.e., the excessive model weights) and elevated latency, particularly in TTFT, which negatively impacts developer experience.

A. Minimizing Data Movement Overheads

Accelerators such as GPUs, TPUs, and NPUs are expensive to provision and maintain, resulting in limited availability within enterprises. Simultaneously, the number and diversity of CodeLLM variants, including multiple fine-tuned models tailored for different tasks and programming languages continues to grow. In modern enterprise software development, developers across various roles, such as front-end engineers, back-end engineers, and DevOps teams often require access to different models concurrently to support their workflows. However, the total accelerator memory is typically insufficient to accommodate all required models at once, necessitating frequent model evictions.

Consequently, model multiplexing between host and accelerator memory becomes essential. Traditional eviction strategies, such as LRU, rely solely on access recency and overlook key model characteristics. Importantly, CodeLLMs vary widely in size and load time, and indiscriminate eviction can lead to substantial cold-start delays. For example, the cost of evicting a large model like DeepSeek-R1 is different from a specialized fine-tuned Qwen-Coder-3B. An effective eviction policy must therefore consider model size, load time, and predicted usage frequency to minimize data movement and ensure timely model availability.

B. Managing Task-Specific Latency Requirements

Different types of AI-assisted coding tasks impose distinct latency requirements that directly impact the developer experience. Code completion tasks are highly sensitive to TTFT latency, as developers expect rapid, near-instantaneous suggestions while typing. These tasks typically involve short output lengths and demand sub-second responsiveness to maintain an uninterrupted development flow. In contrast, code reasoning tasks, such as multi-line code synthesis or complex bug explanation, can tolerate higher E2E latency due to the inherently longer outputs and deeper computation involved. However, even for E2E-sensitive tasks, response time must remain within an acceptable threshold to avoid disrupting the developer's focus or productivity.

With multiple CodeLLMs competing for constrained accelerator memory, it becomes crucial to design model retention and eviction policies. That is to say, when accelerator memory is full and there is an incoming model request, we have to strategically select an existing loaded model to evict. For example, naively evicting a model serving a TTFT-critical task in favor of a less urgent one can significantly degrade the real-time responsiveness required for code completion. Conversely, excessive eviction of large reasoning models may inflate E2E latency due to high reload overhead.

IV. CONTEXT-AWARE CODELLM EVICTION (CACE)

To enhance overall model eviction performance in serving systems, CACE is designed to minimize the number of model evictions, thereby reducing TTFT and E2E latencies. We first describe the system design in Section IV-A, and then further break down the details on the core factors that impact the eviction strategy in Section IV-B.

A. CACE System Architecture and Serving Pipeline

The CACE solution is integrated into a single-node CodeLLM serving architecture designed to meet the demands of enterprise software development environments. These environments require both low-latency responses for interactive tasks like code completion and support for long-running reasoning tasks. CACE serves as an intelligent, context-aware model eviction technique to manage accelerator memory efficiently under dynamic workloads.

The serving pipeline, as described in Figure 1, begins with a model registration step ①, where developers register a new instance of CodeLLMs into the system. Upon registration, the **Profiler** loads the model in a staging area and records static metrics such as model load time and memory usage. These metrics are stored in the **Metadata Store**, which is backed by a persistent database. This is an offline step before the model is made available for serving ②.

At runtime (online), the coding task requests are received ③ by the **Orchestrator**. This will query the **Metadata Store** to retrieve relevant profiling data for models residing
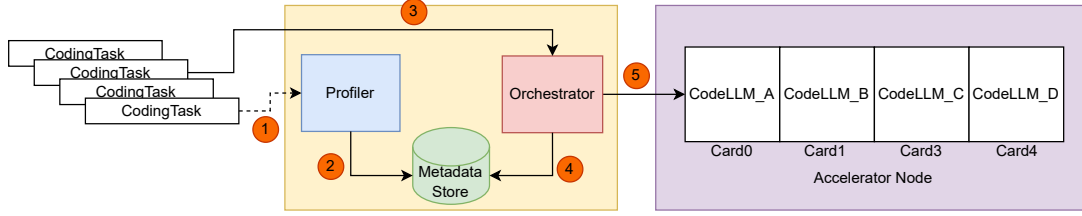
Fig. 1. The Overall Workflow of CACE

in the accelerator memory, which can be evicted ④. It then combines this information with real-time request context, such as task latency sensitivity (e.g., TTFT vs. E2E), expected output token length, and recent model usage, to compute an eviction score for each model currently residing in the accelerator memory. This score is used to guide model eviction decisions (described in Section IV-B), ensuring that latency-critical and frequently used models are prioritized, while less critical models are evicted to make room for incoming ones. The **Orchestrator** makes sure that the requested CodeLLM is loaded in the accelerator and ready before the request is dispatched ⑤. By decoupling profiling from runtime execution and leveraging persistent metadata access, CACE enables task-sensitive, context-aware model management optimized for resource-constrained accelerator environments.

Algorithm 1 CACE Algorithm

1: $L1 \leftarrow$ Models in Accelerator memory
2: $L2 \leftarrow$ Models to be loaded
3: $w \leftarrow$ Sliding window length
4: **procedure** CACE($L1, L2, len$)
5:     Remove duplicates from $L2$
6:     Sort $L1$ by last accessed time
7:     $S \leftarrow \{\}$ ▷ Eviction scores
8:     **for all** $m \in L1$ **do**
9:         $o \leftarrow$ GETEXPECTEDOUTPUTTOKENS(m)
10:         $S[m] \leftarrow$ CALCEVICTIONSCORE($m, L2, w, o$)
11:     **end for**
12:     $m_{evict} \leftarrow \arg\max S[m]$
13:     EVICT(m_{evict})
14: **end procedure**

Unlike traditional approaches that rely solely on LRU policy, our method integrates multiple sources of information to guide model eviction decisions. We will break down our method next.

B. Eviction Score Calculation

To guide evictions in our evicting algorithm, we introduce an *Eviction Score* that combines multiple factors influencing model retention. The score is designed with a few considerations. First, it incorporates anticipated future load requests by monitoring the incoming request queue from the serving system, enabling prioritization of models likely to be required imminently. This information is captured as a list

L2 (models to be loaded). Second, this knowledge pertains to the set of models already loaded into accelerator memory. This information is captured as a list L1 (models already loaded in accelerator memory). We maintain a sliding window containing a fixed number of user-requested and currently loaded models within this window. As new requests arrive, the window dynamically slides to update the set of relevant models. Third, the algorithm accounts for model load times, assigning higher weights to larger models whose reloading incurs greater latency penalties. Finally, unlike previous methods, our approach integrates task criticality by evaluating the expected output token count associated with each request, thereby prioritizing models critical for latency-sensitive tasks such as code completion. By synthesizing these factors, the proposed policy aims to optimize evicting decisions for improved responsiveness and resource utilization.

Let:

- t: last used time of the model in L1 (timestamp),
- l: model load time (in seconds),
- w: sliding window length,
- i: index position of the model in L2,
- o: expected number of output tokens,
- w_1: weighting factor for output tokens.

$$\text{EvictionScore} = \frac{1}{1 + \ln t} + \frac{1}{1 + \frac{l}{100}} + \frac{i}{w} + w_1 \cdot o$$

Based on these design choices, the eviction score incorporates four key factors, **P1 (Recency)**, **P2 (Reloading Cost)**, **P3 (Future Demand)**, and **P4 (Task Criticality)**, each contributing to a more informed and context-sensitive eviction policy, the rationale behind the formula are explained as follows:

- **P1: Recency (Last Used Time in L1)**
  This factor captures how recently a model was accessed in the accelerator memory. A model used more recently will have a smaller t (last used time of the model in L1) value, yielding a larger inverse score contribution ($\frac{1}{1+\ln(t)}$ compresses the time since last used time t). This protects frequently accessed models from eviction. In contrast, models with older access times contribute less to the score, increasing their likelihood of being evicted.
- **P2: Reloading Cost (Model Load Time)**
  This factor reflects the time required to reload a model into the accelerator memory, using l (model load time) as

a proxy ($\frac{1}{1+\frac{l}{100}}$ turns cold-load time l into a 0–1 value). Models with shorter load times (typically smaller models) yield higher contributions to the score, indicating they are cheaper to evict. Models with longer load times are penalized less, making their eviction less favorable due to higher reloading overhead.

- **P3: Future Demand (Sliding Window Forecast)**
  This factor estimates the urgency of future model usage based on the position of the model in the future request window. It is calculated as the fraction between model's i (index position of the model in L2) and the w (which is the sliding window length). A smaller value for this factor implies the model is going to be reuses soon, resulting in increased retention priority. Models needed further in the future contribute less to the score and are more likely to be evicted. If the model is not present in the L2, then a default value of 1 is given.

- **P4: Task Criticality (Expected Output Tokens)**
  This factor reflects task sensitivity by using the expected number of output tokens, weighted by a tunable parameter w_1 (weights the output-token budget so models needed for short-deadline completions stay resident). Larger token counts that are common in E2E-centric tasks, like code reasoning, contribute more to the score. Lower values indicate latency-critical tasks such as code completion, thereby prioritizing models that serve those tasks for retention.

By integrating these four factors, the eviction policy of CACE goes beyond simple recency-based strategies to make context-aware decisions that reflect both system constraints and workload context. This aims to provide more efficient model retention and improved performance under constrained accelerator resources.

V. EVALUATION

In this section, we first describe the case study setup in Section V-A, then we answer the three RQs respectively (Sections V-B to V-D).

A. Case Study Setup

All experiments are conducted on an Ubuntu server equipped with an AMD Ryzen Threadripper PRO 5975WX (64 cores) and four accelerators with 48 GB of memory. We used vLLM as the inference engine and dedicated a single card for each model instance. At the same time, only four model instances will be loaded simultaneously with four accelerators. We used Ray Serve [30] as the baseline system, which implements an LRU-based model multiplexing strategy, similar to Ollama and FaaSwap. We then evaluate CACE against LRU with two benchmark datasets:

- **McEval:** A benchmark for multilingual code completion tasks [12].
- **CRUXEVAL-X:** A benchmark for multilingual code reasoning tasks [11].

For code completion tasks that are highly sensitive to TTFT, we assign a small CodeLLM with 500M parameters per

programming language, following the approach of Semenkin et al. [19], who adopted fine-tuned and lightweight models for individual languages at JetBrains. For code reasoning tasks, which are more sensitive to E2E latency, we utilized large CodeLLMs with 7B parameters for each programming language to better support complex analysis and multi-step inference. We selected eight programming languages based on their consistent ranking in industry-recognized sources such as the RedMonk and TIOBE indexes [42], [43]: Java, Python, C++, C, Go, Rust, C#, and JavaScript. Accordingly, our experimental setup includes eight small CodeLLMs and eight large CodeLLMs, one of each per language.

To evaluate the effectiveness of our approach, we define three workload request arrival patterns that reflect realistic usage scenarios [44]–[46] in enterprise-grade software development environments:

- **Pattern 1: Uniform.** This pattern simulates an equal distribution of code completion and code reasoning requests (50% each) across eight programming languages. It serves as a balanced baseline for evaluating the CACE eviction policy under non-skewed conditions.
- **Pattern 2: IDE Heavy.** An imbalanced workload with 70% code completion and 30% code reasoning requests. This is to simulate a real-world use case where developers often call code completion in the IDE's compared to code reasoning tasks.
- **Pattern 3: Language Popularity Skewed.** A variation of Pattern 2, and reflects real-world language usage trends. Within the 70%-30% allocation for each coding task type, 20% of requests are allocated to Java, 20% to Python, 20% to C++, 20% JavaScript, and the remaining 20% are distributed among Go, Rust, C, and C#. This distribution is informed by recent programming language popularity data. Specifically, the RedMonk [42] and TIOBE [43] lists Python, C++, and JavaScript as the top-ranked languages, with Go, Rust, and JavaScript also featuring within the top 20.

All three usage patterns model an enterprise setting in which many developers, with their IDEs, share a self-hosted CodeLLM service by the enterprise. These patterns enable us to assess the adaptability and efficiency of CACE across both balanced and skewed usage scenarios, ensuring its applicability to a range of real-world development workloads.

Lastly, for every workload pattern, we synthetically generate request streams using a Poisson arrival distribution over a fixed 30-second time window. Each request is then labeled according to the target pattern (Uniform, IDE Heavy, or Popularity Skewed). Before dispatching the stream to the serving stack, we shuffle all requests to avoid bursty language or task-specific clusters and to ensure a realistic, interleaved mix of code-completion and code-reasoning tasks within the 30-second interval.

Notation for CACE *Variants:* To systematically evaluate the impact of each factor in our eviction policy, we define the following notation to distinguish between different variants of CACE:

- **CACE****: The full eviction strategy with all four factors enabled (Recency (P1), Reloading Cost (P2), Future Demand (P3), and Task Criticality(P4)).
- **CACE^{-P1}**: Variant with **Recency** disabled.
- **CACE^{-P2}**: Variant with **Reloading Cost** disabled.
- **CACE^{-P3}**: Variant with **Future Demand** disabled.
- **CACE^{-P4}**: Variant with **Task Criticality** disabled.

These variants are used across RQs to analyze the contribution of each factor to eviction effectiveness, latency reduction, and cache performance.

B. RQ1: Can CACE reduce the overall number of model evictions compared to conventional strategies in a self-hosted CodeLLM environment?

1) Hypothesis: We hypothesize that CACE will significantly reduce the total number of model evictions compared to traditional LRU-based eviction strategies by avoiding unnecessary evictions and retaining frequently used models in accelerator memory. While this research question focuses on eviction behavior, a reduction in model evictions may also contribute to improved latency characteristics by lowering the frequency of cold-start loading events, which will be discussed in the next RQ.

2) Results: For this RQ, we use the setup as described in Section V-A. In addition, we disable the Task Criticality weighting factor in the eviction score equation to isolate and evaluate the performance of model eviction alone and to see what the other three factors can perform on their own.

We evaluate system performance based on the following metrics:

- **Model Cache Hit Rate:** Among the total number of requests being served, the proportion of requests served from models already loaded in accelerators.
- **Model Load Overhead:** Measure of time spent on loading models (highlighting the unproductive time spent on loading).
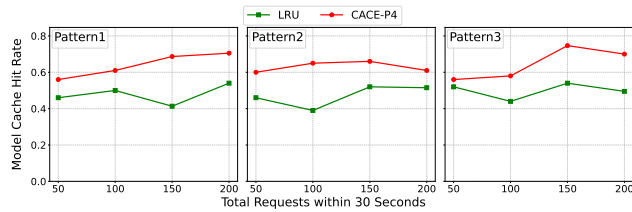
Fig. 2. Comparing Model Cache Hit Rate against different request load patterns of CACE^{-P4} and LRU for RQ1.

Figures 2 and 3 present the comparison between CACE^{-P4} (with Task Criticality disabled) and the baseline LRU strategy under three distinct workload patterns. Figure 2 demonstrates that CACE^{-P4} consistently achieves higher Model Cache Hit Rate across all workloads compared to LRU.

Similarly, Figure 3 reveals that CACE^{-P4} significantly reduces cumulative model load overhead compared to LRU. For instance, under Pattern 2, CACE^{-P4} reduces the total
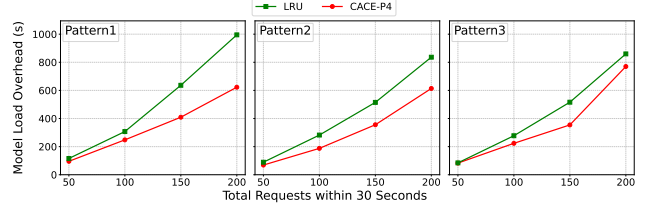
Fig. 3. Comparing Model Load Overhead against different request load patterns of CACE^{-P4} and LRU for RQ1.

Model Load Overhead by about **27%**, and under Pattern 3, by about **31%**.

Findings: Even without the Task Criticality factor, CACE^{-P4} reduces Model Load Overhead by up to 31% and increases Model Cache Hit Rate by up to 41% compared with a pure LRU policy. These improvements shows the effectiveness only by considering Recency, Reloading Cost, and Future Demand in eviction decisions to reduce unnecessary data movements.

Implication: A recency-only strategy is insufficient in multi-model CodeLLM serving. Designers should incorporate at least model reload cost and short-term demand forecasting, and the choice of additional factors, such as latency, can then be aligned with performance goals. We further explore this in the next research question.

C. RQ2: How effectively does CACE reduce TTFT and E2E latency in a self-hosted CodeLLM environment?

AI-assisted coding tasks such as inline code completions are highly sensitive to TTFT latency, as developers expect near-instantaneous responses. In addition, complex tasks such as large scale code refactoring can tolerate long TTFT but still benefit from reduced E2E latency. Traditional model eviction strategies like LRU are agnostic to task specific latency requirements, which is unique in CodeLLM generations. For instance, LRU might fail to accommodate TTFT when deciding to evict a model just because it is not the most recently used model.

In this RQ, we evaluate CACE's eviction policy, which incorporates the Task Criticality factor, such as expected output length and latency sensitivity, by dynamically monitoring the request queue. This approach enables CACE to prioritize retaining models that are critical to latency-sensitive tasks, thereby reducing the frequency of cold starts and improving both TTFT and E2E latency metrics.

1) Hypothesis: We hypothesize that CACE will significantly reduce TTFT for latency-sensitive tasks such as code completion, and lower E2E latency for tasks like code reasoning, at the same time not compromising on the model eviction performance, when compared to conventional LRU-

based eviction strategies. This improvement is expected to manifest in:

- Reduced TTFT for short-response tasks.
- Reduced E2E latency for complex, long-response tasks.
- Fewer model evictions. Expect this to be lower than that in RQ1.

2) Results: We use the same experiment setup as described in Section V-A with the same three request arrival patterns under realistic usage scenarios to test latency sensitivity. We then evaluate system performance based on the following metrics:

- **TTFT:** Time-to-first-token for code completion tasks.
- **E2E Latency:** Total end-to-end latency for code reasoning tasks.
- **Model Cache Hit Rate:** Percentage of requests served without requiring a model reload.

TTFT and E2E latency are evaluated to showcase the benefits of our approach, while Model Cache Hit Rate is used to explain the rationale behind the improvements.
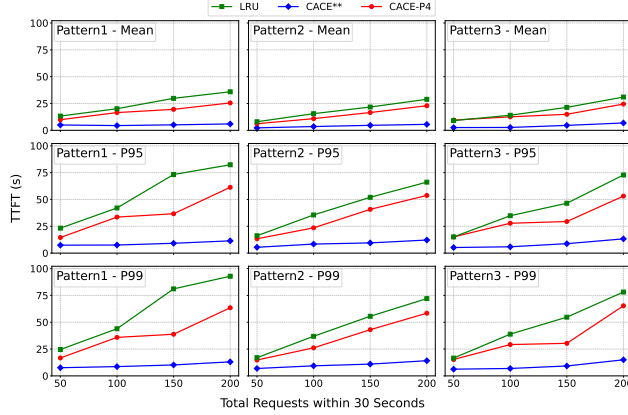

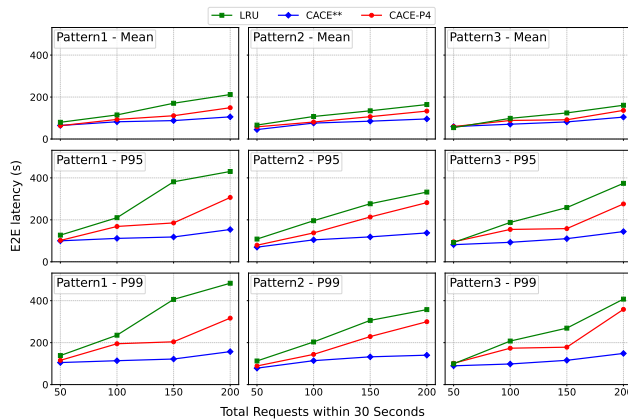Fig. 4. Comparing TTFT against different request load patterns of CACE**, CACE^{-P4}, and LRU for RQ2.

Fig. 5. Comparing E2E latency against different request load patterns of CACE**, CACE^{-P4}, and LRU for RQ2.
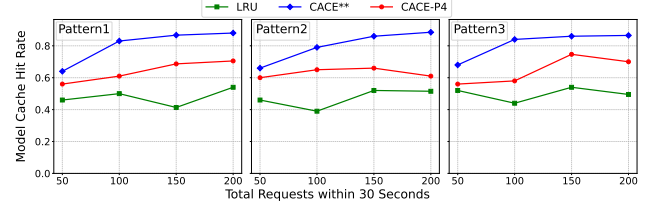

Fig. 6. Comparing Model Cache Hit Rate against different request load patterns of CACE**, CACE^{-P4}, and LRU for RQ2.

Figures 4–6 compare three eviction strategies: baseline LRU, CACE^{-P4} (without Task Criticality), and CACE** (all four factors enabled), across three workload patterns. The results confirm that adding the Task Criticality factor improves latency metrics while preserving the eviction benefits observed in RQ1.

Figures 4–6 show that enabling Task Criticality (CACE**) significantly improves latency and cache efficiency across all workload patterns. Compared to LRU, CACE** lowers mean TTFT by up to 70%, and tail latencies by as much as 80%, with similar improvements in E2E latency, up to 37% on average and 59% at the 99th percentile. These latency gains are supported by higher Model Cache Hit Rates, rising to 0.85–0.86 under mixed workloads and reducing model evictions by 55%. In all cases, CACE^{-P4} (without P4) performs better than LRU, but CACE** provides the most consistent and substantial benefits.

> **Findings:** Including task–criticality in the eviction score reduces TTFT by up to 70 % and E2E latency by up to 37 %, while boosting Model Cache Hit Rate to 0.86 and halving model load overhead compared with a pure LRU eviction.
>
> **Implications:** Latency-critical CodeLLM serving systems should prioritize models based on task importance (e.g., expected output length) in addition to recency and reload cost before evicting them from accelerator memory. CACE addresses this by incorporating a Task Criticality factor, which enhances performance for both interactive and long-running tasks. This suggests that multi-factor eviction policies tailored to the latency characteristics of heterogeneous workloads are more effective than single-factor baselines. Future work should investigate how each factor in the eviction score contributes to these gains and whether simpler subsets can produce similar benefits with lower complexity, which motivates the ablation analysis in RQ3.

D. RQ3: What is the individual impact of each context- aware factor within the CACE eviction policy?

While RQ1 and RQ2 evaluate the efficacy of CACE's eviction scoring strategy under various workload patterns, it is crucial to understand the contribution of each factor within the

eviction score formulation. A systematic ablation study allows us to isolate and quantify the contribution of each factor to the eviction score by removing one at a time and evaluating the impact on key performance metrics.

1) Hypothesis: We hypothesize that disabling individual factors of the eviction score will degrade performance, with certain factors (e.g., Task Criticality) contributing more significantly to metrics like Model Cache Hit Rate and TTFT/E2E latency reduction than others.

We expect:

- A decrease in Model Cache Hit Rate when key factors are omitted.
- An increase in both TTFT and E2E latency due to sub-optimal eviction decisions when key factors are omitted.
- Varying degrees of performance degradation depending on the removed factors.

2) Results: We reused the same system architecture, datasets, and three workload patterns as described in Section V-A to ensure consistency.

As for the request arrival pattern, we use Pattern 3 (Language Popularity Skewed Distribution), as this reflects the real-world language usage with **IDE Heavy** workload. In each ablation variant, we remove one factor from the eviction score computation while keeping the other three active. This setup enables a comparative analysis to quantify the individual contribution of each factor toward reducing model evictions and improving model retention in accelerator memory. Since model eviction directly influences the frequency of cold starts, we also examined the downstream effect on latency, specifically, TTFT for code completion tasks and E2E latency for code reasoning tasks. We captured the same evaluation metrics as RQ2 (TTFT, E2E Latency, and Model Cache Hit Rate).
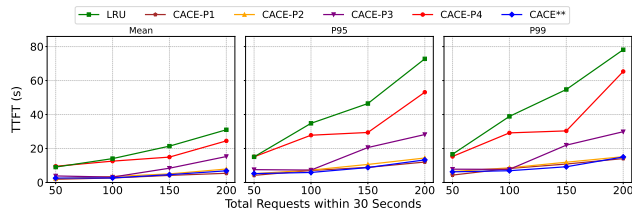

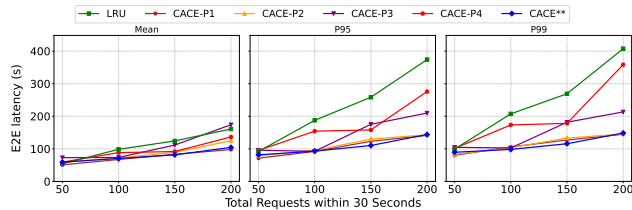Fig. 7. Comparing TTFT of different variants of CACE.

Fig. 8. Comparing E2E latency of different variants of CACE.

Figures 7, 8, and 9 compare CACE** with four ablated variants, CACE^{-P1}, CACE^{-P2}, CACE^{-P3}, and CACE^{-P4}, each obtained by disabling one factor in the eviction score.
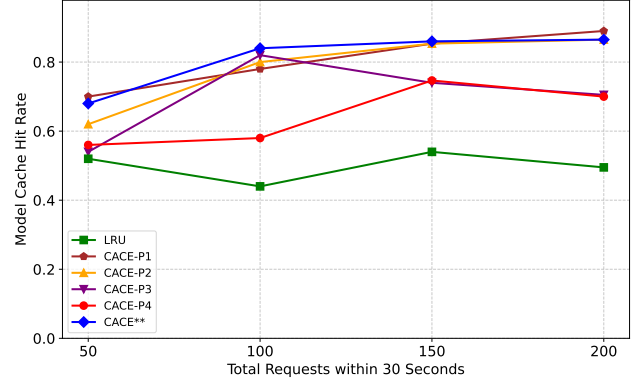

Fig. 9. Comparing Model Cache Hit Rate of different variants of CACE.

Disabling the Task Criticality (CACE^{-P4}) leads to the largest degradation in Model Cache Hit Rate, dropping from **0.85** (CACE**) to **0.7** (a **24%** decrease), and causes the worst latency penalty, with P95 TTFT rising by **72%** and P95 E2E latency increasing by **47s**. Removing Future Demand forecasting (CACE^{-P3}) also substantially harms performance, reducing Model Cache Hit Rate by **22%** and increasing P95 TTFT by **15s**. In contrast, eliminating Recency (CACE^{-P1}) or Reload Time (CACE^{-P2}) has only minor effects on both Model Cache Hit Rate and latency. These results underscore that Task Criticality and Future Demand are the dominant factors in effective eviction decisions.

> **Findings:** Ablation results reveal that the Task Criticality factor (P4) is the most influential in improving Model Cache Hit Rate and reducing latency. Disabling it leads to a significant drop in Model Cache Hit Rate and a substantial increase in P95 TTFT, more than any other factor. Future Demand forecasting (P3) is the second most impactful, while recency (P1) and model load time (P2) have comparatively smaller effects.
> **Implications:** The eviction score benefits most from incorporating Task Criticality and Future Demand factors. While recency and reload cost offer modest gains, prioritizing models based on expected output length and future reuse is key to effective model retention in the accelerator memory. Designers can focus on these high-impact factors to optimize performance with manageable complexity. Future work could explore adaptive weighting or dynamic tuning to improve CACE's responsiveness under changing workloads.

VI. DISCUSSION AND FUTURE DIRECTIONS

The experimental insights from our evaluation highlight several promising directions for enhancing the design and implementation of model eviction policies for self-hosted CodeLLM serving systems. Here we outline key areas for future exploration.

A. CPU Memory as a Secondary Cache Layer

Our current CACE eviction strategy manages models directly between disk storage and accelerator memory. Introducing CPU memory as an intermediate caching layer could significantly reduce reload latencies since model transfers from CPU RAM to accelerator memory are substantially faster than disk-to-accelerator transfers. Models frequently evicted due to accelerator constraints could remain resident in CPU memory, ensuring quicker reload times upon subsequent requests. However, effectively integrating this additional caching layer would introduce new complexities such as accounting for CPU cache capacity, model sizes, and expected reuse frequency. Future work could explore adaptive heuristics or machine learning-based strategies to balance eviction decisions between accelerator, CPU, and disk storage efficiently.

B. CACE *for Heterogeneous Accelerator Clusters*

Extending CACE to distributed heterogeneous clusters containing diverse accelerator hardware represents another promising direction. In heterogeneous environments, less urgent or resource-intensive models could temporarily migrate to less performant accelerators or idle nodes, enhancing overall resource utilization. This approach requires sophisticated profiling of inter-node network latency, bandwidth costs, model load time variations, and cluster topology. Future iterations of CACE could incorporate dynamic scheduling strategies informed by real-time network conditions, accelerator capabilities, and task latency sensitivity. Such an approach has the potential to significantly minimize cold-start latencies and improve resource allocation efficiency across heterogeneous distributed deployments.

C. Improved Predictions of Task Criticality through Token Length Estimation

Accurate prediction of token generation lengths is crucial to the precision of Task Criticality assessments in CACE. Currently, empirical methods (e.g., ExeGPT [47]) estimate expected output lengths based on historical model performance data, yet these estimates vary significantly with model architectures and specific coding scenarios. Future work could explore hybrid approaches combining empirical measurements with explicit developer-provided estimates, enhancing prediction accuracy and enabling more proactive eviction decisions. Integrating more precise token-length forecasts or direct user input promises to further improve latency outcomes and optimize resource utilization.

D. Modeling Developer Behavior and Usage Patterns

Finally, eviction strategies could be further enhanced by explicitly incorporating developer behavior and usage patterns into eviction decisions. Capturing historical developer interaction data, such as typical model usage sequences, working-hour patterns, and task correlations, could enable more accurate forecasts of model demand. By predicting recurring developer interactions, future systems could proactively prefetch necessary models or evict models less likely to be used during certain periods. This direction could substantially reduce cold-start latency, improving the developer experience by better aligning model availability with actual developer workflows.

VII. RELATED WORK

Efficient serving of LLMS, including specialized models for coding tasks (CodeLLMs), has become a central research focus due to their significant computational and memory demands. Prior work can be categorized into three major strategies: full model eviction, KV-cache offloading and paging, and CPU–Accelerator orchestration for Mixture-of-Experts models.

A. Full Model Multiplexing on Shared Accelerators

Model-multiplexing frameworks load, evict, and route multiple models on a shared accelerator pool to maximize accelerator utilization. Ray Serve [30] supports fractional-accelerator allocation and concurrent model hosting; its eviction policy is a simple Least-Recently-Used (LRU) heuristic designed for generic ML workloads. Ollama [31] offers an easy-to-deploy, local multiplexing solution, again relying on recency-based eviction and thus incurring cold-start delays as concurrency grows. ModelMesh [32], the engine behind KServe's "serverless" back-end, targets serving with lazy, on-demand model loading and an LRU-style cache to fit thousands of models into limited accelerator pools. FaaSwap [33] improves accelerator sharing in serverless inference by keeping model checkpoints in CPU RAM and evicting them into accelerators on demand; although it reduces eviction latency relative to disk, the decision is still recency-driven and agnostic to task latency requirements or model reload cost. ServerlessLLM [48] extends this idea with optimized checkpoint storage and faster staging, yet likewise triggers checkpoint loading without considering the heterogeneity of CodeLLM tasks.

Despite engineering advances, these systems share a core limitation: their eviction logic ignores model load cost and task-specific latency. Evicting a large reasoning model can cost hundreds of milliseconds more than a small completion model, yet pure-LRU or throughput heuristics treat them identically, increasing TTFT or E2E latency under mixed IDE workloads, as shown in the experiments.

B. Single Model Optimization with KV-Cache Offloading and Memory Paging

A parallel line of work tackles accelerator memory pressure within a single model instance by virtualizing the key–value (KV) cache or paging model weights between device and host memory. PagedAttention in vLLM [49] slices the KV cache into fixed-size pages and dynamically evict them between accelerator and CPU RAM, achieving high throughput for long-context inference without increasing model-level eviction frequency. Pie introduces performance-transparent offloading that migrates KV blocks on demand while overlapping transfer and computation to hide latency [50]. FlexGen [51] further explores weight and KV offloading across accelerator, CPU,

and even non-volatile memory express (NVMe) tiers to enable very large models on commodity hardware.

These techniques operate below the granularity of whole-model eviction: they optimize memory footprints for a single loaded model and assume the model itself remains resident on the accelerator. Consequently, they do not decide which model to unload when accelerator memory fills, nor do they account for task-specific latency. CACE is complementary: it manages model-level residency across many CodeLLMs, integrating Reloading Cost, Future Demand, and Task Criticality. In practice, a serving stack could combine PagedAttention (for intra-model paging) with CACE (for inter-model eviction), achieving both fine-grained memory efficiency and context-aware model selection.

C. CPU–Accelerator Orchestration for Mixture-of-Experts Models

Mixture-of-Experts (MoE) architectures partition a large LLM into dozens to thousands of "expert" sub-networks, of which only a few are activated per token [52]. This sparse activation reduces floating-point operations per second (FLOPs) but introduces new orchestration challenges: which experts should stay in accelerator memory, and when to migrate cold experts from CPU or NVMe storage.

Fiddler [53] targets inference for mega-scale MoEs whose total parameters exceed accelerator capacity. Fiddler keeps frequently selected experts resident on accelerators and pre-fetches long-tail experts from CPU RAM based on gating probabilities. DeepSpeed-MoE Inference [54] similarly employs dynamic expert evictions and kernel-fusion optimizations to serve billion-parameter MoEs on heterogeneous clusters while maintaining sub-second latency.

MoE orchestration addresses intra-model weight placement: experts are fragments of the *same* model and share an input pipeline, whereas CACE manages inter-model eviction across many distinct CodeLLMs. While MoE systems optimize which experts reside on the accelerator, they assume a fixed set of models already loaded; they do not decide which entire model family (Python-completion vs. Java-reasoning, for instance) deserves residency when accelerator memory is scarce. Hence CACE is complementary: it determines which full CodeLLMs remain hot, whereas Fiddler-style schedulers can further optimize expert placement within those resident models.

VIII. Threats to Validity

In this section, we discuss the threats to validity.

A. Internal Validity

Our study assumes a single active instance of each CodeLLM in accelerator memory. In practical settings, enterprises often deploy multiple replicas to handle concurrent requests, potentially affecting eviction dynamics. Additionally, our evaluation uses a controlled, single-node setup, simplifying factors like distributed scheduling and real-time metadata accuracy. Any hidden interaction or simplification could influence the observed performance benefits, thus, future

evaluations should explore replica-aware eviction policies and multi-node environments.

B. External Validity

We evaluated two primary task categories: TTFT-sensitive code completion and E2E-sensitive code reasoning. Real-world workloads may include additional tasks such as documentation generation or code refactoring, presenting distinct latency characteristics. Furthermore, our experiments employed a homogeneous hardware setup with identical accelerators and a curated set of CodeLLMs, while actual deployments typically involve diverse hardware and broader model varieties. Thus, further validation across heterogeneous hardware and workloads would strengthen the validity of our findings.

IX. Conclusion

AI-assisted coding with CodeLLMs has become an essential part of modern software development workflows. To meet privacy, security, and latency requirements, many enterprises are adopting self-hosted solutions for deploying and serving CodeLLMs. However, the growing number and diversity of available CodeLLMs, coupled with the limited accelerator memory, pose significant challenges for efficiently managing model resources, particularly under latency-critical workloads like code completion and code reasoning.

In this paper, we presented a novel model eviction approach that goes beyond traditional LRU policies by incorporating multiple sources of information, including request queue insights, sliding window tracking of recent model usage, future usage demand, model load times, and task criticality based on the expected output length. By integrating these factors, our proposed policy makes more informed eviction decisions that reduce unnecessary data movements and improve response latency for key developer tasks.

We evaluated our approach using realistic developer workloads and showed that it can reduce both model cache misses and overall latency compared to baseline strategies. While our work focuses on single-instance deployments and isolates the eviction logic from full scheduling pipelines, it lays the groundwork for more advanced memory management strategies in production-grade CodeLLM serving systems.

In future work, we plan to extend our approach to multi-instance and distributed settings, heterogeneous accelerators, and utilize CPU memory as a secondary cache layer. We also aim to explore the dynamic adaptation of eviction thresholds based on workload patterns and resource availability.

X. 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-4.0 was used for copy-editing. All experiments, analysis, writing, and results were performed by the authors, who also thoroughly reviewed the final content. This complies with IEEE and ACM policies on AI use in publications.

REFERENCES

[1] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," *arXiv preprint arXiv:2406.00515*, 2024.

[2] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proceedings of the 6th ACM SIGPLAN international symposium on machine programming*, 2022, pp. 1–10.

[3] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, "Measuring github copilot's impact on productivity," *Communications of the ACM*, vol. 67, no. 3, pp. 54–63, 2024.

[4] Cursor, "Cursor: The ai code editor," 2024, accessed: 2025-07-31. [Online]. Available: https://www.cursor.com/

[5] G. Copilot, "Github copilot: The ai editor for everyone," 2024, accessed: 2025-07-31. [Online]. Available: https://github.com/features/copilot

[6] V. Murali, C. Maddila, I. Ahmad, M. Bolin, D. Cheng, N. Ghorbani, R. Fernandez, N. Nagappan, and P. C. Rigby, "Ai-assisted code authoring at scale: Fine-tuning, deploying, and mixed methods evaluation," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1066–1085, 2024.

[7] A. Q. Developer, "Amazon q developer: The most capable generative ai–powered assistant for software development," 2024, accessed: 2025-07-31. [Online]. Available: https://aws.amazon.com/q/developer/

[8] Jetbrains, "Intellij ai editor," 2024, accessed: 2025-07-31. [Online]. Available: https://www.jetbrains.com/help/idea/use-ai-in-editor.html

[9] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, L. Shen, Z. Wang, A. Wang, Y. Li *et al.*, "Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023, pp. 5673–5684.

[10] X. Gu, M. Chen, Y. Lin, H. Zhang, C. Wan, Z. Wei, Y. Xu, and J. Wang, "On the effectiveness of large language models in domain-specific code generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 3, pp. 1–22, 2025, demonstrates task/domain variability among CodeLLMs across 12 software domains.

[11] R. Xu, J. Cao, Y. Lu, H. Lin, X. Han, B. He, S.-C. Cheung, and L. Sun, "Cruxeval-x: A benchmark for multilingual code reasoning, understanding and execution," *arXiv preprint arXiv:2408.13001*, 2024.

[12] L. Chai, S. Liu, J. Yang, Y. Yin, K. Jin, J. Liu, T. Sun, G. Zhang, C. Ren, H. Guo *et al.*, "Mceval: Massively multilingual code evaluation," *arXiv preprint arXiv:2406.07436*, 2024.

[13] N. Pavlichenko, I. Nazarov, I. Dolgov, E. Garanina, K. Lasocki, J. Reshetnikova, S. Boitsov, I. Bondyrev, D. Karaeva, M. Sheptyakov, D. Ustalov, A. Mukhin, S. Proshev, N. Abramov, O. Kolomyttseva, K. Lysaniuk, I. Zavidnyi, A. Semenkin, V. Tankov, and U. Sazanovich, "Mellum-4b-base," 2025. [Online]. Available: https://huggingface.co/JetBrains/Mellum-4b-base

[14] JetBrains, "Full line code completion in jetbrains ides: All you need to know," 2025, accessed: 2025-07-31. [Online]. Available: https://blog.jetbrains.com/blog/2024/04/04/full-line-code-completion-in-jetbrains-ides-all-you-need-to-know/#under-the-hood

[15] X. Gu, M. Chen, Y. Lin, Y. Hu, H. Zhang, C. Wan, Z. Wei, Y. Xu, and J. Wang, "On the effectiveness of large language models in domain-specific code generation," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 3, pp. 1–22, 2025.

[16] Q. Zhu, J. Cao, Y. Lu, H. Lin, X. Han, L. Sun, and S.-C. Cheung, "Domaineval: An auto-constructed benchmark for multi-domain code generation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 39, no. 24, 2025, pp. 26 148–26 156.

[17] J. Li, G. Li, X. Zhang, Y. Zhao, Y. Dong, Z. Jin, B. Li, F. Huang, and Y. Li, "Evocodebench: An evolving code generation benchmark with domain-specific evaluations," *Advances in Neural Information Processing Systems*, vol. 37, pp. 57 619–57 641, 2024.

[18] D. Zheng, Y. Wang, E. Shi, H. Zhang, and Z. Zheng, "How well do llms generate code for different application domains? benchmark and evaluation," *arXiv preprint arXiv:2412.18573*, 2024.

[19] A. Semenkin, V. Bibaev, Y. Sokolov, K. Krylov, A. Kalina, A. Khannanova, D. Savenkov, D. Rovdo, I. Davidenko, K. Karnaukhov *et al.*, "Full line code completion: Bringing ai to desktop," *arXiv preprint arXiv:2405.08704*, 2024.

[20] O. Dunay, D. Cheng, A. Tait, P. Thakkar, P. C. Rigby, A. Chiu, I. Ahmad, A. Ganesan, C. Maddila, V. Murali *et al.*, "Multi-line ai-assisted code authoring," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, pp. 150–160.

[21] P. Mayer and A. Bauer, "An empirical analysis of the utilization of multiple programming languages in open source projects," in *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, 2015, pp. 1–10.

[22] Z. Li, X. Qi, Q. Yu, P. Liang, R. Mo, and C. Yang, "Multi-programming-language commits in oss: an empirical study on apache projects," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 219–229.

[23] P. S. Kochhar, D. Wijedasa, and D. Lo, "A large scale study of multiple programming languages and code quality," in *2016 IEEE 23Rd international conference on software analysis, evolution, and reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 563–573.

[24] H. Yang, Y. Nong, S. Wang, and H. Cai, "Multi-language software development: Issues, challenges, and solutions," *IEEE Transactions on Software Engineering*, vol. 50, no. 3, pp. 512–533, 2024.

[25] F. Cassano, J. Gouwar, D. Nguyen, S. Nguyen, L. Phipps-Costin, D. Pinckney, M.-H. Yee, Y. Zi, C. J. Anderson, M. Q. Feldman *et al.*, "Multipl-e: A scalable and extensible approach to benchmarking neural code generation," *arXiv preprint arXiv:2208.08227*, 2022.

[26] T. Dunn, "Llms are helpful with python, but what about all of the other programming languages?" 2023, accessed: 2025-07-31. [Online]. Available: https://blog.continue.dev/programming-languages/

[27] E. Parliament, "Eu ai act: first regulation on artificial intelligence," 2023, accessed: 2025-07-31. [Online]. Available: https://www.europarl.europa.eu/topics/en/article/20230601STO93804/eu-ai-act-first-regulation-on-artificial-intelligence

[28] X. Wu, R. Duan, and J. Ni, "Unveiling security, privacy, and ethical concerns of chatgpt," *Journal of Information and Intelligence*, vol. 2, no. 2, pp. 102–115, 2024.

[29] B. C. Das, M. H. Amini, and Y. Wu, "Security and privacy challenges of large language models: A survey," *ACM Computing Surveys*, vol. 57, no. 6, pp. 1–39, 2025.

[30] Ray, "Ray serve model multiplexing," 2025, accessed: 2025-07-31. [Online]. Available: https://docs.ray.io/en/latest/serve/model-multiplexing.html

[31] Ollama, "Handle concurrent requests in ollama," 2025, accessed: 2025-07-31. [Online]. Available: https://github.com/ollama/ollama/blob/main/docs/faq.md

[32] IBM, "Get started with kserve modelmesh for multi-model serving," 2023, accessed: 2025-07-31. [Online]. Available: https://developer.ibm.com/tutorials/awb-get-started-with-kserve-modelmesh-for-multi-model-serving/

[33] M. Yu, A. Wang, D. Chen, H. Yu, X. Luo, Z. Li, W. Wang, R. Chen, D. Nie, and H. Yang, "Faaswap: Slo-aware, gpu-efficient serverless inference via model swapping," *arXiv preprint arXiv:2306.03622*, 2024. [Online]. Available: https://arxiv.org/abs/2306.03622

[34] CACE, "Cace replication package," 2025, accessed: 2025-07-31. [Online]. Available: https://figshare.com/s/aa85c6796722d65b9e12

[35] B. Steenhoek, K. Sivaraman, R. S. Gonzalez, Y. Mohylevskyy, R. Z. Moghaddam, and W. Le, "Closing the gap: A user study on the real-world usefulness of ai-powered vulnerability detection & repair in the ide," *arXiv preprint arXiv:2412.14306*, 2024.

[36] L. Fridman and C. Team, "Future of programming with ai — lex fridman podcast 447," 2024, accessed: 2025-07-31. [Online]. Available: https://lexfridman.com/cursor-team-transcript

[37] H. Li, D. Guo, W. Fan, M. Xu, J. Huang, F. Meng, and Y. Song, "Multi-step jailbreaking privacy attacks on chatgpt," *arXiv preprint arXiv:2304.05197*, 2023.

[38] V. Rathod, S. Nabavirazavi, S. Zad, and S. S. Iyengar, "Privacy and security challenges in large language models," in *2025 IEEE 15th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE, 2025, pp. 00 746–00 752.

[39] E. D. P. Board, "Ai privacy risks & mitigations large language models (llms)," 2023, accessed: 2025-07-31. [Online]. Available: https://www.edpb.europa.eu/our-work-tools/our-documents/support-pool-experts-projects/ai-privacy-risks-mitigations-large_en

[40] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu *et al.*, "Qwen2. 5-coder technical report," *arXiv preprint arXiv:2409.12186*, 2024.

[41] M. AI, "Codestral: Empowering developers and democratising coding with mistral ai," May 2024, accessed: 2025-07-31. [Online]. Available: https://mistral.ai/news/codestral

[42] RedMonk, "The redmonk programming language rankings: June 2024," 2024, accessed: 2025-07-31. [Online]. Available: https://redmonk.com/sogrady/2024/09/12/language-rankings-6-24/

[43] TIOBE, "Tiobe index for may 2025," 2025, accessed: 2025-07-31. [Online]. Available: https://www.tiobe.com/tiobe-index/

[44] CodeSignal, "Developers & ai coding assistant trends," 2023, accessed: 2025-07-31. [Online]. Available: https://codesignal.com/report-developers-and-ai-coding-assistant-trends/

[45] JetBrains, "The state of developer ecosystem 2023," 2023, accessed: 2025-07-31. [Online]. Available: https://www.jetbrains.com/lp/devecosystem-2023/

[46] StackOverflow, "Developer sentiment around ai/ml," 2023, accessed: 2025-07-31. [Online]. Available: https://stackoverflow.co/labs/developer-sentiment-ai-ml/

[47] H. Oh, K. Kim, J. Kim, S. Kim, J. Lee, D.-s. Chang, and J. Seo, "Exegpt: Constraint-aware resource scheduling for llm inference," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 369–384.

[48] Y. Fu, L. Xue, Y. Huang, A.-O. Brabete, D. Ustiugov, Y. Patel, and L. Mai, "{ServerlessLLM}:{Low-Latency} serverless inference for large language models," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 135–153.

[49] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. 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*, 2023, pp. 611–626.

[50] Y. Xu, Z. Mao, X. Mo, S. Liu, and I. Stoica, "Pie: Pooling cpu memory for llm inference," *arXiv preprint arXiv:2411.09317*, 2024.

[51] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Ré, I. Stoica, and C. Zhang, "Flexgen: High-throughput generative inference of large language models with a single gpu," in *International Conference on Machine Learning*. PMLR, 2023, pp. 31 094–31 116.

[52] Z. Chen, Y. Deng, Y. Wu, Q. Gu, and Y. Li, "Towards understanding the mixture-of-experts layer in deep learning," *Advances in neural information processing systems*, vol. 35, pp. 23 049–23 062, 2022.

[53] K. Kamahori, T. Tang, Y. Gu, K. Zhu, and B. Kasikci, "Fiddler: Cpu-gpu orchestration for fast inference of mixture-of-experts models," *arXiv preprint arXiv:2402.07033*, 2024.

[54] S. Rajbhandari, C. Li, Z. Yao, M. Zhang, R. Y. Aminabadi, A. A. Awan, J. Rasley, and Y. He, "Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale," in *International conference on machine learning*. PMLR, 2022, pp. 18 332–18 346.