THE UNIVERSITY OF CHICAGO

RETHINKING AI SYSTEMS THROUGH EFFICIENT MODEL COMMUNICATION

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
YUHAN LIU

CHICAGO, ILLINOIS
AUGUST 2026

ORCID iD: https://orcid.org/xxxx-xxxx-xxxx-xxxx

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest gratitude to my Ph.D. advisors, Junchen Jiang and Shan Lu. This dissertation would not have been possible without their guidance, encouragement, and support. They have shaped not only how I conduct research, but also how I think, communicate, collaborate, and grow as a researcher.

Junchen taught me to always set a higher bar for myself and my work: to compare my work with the strongest and most challenging baselines, carefully reason through each research decision, and use solid experiments to test whether a solution is feasible. He also taught me that good research requires not only technical skills, but also real passion for the problem and confidence in one's ideas. This passion and confidence are especially important when research becomes difficult, the results are not as expected, or the direction is still unclear. Junchen's careful and sharp thinking has influenced me throughout my Ph.D. Shan taught me how to find research problems that are both important and interesting. She helped me understand that research should start from concrete examples and real-world needs. We should first find a problem worth solving and then look for a suitable solution, instead of starting with a solution and searching for a problem where it can be used. She often emphasized that every design decision should have a clear intuition. Shan also taught me to focus on what matters the most. These lessons not only helped me complete my doctoral research, but also shaped my understanding of what makes research good. Their high standards will continue to guide me in my future academic career.

I would also like to sincerely thank my dissertation committee members, Esha Choukse and Ganesh Ananthanarayanan. Esha was my mentor during my internship at Microsoft Research, and I learned a great deal from working with her. Her broad knowledge, clear thinking, and thoughtful guidance helped me grow as a researcher and introduced me to new ways of approaching interdisciplinary research. My collaboration with Ganesh began with the CacheGen project. Since then, his questions, comments, and broader perspective

have continually helped me improve my work and think more deeply about research. He has always been generous with his time and advice, and I am grateful for the many insights he brought to our discussions. I am deeply grateful to Esha and Ganesh for serving on my committee, for writing letters on my behalf, and for their valuable feedback, support, and encouragement throughout this process.

I am grateful to the faculty members at the University of Chicago with whom I had the opportunity to collaborate, including Hank Hoffmann, Michael Maire, and Ari Holtzman. Working with them introduced me to new research areas and perspectives and helped me better understand the value of interdisciplinary collaboration. I am grateful for their insights, encouragement, and contributions to my research. I am also grateful to Madan Musuvathi, who co-mentored me with Esha during my internship at Microsoft Research. He spent countless hours discussing research with me, challenging my thinking, and helping me improve my ideas. These detailed and thoughtful conversations played an important role in my growth as a researcher.

I am also thankful to my labmates and all of my collaborators. First, I would like to thank Chengcheng Wan. She guided me through my first research project as a Ph.D. student and gave me the opportunity to contribute to one of her papers. As someone who was just starting my Ph.D., these experiences helped me learn the full research process and built an important foundation for my later work. I want to give special thanks to Kuntai Du, who contributed to every research project I worked on during my Ph.D. At each stage of these projects, he asked many sharp and important questions and provided detailed and helpful comments on our writing. I am grateful to Jiayi Yao for discussing and improving research ideas with me, and for helping me iterate and implement many ideas that were not yet fully developed. I am thankful to Yihua Cheng, Siddhant Ray, Qizheng Zhang, Xu Zhang, and Zhengxu Xia. Discussing problems, conducting research, and writing papers with all of you were valuable parts of my Ph.D. experience. I am also grateful to the undergraduate

and master's students with whom I worked: Shaoting Feng, Hanchen Li, and Zhuohan Gu. Working with you not only moved our projects forward, but also taught me how to share ideas more clearly, coordinate with collaborators, and work through the uncertainty of research together. I would like to give my gratitude to the entire LMCache team, including Yihua Cheng, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Rui Zhang, and Kuntai Du. Working together to turn a research idea into an open-source system that people could actually use was both challenging and rewarding. During this process, I learned a great deal from everyone about engineering, open-source collaboration, and community building, all of which go far beyond research itself. More importantly, I truly value the experience of solving problems together, supporting one another, and seeing the project grow. I feel very fortunate to have worked with such a talented and passionate group of people.

I would also like to thank my undergraduate advisor, Shivaram Venkataraman. He was the person who first introduced me to research and guided me into the field of machine learning systems. At the beginning of my research journey, he taught me many basic but important research skills: how to read papers, ask research questions, design experiments, and explain ideas clearly. More importantly, he has continued to care about and support my growth over the years. From applying to Ph.D. programs to applying for faculty positions, whenever I faced an important decision or felt unsure about what to do next, he patiently discussed it with me and gave me thoughtful and helpful advice. His guidance has gone far beyond any single research project and has had a lasting influence on both my academic path and my life decisions.

Finally, I would like to thank my family. I am deeply grateful to my parents for their unconditional love, trust, and support. No matter what choices I made or what difficulties I faced, they always stood behind me and allowed me to pursue my goals without worrying about anything else. Without their continued support and sacrifices, I could not have com-

pleted my Ph.D. I also want to thank my beloved husband, Yuyang. During some of the most difficult, stressful, and uncertain moments of my Ph.D., he was always there to comfort and encourage me and help me through these hard times. He not only shared my daily life, but also patiently listened to my early research ideas, discussed and improved them with me, and helped me look at problems from new perspectives. I am deeply grateful for his understanding, patience, and support throughout this journey, which made this long and challenging process much less lonely.

I am also grateful to all the friends who cared about me, encouraged me, and supported me along the way. A Ph.D. journey is made up not only of papers, experiments, and deadlines, but also of many honest conversations, mutual support, and moments shared together. Because of all of you, I was able to continue through the difficult times and reach this point. I would like to use this dissertation to express my sincere gratitude to everyone who has helped and supported me throughout this journey.

ABSTRACT

Artificial intelligence (AI) has become increasingly ubiquitous and powerful in recent years, and the interaction patterns within AI inference systems have evolved with it. These patterns have shifted from single-turn model-to-user exchanges to more complex interactions, such as multi-turn model-to-user, model-to-model, and model-to-software workflows.

This shift greatly changes how inference systems should be designed and implemented. In the classic machine learning era, the interface with a model was often a human-readable object, such as text, images, audio, or video, because a human was commonly the final consumer. In modern AI systems, however, interactions with models often do not involve humans at all: model outputs are consumed by other inference requests, models, or software control logic. This opens the possibility of interacting with models through model-native state instead of only human-readable media, such as passing an LLM's key-value (KV) cache between inference components or converting software context into a differentiable loss function that a model can optimize. When forced to communicate through human-readable data, AI systems may repeat computation, increasing response delay, or miss what downstream applications need, reducing decision quality.

This dissertation argues that, in these settings, the right media for interacting with AI models are model-native states: internal or directly model-digestible states that incorporate a model's understanding of the input more directly than human-readable data. Once AI systems communicate through such states, the central systems problem becomes how to store, move, transform, and specialize them without losing the information that makes them useful to the model. I study this problem through two forms of model-native state: the key-value (KV) cache in large language model (LLM) inference, which represents input contexts in a form the model can directly consume, and model behavior specialized to how software uses the model's outputs.

In this thesis, I first focus on KV cache. Traditionally, KV cache was used only within a

single inference request and kept entirely in GPU RAM. Today, however, KV cache increasingly exceeds available GPU memory, creating the need to move it out of the GPU and store it in CPU DRAM or on disk. This shift creates a loading challenge: before a model can use persisted KV cache, the system must fetch it back into GPU RAM, and that transfer can incur substantial latency. I present CacheGen, a system that makes KV cache practical to fetch and load from secondary storage devices by encoding it into compact bitstreams and streaming it. A second challenge is interoperability: because KV cache is a model-specific internal encoding of the input, it cannot be directly reused by another model. I then present DroidSpeak, a system that enables KV cache reuse across different models.

Finally, I show that model-native state is useful beyond LLM KV cache. ChameleonAPI considers ML APIs that expose generic human-readable labels and scores even though applications care about whether those outputs lead to correct software decisions. ChameleonAPI extracts how an application uses model outputs in control flow and converts that software context into a model-native training objective, allowing the model to internalize which errors matter for the application. Together, these systems demonstrate that the next generation of AI infrastructure should manage not only models, prompts, and outputs, but also the model-native states through which AI systems communicate and reuse computation.

CHAPTER 1

INTRODUCTION: MODEL-NATIVE STATE AS AN AI SYSTEMS INTERFACE

1.1 The Rise of Inference-Dominated AI

Artificial intelligence (AI) has become much more common in people's daily lives. People now use AI systems to answer questions, write and debug code, summarize documents, draft messages, generate media, and support many kinds of work. As AI becomes more widely used, the systems problem around AI is also changing.

Much of the future workload of AI systems is expected to be dominated by inference: running trained models to answer user and application requests. Training remains expensive, but it happens far less often than the many inference calls served after a model is deployed. For large models, each inference request can require substantial computation, memory, and data movement, especially when the request includes long context or conversation history. As a result, inference is becoming a major source of latency and cost. Production-scale studies and benchmarks already identify inference as a major AI systems workload, and recent analyses of LLM deployment and data-center energy demand argue that repeated inference at user scale will be a central driver of future AI infrastructure cost [Jouppi et al., 2017, Reddi et al., 2020, Desislavov et al., 2021, Samsi et al., 2023, Luccioni et al., 2023, Fu et al., 2024, International Energy Agency, 2025].

The key question is therefore: *why can inference be so resource-intensive?*

1.2 Why Modern AI Inference Repeats Work

Modern LLM inference has a prefill phase before the model can generate new tokens. During prefill, the model processes the input prompt and constructs internal key-value (KV) cache

states that will later be used during the token generation phase, which is also called decoding. This prefill phase becomes expensive for long inputs because standard Transformer attention compares tokens against other tokens in the context; as a result, the amount of attention work in prefill grows super-linearly with context length [Vaswani et al., 2023, Dao et al., 2022, Pope et al., 2023b]. A request with a short user query but a long document, chat history, codebase, financial report, or collection of tool outputs can therefore spend much of its latency before the first output token appears.

The problem becomes worse when many requests reuse the same long context. Consider a chatbot that answers questions about a long document. The system typically prepends the document to the user's question so the model can answer with the full context. The first question therefore requires the model to process the document during prefill. But the second question, the third question, and every later question may also prepend the same document again. The user question may change by only a few tokens, but the inference engine may still repeat most of the same long-context prefill work. Thus repeated work and super-linear prefill latency reinforce each other: the system repeatedly pays an expensive cost on almost the same input.

The same pattern appears in multi-model workflows. An agentic coding system may use one model to understand a repository, another model to write code, a third model to generate tests, and a fourth model to produce documentation. All of these models may need access to the same large codebase, or the same set of relevant files, and the same evolving conversation state. In today's systems, each model receives the context as code, then each model must reconstruct its own internal representation (KV cache) from scratch. In such systems, each simple user query may trigger multiple models to repeat the same prefill work on the shared long contexts.

Together, these examples indicate several new interaction patterns in AI inference systems. The chatbot example is a multi-turn model-to-user interaction: the user asks a se-

quence of questions, and each turn depends on the same long context. The agentic coding example is a multi-model interaction: several models need access to the same codebase and evolving conversation state while working toward one task.

1.3 New Interactions in AI Systems

The repeated-prefill examples above are not isolated implementation problems. They reflect a broader change in how AI systems interact. Classic machine learning systems largely fit a simple interaction pattern: a model receives an input and returns a prediction to a human or to a software application in a single turn. A spam classifier returns "spam" or "not spam." An image classifier returns labels. A sentiment model returns a score. The model may be embedded inside a larger application, but the model's role is often a single-turn and stateless interaction with a relatively small output.

Modern AI inference systems expand this pattern in at least three ways. The first two are the ones illustrated in the previous section. First, model-to-user interaction has become multi-turn. The model no longer answers one isolated query; it participates in an ongoing session whose history and external context become part of later inputs. Second, model-to-model interaction has become common. Compound AI systems use multiple models or agents that specialize in different tasks and exchange intermediate results while working toward a shared goal. Third, model-to-software interaction has become deeper. Software does not merely display model outputs to users; it branches on those outputs, invokes tools, schedules work, updates state, and composes model calls into larger workflows.

These interaction types differ from classic model serving because the consumer of a model's output is often another automated component, not only a human reader. In a multi-turn chat system, each new request may include the earlier conversation and the same retrieved document or other shared context. In a multi-agent system, the output of one model may be used primarily to shape the input or execution path of another model. In

an ML API application, output labels may matter only insofar as they drive downstream control flow. In these cases, a human may not always be in the loop when information moves from one component to the next. This raises a systems question: when models interact with other models or software components, what should be the right interface for interacting with models?

1.4 The Limits of Human-Readable Interaction Media

Human users have historically interacted with AI models directly, so human-readable data has been an appropriate interaction medium. Text prompts, documents, labels, scores, and API outputs are also easy for people to inspect, edit, log, and debug.

However, human-readable data is not always the best interaction medium inside AI inference systems, especially given the new interaction patterns described in the previous section. It can be inefficient because it does not preserve the internal computation that has already been performed. When an LLM reads a long context, it constructs the KV cache tensors. If the system throws away those tensors and later sends the same context as text, the model must repeat the prefill computation. The text is readable, but it is not the most direct representation for the downstream model.

This dissertation does not argue that human-readable interfaces should disappear. They remain essential for human oversight and stable external APIs. Instead, the argument is that when there is no human in the loop, what should be the right interface for interacting with AI models?

1.5 Model-Native State as the Interaction Medium

I use the term *model-native state* to refer to internal or directly model-digestible state that incorporates a model's understanding of an input more directly than human-readable data.

Model-native state is not defined by whether a human can interpret it. It is defined by whether it captures information in a form that downstream AI computation can use without reconstructing that information from scratch.

The most concrete example in this dissertation is the KV cache of an LLM. During prefill, each transformer layer produces key and value tensors for the input tokens. During decoding, the model uses these tensors to attend to the previous context while generating new tokens. The KV cache is therefore the model's internal representation of the context for the purpose of future generation. It is not a readable document, but it is a high-dimensional tensor that captures the same information in a form that the model can precisely what the model needs to avoid recomputing the document.

Model-native state can also describe how software context should shape model behavior. In ML API applications, the important question is often not whether the model returns the exact human-labeled output, but whether the model's output leads the application to the correct decision. A training objective that encodes this software decision process becomes a model-native representation of the software context: the model is optimized for the way its output will be consumed.

Using model-native state as the medium for interacting with AI models changes the design space of AI infrastructure. Instead of always communicating through text, labels, or serialized objects, a system can store a model's internal state, move it across storage tiers, compress it for transfer, repair it when it crosses model boundaries, or specialize it to the software that will consume the model output. The remainder of this dissertation explores these possibilities.

1.6 Systems Challenges for Model-Native State

Model-native state is powerful because it avoids unnecessary translation through human-readable data. However, it is also difficult to use because of several unsolved system-level

challenges, which this dissertation focuses on as described below.

The first challenge is size. KV cache is useful because it stores the model's processed representation of a context, but this representation is much larger than the original text. For common LLMs, the KV cache of a context can be 10,000–100,000× larger than the same context in text form, because each token is expanded into key and value tensors across many layers and channels. This size is difficult to manage, especially at large inference scale where many users, documents, conversations, and agents may create many such caches. Model-native state therefore needs size-aware representations and management mechanisms that make it practical to store, move, and load.

The second challenge is interoperability. Model-native state is tied to the model that created it. A KV cache produced by one model is not directly usable by another model, even if both models process the same input. Reuse therefore requires a validity condition: when is the state good enough for the downstream model, and what recomputation or repair is necessary when it is not?

The third challenge is specialization to software use. When a model's output is consumed by software control logic, generic human-readable labels and scores may not capture which errors matter to the application. The system needs instead to express the software context in a form the model can optimize, such as a differentiable training objective that reflects downstream decisions.

Together, these challenges show why model-native state should not be handled as raw, unmanaged tensors or generic outputs. The systems in this dissertation provide concrete examples of such support.

1.7 Thesis Statement and Contributions

This dissertation argues that, when model outputs are consumed primarily by other models or software components, the right media for interacting with AI models are model-native

states. By storing, moving, compressing, repairing, and specializing these states, AI systems can reduce redundant computation and communication while preserving or even improving downstream quality.

I support this thesis through a measurement study and three systems.

I first study one important example of model-native states in modern AI inference systems: the KV cache in LLM serving. Although KV cache is not a new idea, it was long used only within *a single inference query* and kept entirely in GPU RAM. However, as contexts become longer and models become larger, KV cache increasingly needs to be reused *across different user queries* to skip redundant computation and significantly speed up inference, as supported by the measurement study presented in Chapter 2. At the same time, the growing size of KV cache makes it impossible to keep the caches of millions of user queries inside GPU memory: they must instead be moved out of the GPU into CPU DRAM, local storage, or remote storage. This shift elevates KV cache from an ephemeral tensor to a state that needs to be carefully managed: a first-class abstraction that has been attracting great interest in both academia and industry. However, there remain two unsolved system challenges in using KV cache as the interaction medium.

The first challenge is the size of KV cache. KV cache can be 10,000–100,000× larger than the same context in text form, so once it is moved out of GPU memory, the delay of loading it back becomes non-trivial—sometimes even longer than recomputing it from text. To solve this challenge, **CacheGen** reduces the size of KV cache by encoding it into compact bitstreams using layer-aware quantization, token-local delta representations, and entropy coding, and then streams those bitstreams adaptively under changing bandwidth. CacheGen makes KV cache practical to fetch and load from secondary storage devices while preserving output quality.

The second challenge is that KV cache is a model-specific encoding of the input context, so it cannot be directly reused by another model. However, as LLM applications become

more and more complex, models increasingly talk to other models, exchanging the same piece of context. Sharing the KV cache of such shared contexts across LLMs, as a result, can significantly reduce both compute cycles and latency. To solve this challenge, **DroidSpeak** recomputes the layers that are sensitive to the KV cache difference between models, and reuses the remaining layers. This selective repair enables KV cache reuse across different models while avoiding much of the repeated prefill computation.

Finally, my thesis extends the model-native states argument beyond KV cache through the third system, **ChameleonAPI**. ChameleonAPI targets model-to-software interaction in applications built on ML APIs. The challenge there is that the models behind these APIs are trained to label images or audio, while the applications calling them often consume the outputs in other ways, such as taking the correct control-flow branch. This mismatch can greatly reduce the applications' decision accuracy in practice. To solve this challenge, ChameleonAPI extracts how an application uses model outputs in its control flow and converts that software context into an application-specific training objective. The resulting model internalizes which prediction errors matter to the application, reducing incorrect software decisions without changing the external API.

Together, these systems show that AI infrastructure should manage not only models, prompts, and outputs, but also the model-native states through which AI systems communicate and reuse computation.

1.8 Dissertation Roadmap

The rest of the dissertation is organized as follows.

Chapter 2 provides the background for the three systems in this dissertation. It reviews how the KV cache is used in transformer inference, explains why contexts in LLM inputs are both long and reused, presents deployment evidence that KV reuse is common in the real world, and describes why loading the stored cache back fast enough is the key obstacle. It

then introduces the non-LLM models served behind ML APIs, whose outputs applications use to make control-flow decisions, and distills these backgrounds into the three challenges addressed by the following chapters.

Chapter 3 presents CacheGen, a system for compressing and streaming KV cache. It focuses on the loading challenge: model-native state can avoid recomputation only if it can be fetched and loaded faster than it can be recomputed. CacheGen makes this possible by changing the storage and transfer representation of KV cache.

Chapter 4 presents DroidSpeak, a system for cross-model KV-cache reuse. It focuses on the interoperability challenge: state produced by one model is not automatically valid for another. DroidSpeak shows that partial recomputation can repair the reused state well enough to preserve quality while still reducing repeated computation.

Chapter 5 presents ChameleonAPI, which applies the same model-native-state perspective to software-model interaction. Rather than optimizing generic API outputs, ChameleonAPI encodes the software decision context into the model's training objective.

Chapter 6 surveys prior work related to all three systems in this dissertation, organized by theme: handling long contexts; KV cache management, reuse, and compression; faster LLM serving; fine-tuned models and multi-agent systems; and ML APIs and application-aware model serving.

Chapter 7 synthesizes the lessons from these systems and outlines a research agenda for model-native AI infrastructure, including data distribution for the Internet of Agents, analytics for model-native state, cross-architecture state sharing, and security and debugging for non-human-readable state.

CHAPTER 2

BACKGROUND

This chapter provides the background for the three systems in this dissertation. The first two sections cover LLM inference: why long, reused contexts make the KV cache worth storing and reusing (§2.1), and deployment evidence that such reuse is common in the real world—and that the reusable cache must live outside GPU memory (§2.2). Section 2.3 then turns to the non-LLM models served behind ML APIs, whose outputs applications use to make control-flow decisions. Finally, Section 2.4 distills these backgrounds into the three challenges that the following chapters address.

2.1 LLM Inference and Long, Re-usable Contexts

Transformer basics: Transformers Vaswani et al. [2023], Brown et al. [2020], Chowdhery et al. [2022] are the de facto models for most language generative services. At a high level, a transformer takes a sequence of input tokens[1] and generates a sequence of output tokens through two phases.

During the prefill phase, an attention neural network takes in the input tokens. Within each of the l layers in the attention module, the layer's input embedding (E) tensors are projected into a key (K) tensor and a value (V) tensor (shown in Figure 2.1), and the layer's output becomes the input E of the next layer. The K and V tensors contain information essential for the LLM to utilize the context later. All the KV tensors across different layers are together called the *KV cache*; similarly, all the E tensors are together called the *E cache*, a term that will matter when Chapter 4 examines KV reuse across models.

During the generation phase, also called the decoding phase, the KV cache is used to compute the attention score between every pair of tokens, which constitute the attention

1. A "token" can be a punctuation, a word, or a part of a word. Tokenizing an input is much faster than the generation process.

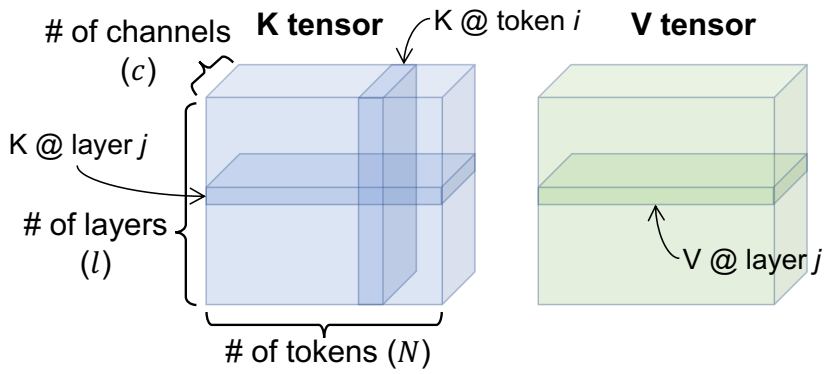


Figure 2.1: *An illustration of Key & Value tensors (KV cache).*

matrix, and generate output tokens in an autoregressive manner. For a long time, the KV cache, which has a large memory footprint Kwon et al. [2023b], is usually kept in GPU memory during this phase and released after a single user query finishes. Some emergent optimizations save and reuse the KV cache across different LLM requests, as we explain in the rest of this chapter.

Common system metrics for LLM inference performance: Two system metrics are used throughout this dissertation when measuring LLM inference performance. The first is *Time-to-First-Token* (*TTFT*), the duration from a query's arrival to the generation of its first output token. Since the prefill phase must be completed before the first output token can be generated, TTFT is dominated by the prefill delay. The second is the inference *throughput*, the number of queries (or tokens) a serving system processes per second; a variant, *goodput*, counts only the queries served within a latency SLO Zhong et al. [2024b]. Because decoding cannot start until prefill completes, a lengthy prefill phase hurts both metrics at once. How much this cost matters depends on how long the inputs actually are—which brings us to how LLMs are used in practice.

Context in LLM input: LLMs may generate low-quality or hallucinated answers when the response requires knowledge not already embedded in the models. Thus, many LLM applications and users supplement the LLM input with additional texts, referred to as the

context Gao et al. [2024e], Lewis et al. [2021]. The LLM can read the context first and use its in-context learning capability to generate high-quality responses[2].

The contexts in LLM input can be used for various purposes. *(i)* a user question can be supplemented with a document about specific domain knowledge, to produce better answers cha [2024], Rubin and Berant [2023a], any [2023], including using latest news to answer fact-checking inquiries per [2024], using case law or regulation documents to offer legal assistance Saleem [2023], Sydorenko [2023], etc.; *(ii)* code analysis applications retrieve context from a code repository to answer questions or generate a summary about the repository Bairi et al. [2023], Jain et al. [2023a], Jimenez et al. [2023], and similarly financial companies use LLMs to generate summaries or answer questions based on detailed financial documents Name [Year of Publicationb]; *(iii)* gaming applications use the description of a particular character as context so that the LLM can generate character dialogues or actions matching the character personality Park et al. [2023b], Wu et al. [2023c], Shi et al. [2023b]; *(iv)* in few-shot learning, a set of question-answer pairs are used as context to teach the LLM to answer certain types of questions Ahmed and Devanbu [2023a], Mani et al. [2023], Song et al. [2023b]; *(v)* in chatting apps, the conversational history with a user is often prepended as the context to subsequent user input to produce consistent and informed responses jwatte [2023], AuthorName [Year].

In practice, contexts are often **long** and often **reused** to supplement different user inputs.

Contexts are long: The contexts discussed above, such as case law documents, financial documents, news articles, code files, and chat history accumulated in a session, easily contain thousands of tokens or more. Intuitively, longer contexts are more likely to include the right information and hence may improve the quality of the response. Indeed, FiD Izacard and Grave [2021] shows that the accuracy increases from 40% to 48% when the context

2. An example of this process is retrieval-augmented generation (RAG), which uses a separate logic to select the context documents for a given query. It is well-studied in the natural-language literature and widely used in industry.

increases from 1K tokens to 10K. Retro Borgeaud et al. [2022] similarly shows that the generation quality (perplexity) improves significantly when the context increases from 6K tokens to 24K. The rise of LLM agents pushes context lengths further by another order of magnitude, because an agent accumulates instructions, tool outputs, file contents, and intermediate results in its context as it works: a recent study of over 100 trillion production tokens shows that the average prompt length has roughly quadrupled within two years, with programming prompts exceeding 20K tokens on average Aubakirova et al. [2025], and a trace analysis of Claude Code, a popular coding agent, shows a *single* task issuing 92 LLM calls and consuming about 2 million input tokens in total LMCache Team [2025]. This demand has fueled the ongoing race to train LLMs that accept ever longer inputs, from 2K tokens in the early ChatGPT, to 100K in Claude lon, and to a million tokens or more in recent frontier models Google DeepMind [2025].

Contexts are reused: These long contexts are often *reused* by different inputs. In the financial analysis example, consider two queries, "write a short summary based on the company's earning report last quarter" and "what were the company's top sources of revenue in the last quarter"; the same earning report is likely to be supplemented to both queries as the context. Similarly, the same case law document or latest news article can be used to answer many different queries in legal-assistant or fact-checking apps. As another example, during a chat session, early chat content keeps getting reused as part of the context for every later chat input. The reuse is even more pronounced in agentic workloads: in the Claude Code trace above, 92% of all input tokens are prefixes reused from earlier LLM calls in the same task LMCache Team [2025].

Contexts are increasingly reused across different LLMs: The reuse discussed so far happens across queries to the *same* LLM, but an emerging trend extends it further: the same context is often processed by *different* LLMs. In agentic workflows, multiple LLM agents—often distinct models or fine-tuned variants, each specialized for one task—collaborate to

solve complex, multi-step problems Li et al. [2024b], Hu et al. [2025b], Xi et al. [2025], and they typically share a common context, such as the conversation history of other agents or the same set of background documents, to stay coherent and consistent Hong et al. [2024b], Wu et al. [2024f], Guo et al. [2024]. In a coding workflow, for instance, the testing agent must read both the user's instructions and the code produced by the coding agent to write appropriate unit tests Hong et al. [2024b], Qian et al. [2024], Wu et al. [2024f]; in a trace of a LangGraph-based multi-agent application, where four agents collaborate to generate research ideas, 49% of the tokens in each conversation are prefixes reused from other LLM calls AI [2025], Du et al. [2025b]. Similar cross-LLM reuse arises when personalized assistants, each fine-tuned for a different user, answer queries over shared conversation histories or knowledge bases Li et al. [2024d], and when a deployed model is periodically replaced by a newer fine-tuned version (*e.g.,* ChatGPT APIs release new versions based on the same foundational model about every two months Microsoft Docs [2025]) or served alongside many LoRA adapters Sheng et al. [2024a], Chen et al. [2024d]—in both cases, the new or concurrent models re-process the same popular contexts processed before. Chapter 4 examines how to make KV cache reuse work across such different models.

In short, longer contexts lead to higher prefill delays and hence longer TTFT, yet the same contexts are shared by many inputs, so the same expensive prefill is repeated again and again—once per query that carries the context. It is therefore promising to store the KV cache of a shared context when it is first processed and reuse it for later queries, skipping the repeated prefill entirely. Indeed, this idea has been explored recently Kwon et al. [2023b], Anonymous [2024], Gim et al. [2023b], Zheng et al. [2023b], Gim et al. [2024] and shown its potential. What has been missing is evidence of how much reuse actually happens in real world —evidence I present next.

2.2 Real-World Evidence for KV Reuse

KV cache has long been used in auto-regressive Transformer models to accelerate decoding by caching the K and V tensors generated at the prefill phase and reusing them in the decoding phase, *within a single user query* Vaswani et al. [2023]. This means that the lifecycle of KV cache, defined as the time between when it is initially generated and when it is last accessed, is limited to a single query.

Nowadays, as LLMs are used in more and more applications and the user base grows significantly, this single-query view no longer holds. The same long context, such as a system prompt, a retrieved document, or a code repository, is often shared across *different* queries, sometimes issued by millions of users Gim et al. [2024, 2023b], Zheng et al. [2023b], Jin et al. [2025a]. In these cases, reusing the KV cache across the different queries that share the same long context skips the re-processing of that context and, in turn, greatly reduces inference latency.

I observe this pattern in real deployments of LMCache Liu et al. [2025], Authors [2024], an open-source KV caching layer that enables KV cache to be stored in different storage devices, such as CPU memory or disk, and reused across queries. LMCache has been deployed in a wide range of enterprise services. Along the way, we also implemented a usage tracker, an opt-in telemetry that the service operators can voluntarily turn on, which records the size, reuse, and lifecycle of the stored KV cache, aggregated across many services and users over several weeks. Note that this usage tracker does not record any actual content of the user data; only the anonymized statistics of the KV cache are collected.

From the data collected by the usage tracker, we observe three interesting findings that together motivate the need for a KV cache management layer that enables KV cache to be moved out of GPU RAM. First, the KV cache generated by one query is increasingly reused by other queries and users, so it is worth keeping around rather than discarding after a single query. Second, the reusable KV cache quickly outgrows GPU memory, so it has to be stored

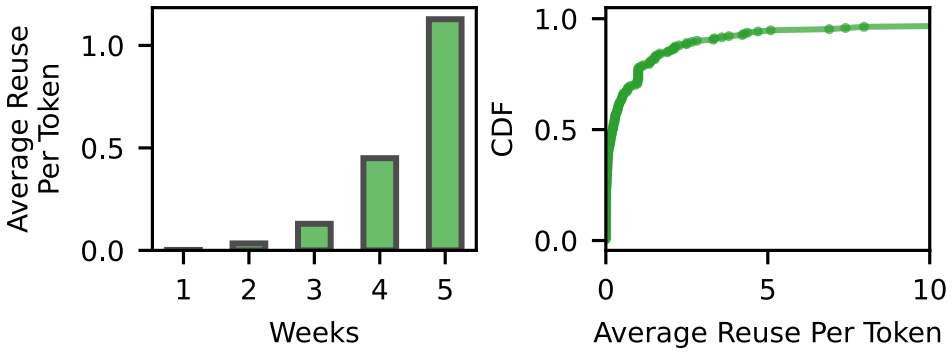


Figure 2.2: *Left: average reuse per token for the top-10 users, growing over successive weeks. Right: distribution of reuse per token across users, where more than 19% reuse a stored token over* 1.5×.

across a hierarchy of tiers beyond the GPU. Third, a KV cache is often reused long after it is generated, and how long it stays reusable should indicate which tier it sits on. We describe each finding in turn.

How often the KV cache is reused: Figure 2.2 plots the *reuse per token*, *i.e.,* the ratio between reused tokens and all stored tokens. For the top-10 users, the reuse per token keeps growing week over week (left), and across all users, more than 19% of them reuse a stored token over 1.5× (right). In other words, the KV cache generated by one query is increasingly likely to be useful to another query, or another user.

This clearly demonstrates that the KV cache generated by one query is increasingly useful to other queries. However, in order for the KV cache to be reused, it must be kept around after its initial generation, rather than discarded after a single query. This leads to a natural question: *given that a KV cache needs to be kept around (stored) for reuse, should we store it in GPU memory or elsewhere, and for how long?* We answer these two questions using more statistics gathered from the usage tracker.

Where to store the KV cache: The most natural place to keep the KV cache is the GPU's high-bandwidth memory (HBM), where it is created during prefill and where it will be used again when it is needed by another user query. The problem is that GPU memory is limited in capacity: modern GPUs have 80–140 GB of HBM, and when contexts are long

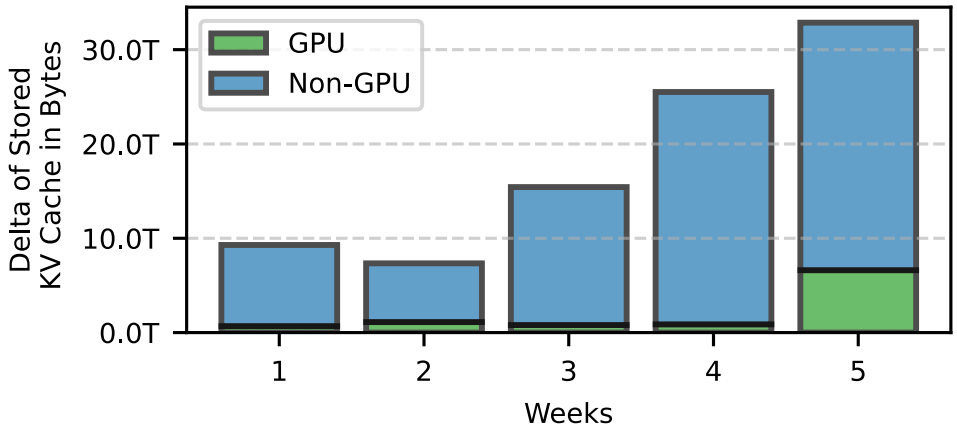


Figure 2.3: *Weekly growth of the stored KV cache, split into the part that fits in GPU memory and the part that exceeds it. The part that exceeds GPU memory grows the fastest.*

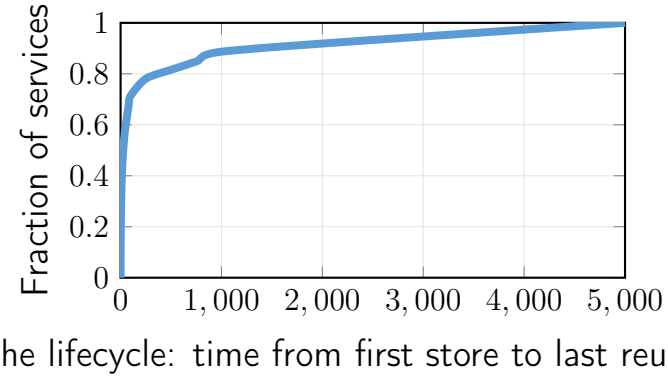


Figure 2.4: *Distribution of KV cache* lifecycle *— the elapsed time between when a cache is first stored and when it is last reused. About 40% of services re-access a cache at least one hour (60 min) after it is stored, and over 10% at least one day (1440 min) later; both far exceed the GPU residency of a cache.*

and many inference requests run concurrently, the KV cache can easily exceed the available GPU memory. Figure 2.3 shows the weekly growth of the stored KV cache, split into the part that fits in GPU memory and the part that exceeds it. From the figure, we can see that the part that exceeds GPU memory grows quickly, so keeping every reusable cache in the GPU is simply impractical. The reused KV cache thus has to be moved out of the GPU, *e.g.,* offloaded to CPU memory, local disk, or remote storage, and loaded back into HBM when its context is queried again.

How long to keep the KV cache: The second finding is that a KV cache is often reused

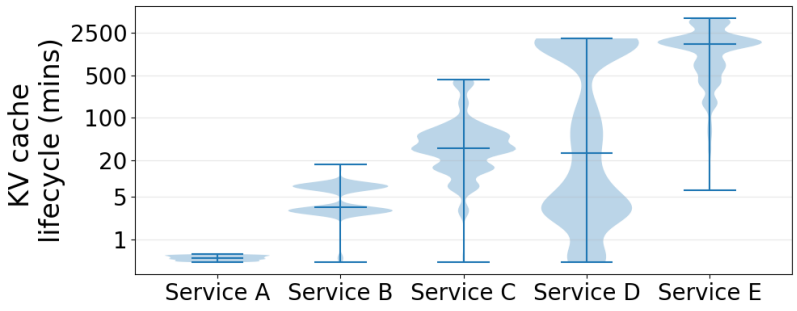


Figure 2.5: *Distribution of KV cache lifecycle across five production services (log scale). Lifecycles vary widely: some services reuse caches within minutes, while others reuse them hours or days later. This spread means the storage tier for a cache has to be chosen per service.*

long after it is generated. We define the *lifecycle* of a KV cache as the time between when it is first stored and when it is last reused. Figure 2.4 shows the distribution of lifecycles across services: about 40% of the services re-access a cache at least one hour after it is stored, and over 10% re-access it at least one day later. This is far longer than a cache can stay in GPU memory. As a concrete example, when serving Qwen3-Coder-480B-FP8 on 8×H100 with about 100 GB of GPU memory for the KV cache, at one query per second with 2K-token inputs, the ∼1 GB KV cache of a query is evicted from the GPU in about 100 seconds. A reuse one hour or one day later would then miss in GPU memory and pay the full prefill again, unless the cache had already been moved to a lower tier. Since the lifecycle is much longer than the time a cache can stay in GPU memory, the KV cache must live outside the GPU, and its lifecycle decides which tier it should sit on—short-lived caches in CPU memory, and long-lived caches on disk or remote storage.

More interestingly, there is no single lifecycle that fits all services. Figure 2.5 shows the lifecycle distribution of five example services, and the differences across them span more than three orders of magnitude. At one extreme, Service A re-accesses almost all of its caches within a minute, a pattern typical of multi-turn chat where consecutive queries in the same session arrive back to back; its caches only need a fast, short-lived tier such as GPU HBM. At the other extreme, Service E has a median lifecycle of over a day, closer to a document-

or repository-style workload where the same context is queried again much later; its caches are useless unless they survive on disk or remote storage. Between them, Services B and C sit at minutes to tens of minutes, and Service D is distinctly *bimodal*: one mass of caches is reused within a few minutes while another is reused more than a day later, meaning even a single service can mix both short- and long-lived caches. A single global retention policy would therefore be wrong for most services: a policy tuned to Service A would evict Service E's caches long before their reuse arrives, while a policy tuned to Service E would waste capacity holding Service A's caches for days after their last access. The placement and eviction decision thus has to be made per service based on the observed lifecycle distribution.

Put together, these statistics deliver one simple message: *the KV cache is likely to be re-used frequently, but reusable KV cache does not fit in GPU memory.* It is reused too many times to be discarded, too large to be held in HBM, and reused long after it would have been evicted from the GPU. The KV cache therefore should be *offloaded* to lower storage tiers and loaded back into the GPU when its context is queried again. Offloading, however, comes with a catch: reusing an offloaded cache pays off only if loading it back is faster than recomputing it from the context text, and that Section 2.4 formulates as the first challenge of this dissertation.

2.3 Beyond LLMs: ML Models Behind APIs

The background so far has focused on LLMs, but LLMs are only the newest family of models that software interacts with. Long before the LLM wave, applications had been building on *non-generative* models: image classification, object detection, sentiment analysis, entity recognition, text classification, and more.

ML models as a service: Traditionally, using such a model in an application meant designing, training, and managing it using frameworks like TensorFlow and PyTorch — collecting labeled training data, picking an architecture, tuning it, and provisioning hardware

to train and serve it — leaving a prohibitively high barrier for most application developers. The rise of ML APIs has greatly changed this landscape: cloud providers, including Google, Microsoft, and Amazon, train and host the models themselves and expose them through generic, pay-per-query APIs that reduce a common ML task (*e.g.,* object detection, facial emotion analysis) to just one invocation of an ML API within a single line of code McEnroe et al. [2022], Chang et al. [2021], Wan et al. [2021]. The developer never touches the model: no dataset, no training loop, no GPU—just a remote procedure call.

How developers use ML APIs: These APIs cover the most common traditional vision and language tasks. On the vision side, Google's `label_detection` and Amazon's `detect_labels` classify an image into categorical labels, and Google's `object_localization` additionally returns bounding boxes; on the language side, `analyze_sentiment`, `analyze_entities`, and `classify_text` score or categorize a piece of text. The output is a small structured value—typically a list of labels ranked by descending confidence score, or a pair of floating-point values (*e.g.,* a sentiment `score` and `magnitude`). This convenience gives rise to a variety of ML applications on smartphones, tablets, sensors, and personal assistants Wan et al. [2022], Chen et al. [2021a]: a garbage-classification application photographs a discarded item and calls an image-classification API to decide how to dispose of it Matthew [2019]; a plant-monitoring application checks whether a camera frame contains a plant Plant-Watcher; a food-delivery application runs sentiment analysis on restaurant reviews to triage negative feedback Martincu [2020]. In each case, the ML API largely eases developers' burden to deploy ML models.

One generic model serves every application: Behind each API, the service provider (*e.g.,* Google, Amazon) trains and serves one *generic* model for all the applications that invoke it. Genericity is what makes the economics work: a single image-classification model recognizing as many as 19.8K different labels can answer queries from the garbage classifier, the plant monitor, and thousands of other applications at once, and the provider optimizes

this one model for average accuracy across all the 19.8K labels. The application, on the other hand, has no channel to tell the provider what it actually needs—the API interface accepts an input image/audio/text and returns an output, nothing more.

Applications branch on API outputs: Applications do not merely display the API output; they *branch* on it, through a piece of logic that this dissertation refers to as the application's *decision process* (Chapter 5 defines it formally). The garbage-classification application above, for instance, maps the returned labels onto three pre-defined actions, with *many different labels leading to the same action*: `"Plastic"`, `"Glass"`, and `"Paper"` all route the item to `Recycle`, while `"Shirt"` and `"Jacket"` both route it to `Donate`. This many-to-few collapse means the model's raw accuracy and the application's accuracy are not the same thing: for a shirt image, misclassification as `"Jacket"` still yields the correct decision of `Donate`, whereas misclassification as `"Paper"` routes it to `Recycle`—a wrong decision. In other words, some API output errors are *critical* to a given application while others are harmless, and which is which is determined by the application's decision process—written in its source code, in a form the generic model never sees. Chapter 5 studies this mismatch empirically and shows how to resolve it.

2.4 Three Challenges Addressed in This Dissertation

Put together, this chapter has described two gaps between how models are served and how they are used. For LLMs, the KV cache of long, reused contexts must live outside the GPU, yet loading it back can be slower than recomputing it—and even fast loading only helps when the *same* model reuses the cache. For ML APIs, one generic model serves every application, yet each application has its own definition of which model errors matter. These gaps translate into three concrete challenges, which structure the rest of this dissertation.

Challenge 1: loading the KV cache can be slower than recomputing it: The loading bandwidth between secondary storage tiers and GPU is typically low (*e.g.,* below ten GBps),

while the KV cache consists of large floating-point tensors whose size grows with both the context length and the model size. To make it concrete, consider serving the 111B-parameter Cohere Command A model Cohere [2025] on 8×H100 GPUs, where a 100K-token context produces about 30 GB of KV cache: loading this cache back from a local SSD at 4 GB/s takes about 7.5 seconds, while recomputing it by re-running prefill takes only about 6 seconds. In other words, the "shortcut" of reusing the cache can be slower than the computation it is meant to skip—unless loading is made substantially faster, offloading is pointless. The first challenge is therefore to make loading fast. The key observation is that the KV cache does not have to stay a tensor during transfer: LLM inference requires the KV cache to be in its tensor format, but network transmission does not. Chapter 3 presents CacheGen, which reduces the size of the KV cache by encoding it into compact bitstreams and streaming them under varying bandwidth, so that the loading delay can be much shorter than recomputation even over slow links.

Challenge 2: the KV cache is not reusable across models: Even with loading made fast, a strong assumption of KV cache re-use still remains: KV cache can *only* be re-used if two requests are quering the *same* model. As LLM applications become more complicated, a single application increasingly involves multiple models—agentic pipelines where several fine-tuned variants of a base model process the same document, or serving fleets that route queries across many variants of one base model. Because the KV cache is the internal representation of a specific model, the cache produced by one model cannot be directly fed to another: doing so would greatly degrade generation quality, as shown in Chapter 4. The consequence is that when two models share the same context, the KV cache reuse rate across them drops to zero—every query served by the other model misses the cache and pays the full prefill, no matter how much the context text overlaps. Chapter 4 presents DroidSpeak, which enables the KV cache produced by one model variant to be reused by another at a fraction of the recomputation cost, so that reuse can happen even across model boundaries.

Challenge 3: generic model outputs are misaligned with application decisions: For the models behind ML APIs, the problem is not moving state but the model's ignorance of the software that consumes its output: the generic model is trained as if all errors were equal, while the application's decision process makes some errors critical and others harmless (§2.3). The third challenge is therefore to encode the application's decision process into the model itself, without changing the API interface or the application source code. Chapter 5 presents ChameleonAPI, which extracts the decision process from application source code, compiles it into an application-specific loss function, and re-trains the model behind the API so that the errors the model does make are the ones the application can tolerate.

In short, the three challenges correspond to the size, interoperability, and specialization challenges of model-native state laid out in Section 1.6: CacheGen makes the KV cache small enough to move fast, DroidSpeak makes it reusable across the models that increasingly make up a single application, and ChameleonAPI makes a model's behavior reflect the software context it serves. Together, they let AI applications interact with models through model-native state in an efficient and accurate manner.

CHAPTER 3

CACHEGEN: KV CACHE COMPRESSION AND STREAMING

3.1 Introduction

With impressive generative quality, large language models (LLMs) are ubiquitously used llm [a,b,c,d] in personal assistance, AI healthcare, and marketing. The wide use of LLM APIs (*e.g.,* OpenAI GPT-4 gpt) and the industry-quality open-source models (*e.g.,* Llama lla [a]), combined with popular application frameworks (*e.g.,* HuggingFace hug [2024], Langchain lan [a]), further boosts LLMs' popularity.

To perform complex tasks, users or applications often prepend the LLM inputs with *long contexts* containing thousands of tokens or more. In this paper, we regard a long prefix used by different LLM inputs as a context. For example, some context supplements user prompts with domain-knowledge text so that the LLM can generate responses using specific knowledge not embedded in the LLM itself. As another example, a user prompt can be supplemented with the conversation histories accumulated during the interactions between the user and the LLM. Though short inputs can still be useful Liu et al. [2023a], Sun et al. [2021a], longer inputs often improve response quality and coherence Beltagy et al. [2020], Tworkowski et al. [2023], Wu et al. [2022b], Dai et al. [2019b], Roy et al. [2021], Bertsch et al. [2023], Izacard and Grave [2021], Borgeaud et al. [2022], which drives the ongoing race to train LLMs that accept ever longer inputs, from 2K tokens in ChatGPT to 100K in Claude lon.

Using long contexts poses a challenge to the response-generation *latency*, as no response can be generated until the whole context is loaded and processed by the LLM. The amount of computation in processing a long context grows super-linearly with the context length Vaswani et al. [2023], Dao et al. [2022], Beltagy et al. [2020], Zaheer et al. [2021], Roy et al. [2021]. While some recent works increase the throughput of processing long con-

text Agrawal et al. [2023], the *delay* of processing the context can still be several seconds for long contexts (2 seconds for a 3K context) Agrawal et al. [2023], Gim et al. [2023b]. Many systems reduce the context-processing delay by storing and reusing the *KV cache* of the context to skip redundant computation when the context is used again (*e.g.,* Kwon et al. [2023b], Anonymous [2024], Gim et al. [2023b]).

Yet, the KV cache of a reused context may not always be in local GPU memory when the next input comes; instead, the KV cache may need to be retrieved from another machine(s) first, causing extra network delays. For instance, a database of background documents might reside in a separate storage service, and the documents (*i.e.,* context) assisting LLM inference are only to be selected and fetched to the LLM when a relevant query is received lan [b], arx, gen, aut, Beltagy et al. [2020].

The extra network delay for fetching the KV cache has not yet received much attention. Previous systems assume the KV cache of a context is always kept in the same GPU memory between different requests sharing the same context Gim et al. [2023b], or the KV cache is small enough to be sent quickly by a fast interconnection Zhong et al. [2024a]. Yet, as elaborated in §3.3, the delay for fetching a KV cache can be non-trivial, since a KV cache consists of large high-dimensional floating-point tensors, whose sizes grow with both the context length and model size and can easily reach 10s GB. The resulting network delay can be 100s milliseconds to over 10 seconds, hurting the interactive user experience lat [2023, 2021], Lew et al. [2018].

Some emergent systems reduce the *run-time* size of KV cache in order to fit in GPU memory constraint or LLM's input-window constraint. One line of prior works drop unimportant tokens from KV cache or context text Liu et al. [2023d], Zhang et al. [2023f], Jiang et al. [2023a,b], and another line of prior works apply smart quantization on KV cache Kang et al. [2024], Liu et al. [2024f], Hooper et al. [2024]. On the other hand, we want to reduce the *transmission-time* size of KV cache to reduce *network delay*, thus do not need to

Technique	KV cache size (in MB, Lower the better)	Accuracy (Higher the better)
CacheGen	176	0.98
8-bit quantization	622	1.00
CacheGen on H2O	71	0.97
H2O Zhang et al. [2023f]	282	0.97
CacheGen on LLMLingua	183	0.94
LLMLingua Jiang et al. [2023b]	492	0.94

Table 3.1: *Performance of CacheGen and the baselines on Mistral-7B with LongChat dataset Li* et al. [2023]. Full results are shown in §3.7.*

keep the tensor format of KV cache, and can encode it into more compact bitstreams. In short, when loading contexts from other machines, solely optimizing computational delay may cause *higher* response latency, as loading the KV cache increases the network delay.

We present CacheGen, a fast context-loading module in LLM systems for reducing the network delay in fetching and processing long contexts. It entails two techniques.

KV cache *encoding*: CacheGen encodes a precomputed KV cache into more compact *bitstream* representations, rather than keeping the tensor shapes of the KV cache. This greatly saves the bandwidth and delay for sending a KV cache. Our KV cache encoder employs a custom quantization and arithmetic coding strategy to leverage the distributional properties of KV cache, such as locality of KV tensors across nearby tokens and different sensitivities towards quantization losses at different layers of a KV cache. Moreover, the KV cache encoder incurs negligible compute overhead compared to LLM inference, and the encoding is pipelined with network transmission to minimize its impact on the end-to-end delay.

KV cache *streaming*: CacheGen streams the encoded bitstreams of a KV cache in a way that adapts to changes in available bandwidth. Similar to video streaming, before a user query arrives, CacheGen splits a long context into chunks and encodes the KV of each chunk separately at various compression levels. When streaming a context, CacheGen fetches the chunks one by one and adapts the per-chunk compression level to maintain high generation

quality while keeping the network delay within a Service-Level Objective (SLO). When the bandwidth is too low, CacheGen can also fall back to sending a chunk in text format and leave it to the LLM to recompute the KV cache of the chunk.

In short, unlike prior systems that optimize the KV cache in GPU memory, CacheGen focuses on the *network* delay for sending the KV cache. We compare CacheGen with a range of baselines, including KV quantization Sheng et al. [2023], loading contexts in text form, and state-of-the-art context compression Jiang et al. [2023b], Zhang et al. [2023f], using four popular LLMs of various sizes (from 7B to 70B) and three datasets of long contexts (662 contexts with 1.4 K to 16 K tokens) (Table 3.1 gives a preview of results). Our key findings are:

- In terms of the delay of transmitting and processing contexts (*i.e.,* time-to-first-token), CacheGen is 3.2-3.7× faster than the quantization baseline at the similar generation quality (F1 score and perplexity), and 3.1-4.7× faster than loading the text contexts with less than 2% accuracy drop. Notably, under lossless compression, CacheGen is able to reduce the delay of loading context by 1.67-1.81×.

- In terms of the bandwidth usage for sending KV cache, CacheGen achieves the same generation quality while using 3.5-4.3× less bandwidth than the quantization baseline.

- Compared with applying quantization on context compression methods Zhang et al. [2023f], Jiang et al. [2023b], CacheGen further reduces the bandwidth usage for sending their KV caches by 3.3-4.2×.

Our code is publicly available at: **https://github.com/UChi-JCL/CacheGen**.

3.2 Background and Motivation

Chapter 2 introduced the two trends that CacheGen builds on. First, LLM inputs increasingly carry long contexts—retrieved documents, code repositories, few-shot examples, chat

histories—and processing a long context inflicts a prefill delay that grows superlinearly with its length (§2.1). Second, the same contexts are often reused across different queries and users, as confirmed by the deployment statistics in Section 2.2. Together, the two make it promising to store the KV cache of a shared context and reuse it to skip the repeated prefill Kwon et al. [2023b], Anonymous [2024], Gim et al. [2023b], thereby reducing TTFT.

This chapter targets the point where that promise meets the storage hierarchy: the KV cache of a reused context often cannot stay in the GPU—or even the machine—where it was produced (§2.2), so it must be fetched over a network whose bandwidth is orders of magnitude below that of GPU memory. The next section quantifies this network bottleneck and explains why existing KV cache optimizations do not remove it.

3.3 The Hidden Network Bottleneck

While reusing the KV cache of a long context could drastically reduce TTFT, this benefit comes with an catch—the reused KV cache must be in the local GPU memory in the first place Kwon et al. [2023b], Anonymous [2024], Gim et al. [2023b], Zheng et al. [2023b].

Why KV cache needs to be loaded: In practice, however, the reused KV cache may need to be fetched from another machine(s). This is because GPU memory is likely not enough to store the KV caches of many repeated contexts. For example, in a financial assistance application, an LLM performs data analysis on long financial reports Nucci [2024], which can have thousands or tens of thousands of tokens, leading to a large KV cache size. To make it concrete, processing Amazon's annual report for 2023, which has ∼80,000 tokens Amazon.com Inc. [2023], with the model of Llama-34B produces a KV cache of 19 GB, which is on par with the size of the LLM itself. As different queries that reuse a KV cache may be several hours apart, the reused KV cache may have to be offloaded to make space for fresh chat sessions. Moreover, as newer LLMs can accept ever longer contexts Wu et al. [2024a], Lin et al. [2024a], Ge et al. [2024], Ding et al. [2024], Hsieh et al. [2024], storing

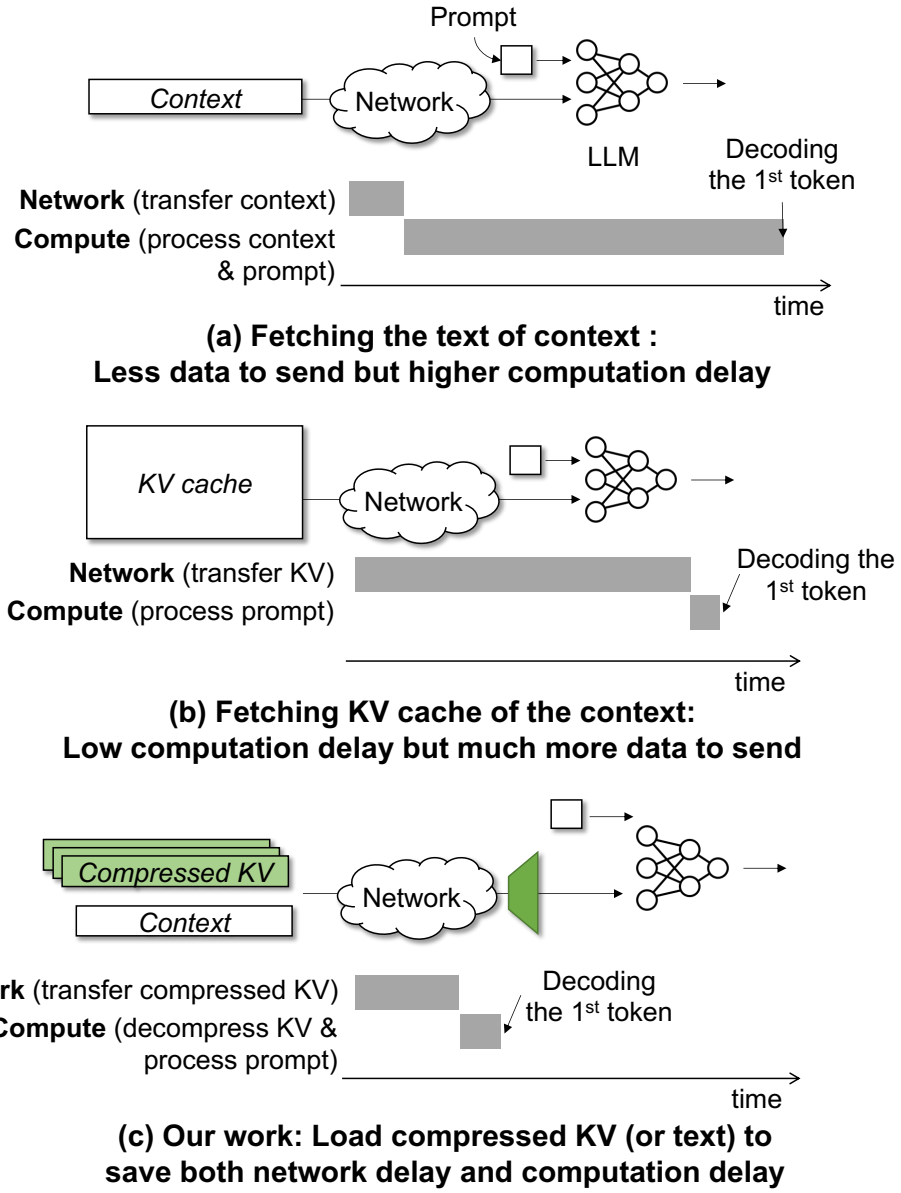


Figure 3.1: *How different ways of loading context affect the network delay (to transfer context or KV cache) and the computation delay (to run the attention module on the context).*

them on dedicated storage servers, rather than CPUs or GPU, would be more practical and economical. Besides, different requests that reuse KV cache may not always hit the same GPU, which also requires the KV cache to be moved between machines.

Fetching KV cache from another machine causes a substantial delay, yet this network delay has not received sufficient attention.

Is it a new problem? Although some recent efforts also send KV cache across GPUs to run multi-GPU inference, these systems assume that the KV cache is shared via high-speed links Zhong et al. [2024a], Patel et al. [2023a], *e.g.,* direct NVLinks, which has bandwidth of up to several hundred Gbps. In these settings, the network delay to fetch KV cache can be negligible. However, KV caches also need to be fetched over lower-bandwidth links, such as between regular cloud servers, where the bandwidth is usually in the single-digit Gbps range Jain et al. [2023b]. As illustrated in Figure 3.1b, in this setting, the delay of fetching KV cache into GPU memory can be as long as (or even longer than) prefill without the KV cache.

Our approach: This paper focuses on reducing the network delay in fetching the KV cache. To this end, we compress the KV cache by ***encoding*** it into more compact bitstream representations (shown in Figure 3.1c). This goal may seem similar to the recent works that drop words (tokens) from the text context or quantize the KV cache tensors Zhang et al. [2023f], Liu et al. [2023d], Hooper et al. [2024], Kang et al. [2024], Liu et al. [2024f]. However, there is a key difference. These techniques reduce the ***run-time*** GPU memory footprint of KV cache, thus retaining the tensor shapes of KV cache. In contrast, we reduce the ***transmission-time*** size of KV cache by encoding it into compact bitstreams to reduce the network delay of sending it. Moreover, there is a natural synergy—the KV cache shrunk by these recent works can still be encoded to further reduce the KV cache size and the network delay of sending KV caches.

3.4 CacheGen: KV Cache Encoding and Streaming

The need to reduce KV cache transmission delay motivates a new module in LLM systems, which we call a *KV cache streamer*. The KV cache streamer serves three roles:

- *(1) Encoding* a given KV cache into more compact bitstream representations — *KV bitstreams*. This can be done offline.
- *(2) Streaming* the encoded KV bitstream through a network connection of varying throughput.
- *(3) Decoding* the received KV bitstream into the KV cache.

At first glance, our KV cache streamer may look similar to recent techniques (*e.g.,* Jiang et al. [2023b], Zhang et al. [2023f], Liu et al. [2023d]) that compress long contexts by dropping less important tokens. Yet, they differ in crucial ways:

Those recent techniques aim at reducing the *run-time* size of the KV cache to accommodate the GPU memory-size constraint or LLM input-window constraint, and yet we aim at reducing the *transmission-time* size of the KV cache to reduce network delay. As a result, previous techniques have to maintain the KV caches' shapes of large floating-point tensors so that the shrunk KV caches can be directly consumed by the LLM at the run-time; meanwhile, they can use information during the generation phase to know which tokens in the context are more important to the particular query under processing. In contrast, we need *not* to maintain the original tensor shapes, and can encode them into more compact bitstreams and adapt their representation to network bandwidth. Meanwhile, we have to decide which compression scheme to use before a particular query is processed and hence cannot use information from the generation phase.

This paper presents **CacheGen**, a concrete design of the KV cache streamer. First, CacheGen uses a custom KV cache codec (encoder and decoder) to minimize the size of KV bitreams, by embracing several distributional properties of KV cache tensors (§3.5.1). This

greatly reduces the bandwidth demand to transmit the KV cache, thus directly reducing TTFT. Second, when streaming the KV bitstreams under dynamic bandwidth, CacheGen dynamically switches between different encoding levels or computing the KV cache on demand, in order to keep the TTFT within a given deadline while maintaining a high response quality. The KV encoding/decoding incurs a negligible compute overhead and is pipelined with network transmission to minimize the impact on end-to-end delay.

3.5 CacheGen Design

We now describe the design of CacheGen, starting with the insights on KV cache (§3.5.1) that inspires KV cache encoder (§3.5.2), followed by how CacheGen adapts to bandwidth (§3.5.3).

3.5.1 Empirical insights of KV cache

We highlight three observations on the characteristics of KV cache values. Though it is intrinsically hard to prove they apply to any LLM on any context, here, we use a representative workload to empirically demonstrate the prevalence of these observations. The workload includes two LLMs of different capacities (Llama-7B and Llama-13B) and LongChat dataset Li* et al. [2023] (which contains 100 long contexts between 9.2K and 9.6K tokens, randomly sampled from the whole set of 200 contexts), one of the largest datasets of long contexts. Details of this workload can be found in §3.7.1.

Token-wise locality

The first observation is about how the K and V tensor values change *across tokens* in a context. Specifically, we observe that

Insight 1. *Within the same layer and channel, tokens in closer proximity have more similar*

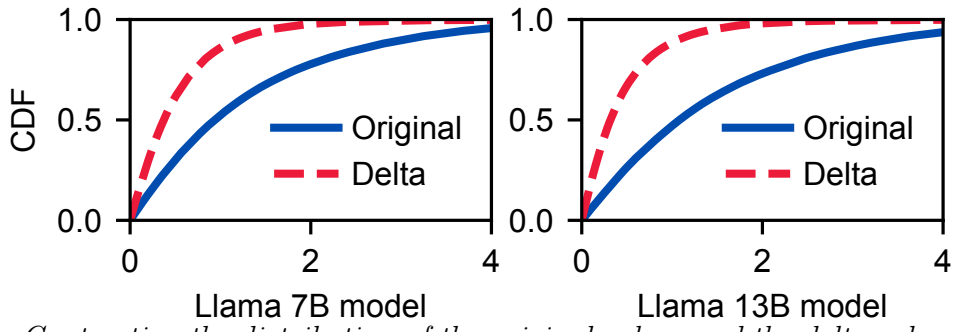


Figure 3.2: *Contrasting the distribution of the original values and the delta values. We model two Llama models with various long contexts (§3.5.1). We show absolute values for clarity.*

K/V tensor values compared to tokens that are further apart.

For each model, we contrast the distribution of K (or V) tensors' original values and the distribution of the *deltas*—the differences between of K (or V) tensors' values at the same layer and channel between every pair of consecutive tokens in the contexts. Figure 3.2 shows the distribution of absolute values in the original tensor and the deltas of one layer across all the contexts[1]. In both models across the contexts, we can see that the deltas are much more concentrated around zero than the original values. Consequently, the variance of the deltas is 2.4-2.9× lower than that of the original values. The token-wise locality of K and V tensors inspires CacheGen to encode deltas rather than original values.

This token-wise locality can be intuitively explained by the transformer's self-attention mechanism, which computes the KV tensors. Though KV cache is computed in a layer-by-layer manner, this is mathematically equivalent to calculating the KV tensors of one token based on the KV tensors of the previous token. This means KV tensors at one token are intrinsically correlated with those of the previous token.

1. We randomly sampled a single layer from the K tensor because the values in the different layers have different ranges.

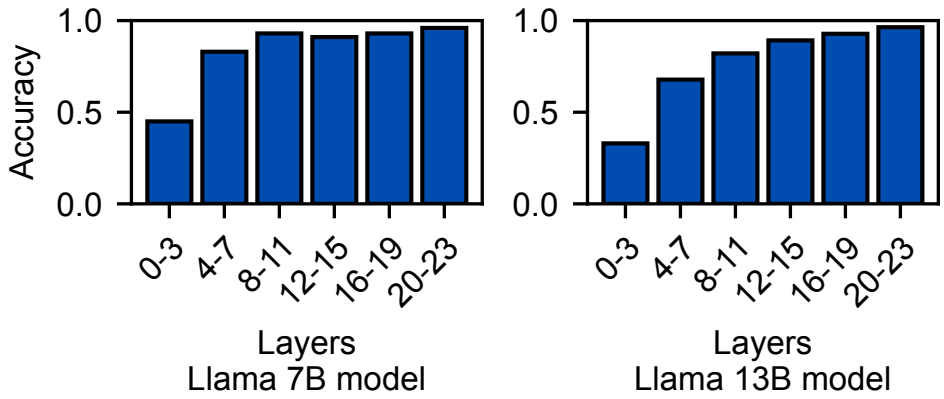


Figure 3.3: *Applying data loss to different layers of a KV cache has different impact on accuracy. (Same workload as Figure 3.2).*

Layer-wise sensitivity to loss

The second observation concerns how sensitive different values in the K and V tensors are to data loss. Our observation is the following.

Insight 2. *The output quality of LLM is more sensitive to losses in the KV cache values of the shallower layers than to those in the deeper layers.*

The heterogeneous loss sensitivity on different layers suggests that our KV cache encoder should compress different layers differently. Figure 3.3 shows how much accuracy is affected by applying data losses to the values of a specific layer group in the K and V tensors. Here, we apply rounding as the data loss, and we compute the average resulting response accuracy (defined in §3.7.1) across 100 contexts in the dataset. We can see that the average response accuracy drops significantly when the loss is applied to the early layers of a model while applying the same loss on the deeper layers has much less impact on the average response accuracy. This result holds consistently across different models we tested.

Intuitively, the deeper layers of a KV cache extract higher-level structures and knowledge than the shallower layers of a KV, which embed more primitive information Wang et al. [2023b], Shao et al. [2024]. As a result, the loss of information by removing precision on the early-layer cache might propagate and affect the later-layer cache and thus hinder the

model's ability to grasp the higher-level structures necessary to produce quality responses.

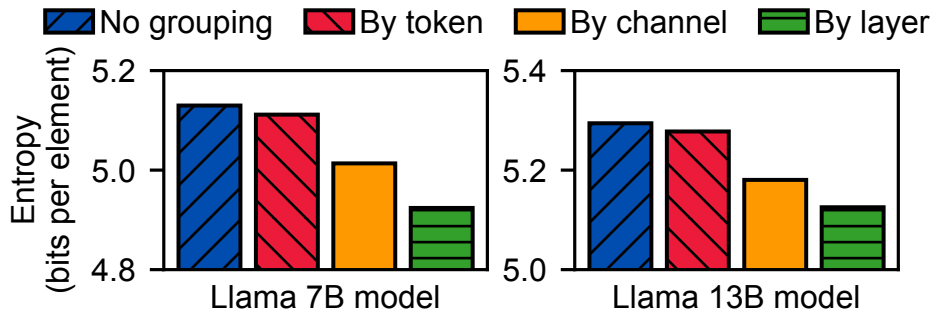


Figure 3.4: *Entropy (bits per element) when using different grouping strategies. (Same workload as Figure 3.2.) Grouping values by token position (red) reduces entropy much less than grouping by channel (yellow) or by layer (green).*

Distribution along layers, channels, and tokens

Finally, regarding the distributions of values along the three dimensions of KV cache—layers, channels, and token positions—we make the following observation.

Insight 3. *Each value in a KV cache is indexed by its channel, layer, and token position. The information gain of grouping values by their channel and layer is significantly higher than the information gain of grouping values by their token position.*

Intuitively, this can be loosely interpreted as: different KV values in the same channel (or layer) are more similar to each other than different KV values belonging to the same token position. A possible explanation is that different channels or layers capture various features in the input Lin et al. [2021], Devlin et al. [2018]. Some channels capture subject-object relationships, while others focus on adjectives. As for different layers, later layers capture more abstract semantic information than earlier ones according to prior works Lin et al. [2021], Devlin et al. [2018]. On the other hand, within a given layer and channel, the KV values for different tokens are similar, likely because of the self-attention mechanism, wherein

each token's KV is derived from all preceding tokens. We leave a more detailed examination to future work.

To show the insight empirically, we first group the values in the KV caches produced by the two models and 100 contexts based on their layers, channels, or token positions, and compute the entropy of each group. Figure 3.4 shows the average entropy (bits per element) when different grouping strategy is applied, including no grouping, grouping by tokens positions, grouping by channels, and grouping by layers.

3.5.2 KV cache encoding

The aforementioned insights inspire the design of CacheGen's KV cache encoder. The encoding consists of three high-level steps (elaborated shortly):

First, it calculates the *delta tensors* (defined later) between the K and V tensors of nearby tokens. This is inspired by the token-wise locality observation (§3.5.1) which suggests deltas between tokens might be easier to compress than the original values in the KV tensors.

Second, it applies different levels of quantization to different layers of the delta tensors. The use of different quantizations at different layers is inspired by the observation of heterogeneous loss sensitivity (§3.5.1).

Third, it runs a lossless arithmetic coder to encode the quantized delta tensors into bitstreams. Specifically, inspired by the observation in §3.5.1, the arithmetic coder compresses the values in each layer and channel separately (§3.5.1).

These steps may seem similar to video coding, which encodes pixels into bitstreams. Video coding also computes the delta between nearby frames, quantizes them, and encodes the delta by arithmetic coding Sze and Budagavi [2012]. Yet, blindly applying existing video codecs could not work well since they were only optimized for pixel values in natural video content. Instead, the exact design of CacheGen is inspired by domain-specific insights on LLM-generated KV cache (§3.5.1).

Next, we explain the details of each step.

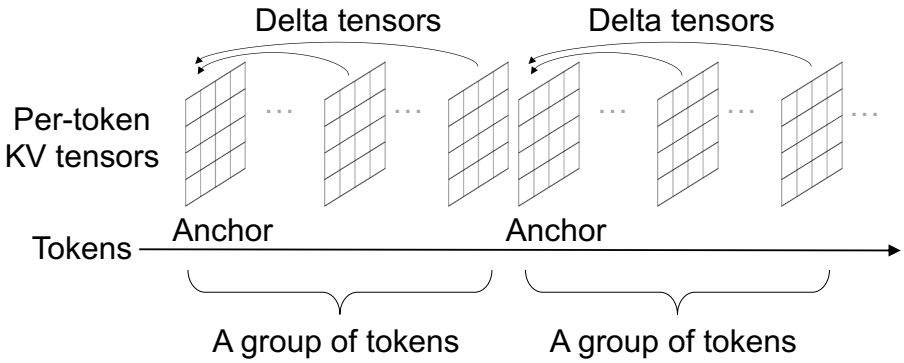


Figure 3.5: *Within a token group, CacheGen computes delta tensors between KV tensors of the anchor token and those of remaining tokens.*

Change-based encoding: To leverage the token-wise locality, we first split the context into *groups of tokens* each containing ten contiguous tokens. As shown in Figure 3.5, in each group, we independently (*i.e.,* without referencing other tokens) compress the KV tensor of the first token, called the *anchor token*, and then compress and record the *delta tensors* with respect to the anchor token for every other token.

This process is analogous to video coding, where the frames are separated into groups of *pictures*, within which it runs similar delta-based encoding. The difference, however, is that instead of compressing the delta between each pair of consecutive tokens, we reference the same anchor token for every token in the chunk. This allows us to do compression and decompression in parallel and saves time.

Layer-wise quantization: After partitioning the tokens into groups, CacheGen uses quantization to reduce the precision of elements (floating points) in a KV cache so that they can be represented by fewer bits. Quantization has been used recently to reduce attention matrices to pack longer contexts in GPU memory Sheng et al. [2023]. However, in previous work, elements are uniformly quantized with the same number of bits without leveraging any unique properties of KV cache. Driven by the insight of heterogeneous loss sensitivity (§3.5.1), we apply more conservative quantization (i.e., using more bits) on the delta tensors

of earlier layers. Specifically, we split the transformer layers into three layer groups, the first (earliest) 1/3 of layers, the middle 1/3 of layers, and the last 1/3 of layers, and apply different amounts of quantization bin size on the delta tensors at each layer group respectively. The size of the quantization bin grows larger (*i.e.,* larger quantization errors) from earlier to later layer groups. Following previous work Dettmers et al. [2022], we use the `vectorwise` quantization method, which has been usually used for quantizing model weights.

Note that we still use 8-bit quantization, a relatively high precision, on the KV cache of the anchor token (the first token of a token chunk). This is because these anchor tokens account for a small fraction of all tokens, but their precision affects the distribution of all delta tensors of the remaining tokens in a chunk. Thus, it is important to preserve higher precision just for these anchor tokens.

Arithmetic coding: After quantizing the KV cache into discrete symbols, CacheGen uses *arithmetic coding* Witten et al. [1987] (AC) to losslessly compress the delta tensors and anchor tensors of a context into bitstreams. Like other entropy coding schemes, AC assigns fewer bits to encode more frequent symbols and more bits to encode less frequent symbols. For it to be efficient, AC needs *accurate, low-entropy* probability distributions of the elements in the KV cache.

Driven by the observation of the KV value distributions along layers, channels, and token positions (§3.5.1), we group KV values by channel and layer to obtain probability distributions. Specifically, our KV encoder offline profiles a separate probability distribution for each channel-layer combination of delta tensors and another for anchor tensors produced by an LLM, and uses the same distributions for all KV caches produced by the same LLM. CacheGen uses modified AC library Mentzer et al. [2019] with CUDA to speed up encoding and decoding (§3.6). In §3.7.5, we empirically show that our method reduces the bitstream size by up to 53% compared to the strawman of using one global symbol distribution.

3.5.3 KV cache streaming adaptation

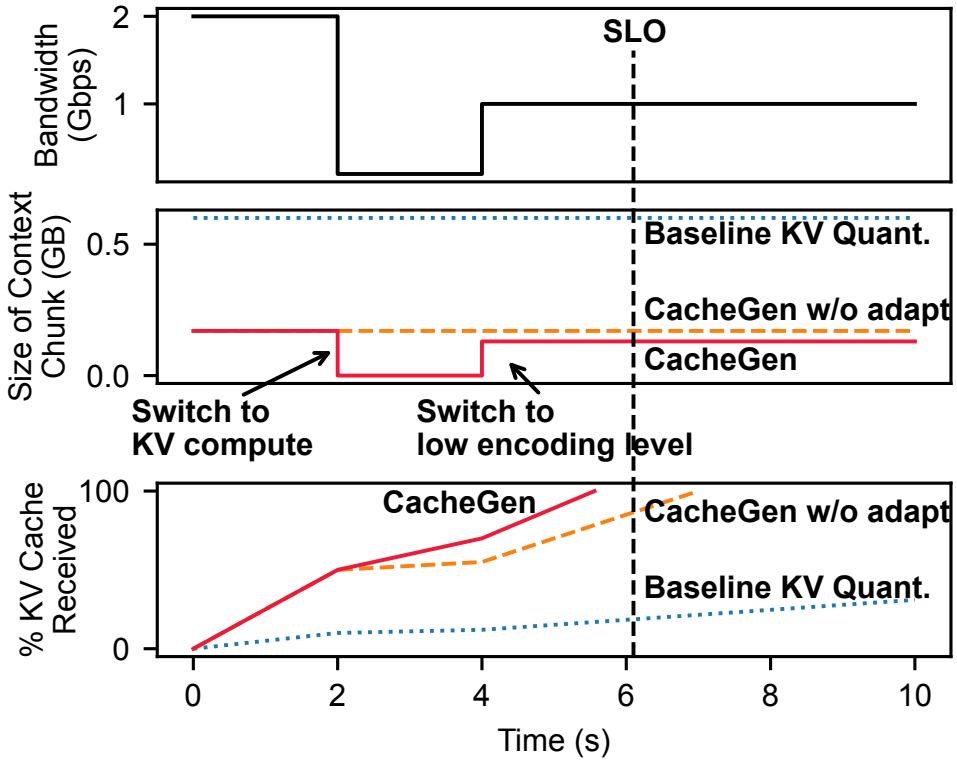


Figure 3.6: *Time Series demonstrating CacheGen's adaptation logic under bandwidth variation.*

Since the transmission of a KV cache may take up to hundreds of milliseconds to a few seconds, the available bandwidth may fluctuate during a transmission. Thus, streaming the encoded KV bitstreams at a fixed encoding level may violate a given service-level objective (SLO) Beyer et al. [2016] of fetching the KV cache.[2] In Figure 3.6, for example, at the start of the transmission, the available throughput is 2 Gbps, and if the bandwidth remains at 2 Gbps, sending a KV stream of 1 GB can meet the SLO of 4 seconds. However, at $t = 2s$, the throughput drops to 0.2 Gbps and only increases to 1 Gbps at $t = 4s$, so the actual transmission delay increases from 4 seconds to 7 seconds, which violates the SLO.

Workflow: To handle variations in bandwidth, CacheGen splits a context into multiple

2. In practice, SLO is defined on TTFT. Once the KV cache of the long context is loaded in GPU, the remaining delay of one forward pass is marginal Kwon et al. [2023b].

context chunks (or **chunks** for short) of consecutive tokens and uses the KV cache encoder to encode each chunk into multiple bitstreams of different encoding (quantization) levels that can be decoded independently (explained shortly). This can be done offline. When fetching a context, CacheGen sends these chunks one by one, and each chunk can choose one of several *streaming configuration* (or **configurations** for short): it can be sent at one of the encoding levels or can be sent in the text format to let the LLM recompute K and V tensors.

CacheGen adapts the configuration of each chunk while streaming the KV cache to keep the transmission delay within an SLO. Figure 3.6 illustrates an example adaptation where CacheGen switches to sending text context and recomputing KV cache from the text at $t = 2s$ due to the bandwidth drop, and at $t = 4s$, since the bandwidth increases back to 1 Gbps, and CacheGen switch to sending KV bitstreams of subsequent chunks at smaller size. With our adaptation logic (specific algorithm in §A.3.1), CacheGen can meet the SLO.

However, to adapt efficiently, several questions remain.

First, *how to stream multiple chunks at different streaming configurations without affecting compression efficiency?* To encode the chunks offline, CacheGen first computes the KV cache of the entire context (*i.e.,* prefill) and splits the K and V tensors of the KV cache along the token dimension into sub-tensors, each of which contains the layers and channels of the tokens in the same chunk. It then uses the KV encoder to encode the K or V sub-tensor of a chunk with different encoding (quantization) levels. Each chunk is encoded *independent* to other chunks *without* affect the compression efficiency as long as a chunk is longer than a group of tokens. This is because encoding the KV tensor of a token only depends on itself and its delta with the anchor token of the group of tokens (§3.5.2). Thus, chunks sent with different encoding levels can be independently decoded and then concatenated to reconstruct the KV cache. In the case that a chunk is sent in text format, the LLM will compute its K and V tensors based on the previous chunk's KV tensors that have been received and

decoded.[3]

Would streaming chunks at different configurations affect generation quality? If one chunk is sent at a smaller-sized encoding level than other chunks (due to low bandwidth), it will have high compression loss on *that* single chunk, but this will not affect the compression loss of other chunks. That said, we acknowledge that if the bandwidth is too low to send most chunks in a high encoding level, the quality will still suffer.

Second, *how long should a context chunk be?* We believe that the chunk length depends on two considerations.

1. The encoded KV bitstream of a chunk size should not be too big because, otherwise, it cannot react to bandwidth changes in a timely manner.
2. The chunk should not be too small either since then we can not fully utilize the batching ability of GPU to compute KV tensors if text format is chosen.

With these considerations in mind, we empirically pick 1.5K tokens as the default chunk length in our experiments[4], though more optimization may find better chunk lengths.

Thirdly, *how does CacheGen decide the streaming configuration of the next chunk?* CacheGen estimates the bandwidth by measuring the throughput of the previous chunk. It assumes this throughput will remain constant for the remaining chunks and calculates the expected delay for each streaming configuration accordingly. If there are bandwidth fluctuations, CacheGen's reaction will be delayed by at most one chunk. Since one chunk is a small subset of the entire KV cache, this reaction is sufficiently fast to meet SLO (details in §3.7.4). The delay of sending an encoded KV bitstream is calculated by dividing its size by the throughput (more details in §A). It then picks the configuration that has the least compression loss (*i.e.,* text format or lowest encoding level) with an expected delay still within

3. A similar concept has been used to split LLM input into prefill chunks for more efficient batching Agrawal et al. [2023].

4. The chunk length is also long enough for the KV bitstream of each chunk to fill the sender's congestion window in our experiment setting.

the SLO, and uses the configuration to send the next chunk. For the first chunk, if some prior knowledge of the network throughput is available, CacheGen will use it to choose the configuration of the first chunk the same way. Otherwise, CacheGen starts with a default medium encoding level (140 MB per chunk for Llama 7B, detailed setting in §A.3.2).

Finally, *how does CacheGen handle the streaming of multiple requests?* When multiple requests arrive concurrently within T seconds, CacheGen batches and streams them together. It can batch up to B requests, which is the maximum number that the GPU server can handle simultaneously. Each request is divided into chunks of the same size, even though the total number of chunks may differ among requests. For each chunk index c, CacheGen determines the number of requests N_c that include chunk c. Using the throughput measured for the previous chunk $c-1$, CacheGen calculates the expected delays for each configuration by multiplying N_c by the delay for a single request. On the GPU servers, the requests are batched by padding their KV caches and processing them together.

3.6 Implementation

We implement CacheGen with about 2K lines of code in Python, about 1K lines of CUDA kernel code, based on PyTorch v2.0 and CUDA 12.0.

Integration into LLM inference framework: CacheGen operates the LLM through two interfaces:

- `calculate_kv(context) -> KVCache`: given a piece of context, CacheGen invokes LLM through this function to get the corresponding KV cache.
- `generate_with_kv(KVCache) -> text`: CacheGen passes a KV cache to the LLM and lets it generate the tokens while skipping the attention computation (prefill) of the context.

We implement these two interfaces in HuggingFace models using the `transformers` li-

brary hug [b] with about 500 lines of Python code. Both interfaces are implemented based on the `generate` function provided by the library. For `calculate_kv`, we let LLM only calculate the KV cache without generating new text, by passing the options of `max_length = 0` and `return_dict_in_generate = True` when getting the KV cache. The `generate_with_kv` is implemented by simply passing the KV cache through the `past_key_values` argument when calling the `generate` function. Similar integrations are also applicable to other LLM libraries, such as FastChat Zheng et al. [2023a], llama.cpp lla [b], and GGML ggm.

We have also integrated CacheGen in LangChain lan [a], a popular LLM application framework. CacheGen is activated in the `_generate` function of LangChain's `BaseLLM` module. CacheGen first checks whether the KV cache of the current context already exists (explained shortly). If so, CacheGen will invoke `generate_with_kv` to start generating new texts. Otherwise, CacheGen will invoke `calculate_kv` to create the KV cache first before generating new texts.

KV cache management in CacheGen: To manage the KV cache, CacheGen implements two modules:

- `store_kv(LLM) -> {chunk_id: encoded_KV}`: calls `caculate_kv`, splitting the returned KV cache into context chunks, and encodes each chunk. Then, it stores a dictionary on the storage server, where it maps the `chunk_id` to the encoded bitstreams for the K and V tensors for the corresponding chunk.
- `get_kv(chunk_id) -> encoded_KV` fetches the encoded KV tensors corresponding to `chunk_id` on the storage server and transmits it to the inference server.

Whenever a new piece of context comes in, CacheGen first calls `store_kv`, which first generates the KV cache, and then stores the encoded bitstreams on the storage server. At run time, CacheGen calls `get_kv` to fetch the corresponding chunk of KV cache and feed into `generate_with_kv`.

Speed optimization for CacheGen: To speed up the decoding of KV cache in GPU, we pipeline the transmission of context chunk i with the decoding of context chunk $i-1$. We implemented a GPU-based AC library Mentzer et al. [2019] with CUDA to speed up encoding and decoding. Specifically, each CUDA thread is responsible for decoding the KV cache from bitstreams for one token. The probability distributions are obtained by counting the frequencies of quantized symbols in the KV feature for the corresponding context.

3.7 Evaluation

The key takeaways of our evaluation are:

- Across four datasets and models, CacheGen can reduce TTFT (including both network and compute delay) by 3.1-4.7× compared to prefill from text context, and by 3.2-3.7× compared to the quantization baseline (§3.7.2).
- CacheGen's KV encoder reduces the bandwidth for transferring KV cache by 3.5-4.3× compared to the quantization baseline (§3.7.2).
- CacheGen's reduction in bandwidth usage is still effective when applied to recent context compression baselines Zhang et al. [2023f], Jiang et al. [2023b]. CacheGen further reduces the bandwidth usage by 3.3-4.2×, compared to applying quantization on context compression baselines (§3.7.2).
- CacheGen's improvement is significant across various workloads, including different context lengths, network bandwidths, and numbers of concurrent requests (§3.7.3).
- CacheGen's decoding overhead is minimal, in delay and compute, compared with LLM inference itself (§3.7.5).

Dataset	Size	Med.	Std.	P95
LongChat Li* et al. [2023]	200	9.4K	164	9.6K
TriviaQA Joshi et al. [2017]	200	9.3K	4497	15K
NarrativeQA Kočiský et al. [2017]	200	14K	1916	15K
WikiText Merity et al. [2016]	62	5.9K	4548	14.8K

Table 3.2: *Size and context lengths of datasets in the evaluation.*

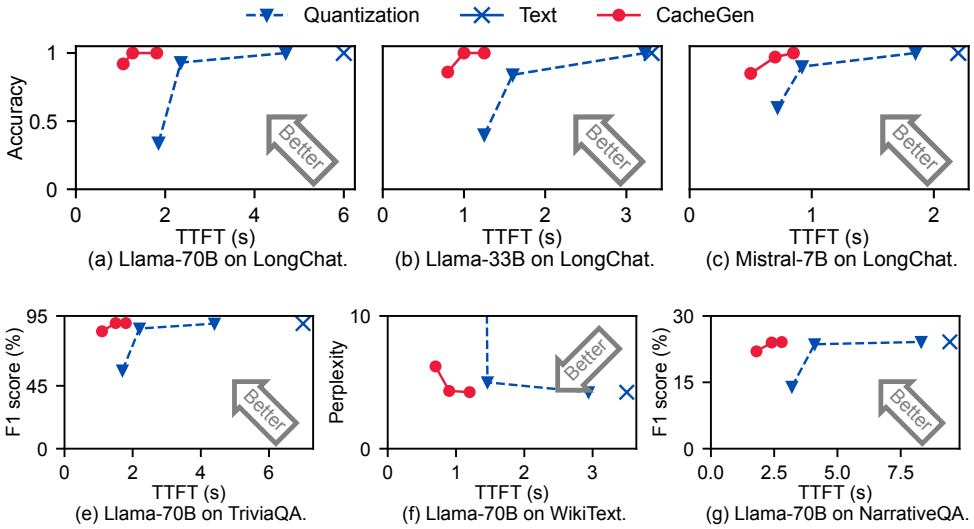


Figure 3.7: **Time-to-first-token (TTFT):** *Across different models and different datasets, CacheGen reduces TTFT with little negative impacts on quality (in accuracy, perplexity or F1 score).*

3.7.1 Setup

Models: We evaluate CacheGen on four models of different sizes, specifically the fine-tuned versions of Mistral-7B, Llama-34B, Llama-70B. All models are fine-tuned such that they can take long contexts (up to 32K). We did not test CacheGen on other LLMs (*e.g.*, OPT, BLOOM) because there are no public fine-tuned versions for long contexts to our best knowledge.

Datasets: We evaluate CacheGen on 662 contexts from four different datasets with different tasks (Table 3.2):

- *LongChat:* The task is recently released Li* et al. [2023] to test LLMs on queries like "What was the first topic we discussed?" by using all the previous conversations as the

context. Most contexts are around 9-9.6K tokens.

- *TriviaQA:* The task tests the reading comprehension ability of the LLMs Bai et al. [2023], by giving the LLMs a single document (context), and letting it answer questions based on it. The dataset is part of the LongBench benchmark Bai et al. [2023] suite.
- *NarrativeQA:* The task is used to let LLMs answer questions based on stories or scripts, provided as a single document (context). The dataset is also part of LongBench.
- *Wikitext:* The task is to predict the probability of the next token in a sequence based on the context consisting of relevant documents that belong to a specific Wiki page Merity et al. [2016].

The dataset we used to design CacheGen's encoder is a subset of the datasets we used to evaluate CacheGen. This is for showing the insights in §3.5.1 are generalizable to different datasets.

Quality metrics: We measure generation quality using the standard metric of each dataset.

- *Accuracy* is used to evaluate the model's output on the LongChat dataset. The task predicts the first topic in the conversational history between the user and the LLM. The accuracy is defined as the percentage of generated answers that exactly includes the ground-truth topic.
- *F1 score* is used to evaluate the model's response in the TriviaQA and NarrativeQA datasets. It measures the probability that the generated answer matches the ground-truth answer of the question-answering task.
- *Perplexity* is used to evaluate the model's performance on the Wikitext dataset. The perplexity is defined as the exponentiated average negative log-likelihood of the next token Azzopardi et al. [2003], Chen et al. [2008]. A low perplexity means that the model likely generates the next token correctly. While perplexity does not equate to text-generation quality, it is widely used as a proxy Adiwardana et al. [2020] to test

the impact of pruning or quantizing LLMs on generation performance Dettmers et al. [2022], Xiao et al. [2023], Liu et al. [2023c], Roy et al. [2021].

System metrics: We compare CacheGen with baselines with two system-wise metrics.

- *Size of KV cache* is the size of the KV cache after compression, this measures the bandwidth needed to load KV caches.
- *Time-to-first-token (TTFT)* is the time from the arrival of the user query to the generation of the first token. This includes the loading delay of the KV cache and the prefill delay of the new questions. This is a metric widely used in industry Anyscale Team [2023], Kadous et al. [2023], Agarwal et al. [2023] and recent works Gim et al. [2023b], Liu et al. [2024a].

Baselines: We compare CacheGen with baselines that do not change the contexts or model (more baselines in §3.7.5).

- *"Default quantization"* uses the uniform quantization of KV cache, specifically the same quantization level (*i.e.,* 3, 4, 8 bits) for every layer in the LLM (which was used in Sheng et al. [2023]).
- *"Text context"* fetches the text of the context and feeds it to LLM to generate the KV cache for it. It represents the design of minimizing data transmission but at the expense of high computation overhead. We use the state-of-the-art inference engine, vLLM Kwon et al. [2023b], to run the experiments. vLLM's implementation already uses xFormers Lefaudeux et al. [2022], which includes speed and memory-optimized Transformers CUDA kernels and has shown much faster prefill delay than HuggingFace Transformers. This is a very competitive baseline.
- *"Context compression"* either drops tokens in the text context (LLMlingua Jiang et al. [2023b]) or in the KV cache (H2O Zhang et al. [2023f]).

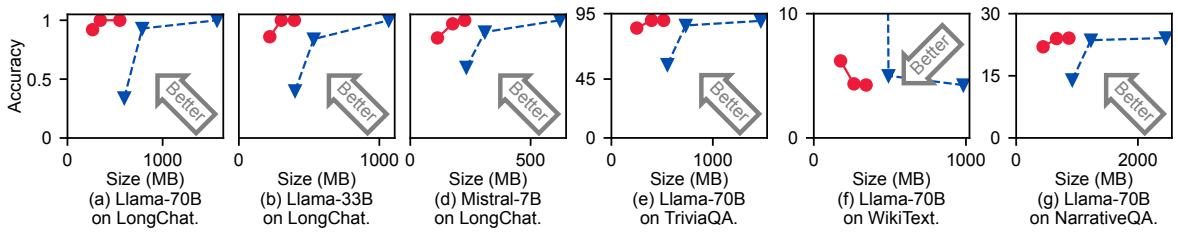


Figure 3.8: **Reducing KV cache size:** *Across various models, CacheGen reduces size of KV cache with little accuracy decrease on various datasets.*

Hardware settings: We use an NVIDIA A40 GPU server with four GPUs to benchmark our results. The server is equipped with 384GB of memory and two Intel(R) Xeon(R) Gold 6130 CPUs with Hyper-threading and Turbo Boost enabled by default.

3.7.2 Overall improvement

We first show the improvement of CacheGen over the baselines, as described in §3.7.1.

TTFT reduction: Figure 3.7 demonstrate CacheGen's ability to reduce TTFT, across four models and four datasets. Under bandwidth of 3 Gbps, compared to text context, CacheGen is able to reduce TTFT by 3.1-4.7×. Compared to default quantization, CacheGen is able to reduce TTFT by 3.2-3.7×.

It is important to note that even when the compression is lossless, specifically applying our AC on the KV cache after 8-bit quantization, CacheGen can still reduce the TTFT by 1.67-1.81×. CacheGen's reduction in TTFT is a result of a shorter transmission delay to send the smaller KV caches.

Reduction on KV cache size: Figure 3.7 show that, across three datasets and four models, CacheGen's KV encoder reduces the KV cache size by 3.5-4.3× compared to default quantization when achieving similar performance for downstream tasks after decoding. Thus, it achieves better quality-size trade-offs across different settings. The degradation caused by lossy compression is marginal—the degradation is no more than 2% in accuracy, less than 0.1% in F1 score, and less than 0.1 in perplexity hug [a].

Some example text outputs for different baselines are available in §A.

Gains over context compression baselines: We also apply CacheGen to further reduce the size of context compression baselines's KV cache, including H2O and LLMlingua. Note that H2O drops tokens from KV cache which have low attention scores. Specifically, it requires the query tensors of the prompt to compute the attention scores in order to determine which tokens to drop. The query tensors of the prompts are not present in the offline compression stage. In our experiments, we implement an *idealized* version of H2O, where the query tensors of the prompts are used in the offline compression stage.

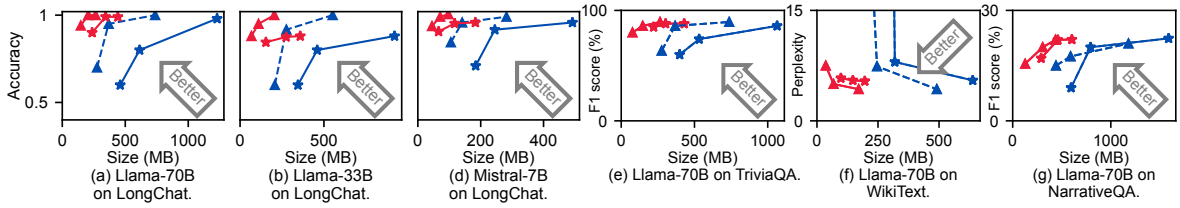


Figure 3.9: **Reducing KV cache size *on top of* H2O Zhang et al. [2023f] and LLMlingu Jiang et al. [2023b]:** *Across different models, CacheGen further the size of KV cache, compared to the KV cache shortened by H2O, with little accuracy decrease on different datasets.*

As shown in Figure 3.9, compared to the context compression baseline, H2O Zhang et al. [2023f], CacheGen can further reduce compressed KV cache (in floating point). Specifically, CacheGen reduces the size of KV cache by 3.5–4× compared to the H2O's quantized KV caches, and 3.3–4.2× compared to LLMlingu's quantized KV caches, without losing quality. This suggests that even after condensing contexts by H2O and LLMlingua, the resulting KV caches may still have the statistical observations behind CacheGen's KV encoder. Thus, the techniques used in CacheGen's encoder remain beneficial when we encode the KV cache after applying these techniques.

Understanding CacheGen's improvements: CacheGen outperforms various baselines for slightly different reasons. Compared to the text context baseline, CacheGen has lower TTFT, because it reuses KV cache to avoid the long prefill delay for processing long contexts.

Compared to the basic quantization baseline, CacheGen compresses KV cache with layer-wise dynamic quantization and further encodes the KV cache tensors into bitstreams, thus able to reduce the transmission delay.

Finally, compared to H2O and LLMlingua, two recent context-condensing techniques, CacheGen can still compress the KV cache produced by H2O. In short, H2O and other context-condensing techniques all prune contexts at token level and their resulting KV caches are in the form of floating-point tensors, so CacheGen is complementary and can be used to further compress the KV cache into much more compact bitstreams.

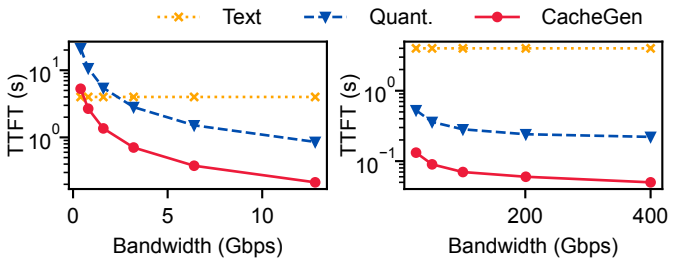


Figure 3.10: *CacheGen improves TTFT under a wide range of different bandwidths. Plotted with Mistral-7B. y-axis is log scale.*

3.7.3 Sensitivity analysis

Available bandwidth: The left and right figures in Figure 3.10 compares the TTFT of CacheGen with baselines under a wide range of bandwidth from 0.4–10 Gbps and 10–400 Gbps, while we fix the context length at 16K tokens. We can see that CacheGen consistently outperforms baselines under almost all bandwidth situations. Arguably, the *absolute reduction* in TTFT becomes smaller under high bandwidth (over 20Gbps), compared to the quantization baseline, since both the quantization baseline and CacheGen can transfer KV caches much faster.

Number of concurrent requests: Figure 3.11a shows the TTFT under different numbers of concurrent requests. When the number of concurrent requests increases (*i.e.,* fewer

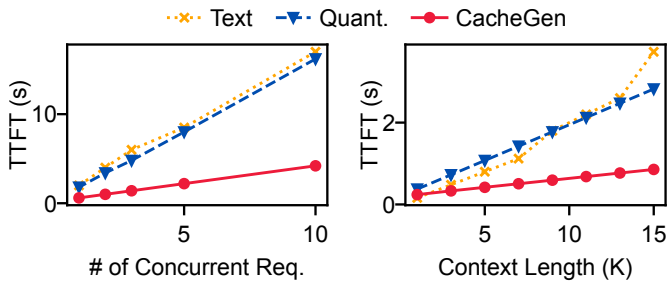


Figure 3.11: *CacheGen consistently reduces TTFT when there are multiple concurrent requests on one GPU. Plotted with Mistral-7B.*

available GPU cycles for one individual query), CacheGen significantly reduces TTFT than the baselines. This is because the amount of computation required for prefilling on a long input (9.6K in this case) is huge, as discussed in §3.2. §A.4 shows CacheGen's improvement over a complete space of workloads of different bandwidth and GPU resources.

Context lengths: Figure 3.11b compares CacheGen's TTFT with the baselines under different input lengths from 0.1K to 15K tokens under a fixed network bandwidth of 3 Gbps. When the context is long, the gain of CacheGen mainly comes from reducing the KV cache sizes. And when the context is short (below 1K), CacheGen will automatically revert to loading the text context as that yields a lower TTFT.

3.7.4 KV streamer adaptation

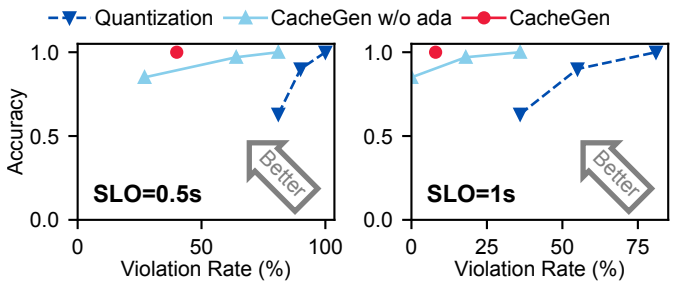


Figure 3.12: *CacheGen reduces SLO violation rate over CacheGen without adaptation and the quantization baseline. Plotted with Mistral-7B model.*

The adaptation logic described in §3.5.3 allows CacheGen to adapt to bandwidth changes and achieve good quality while meeting the SLO on TTFT. In Figure 3.12, we generate bandwidth traces where each context chunk's bandwidth is sampled from a random distribution of 0.1 – 10 Gbps. Each point is averaged across 20 bandwidth traces on the LongChat dataset. We can see that CacheGen significantly outperforms the quantization baseline and CacheGen without adaptation. Specifically, given an SLO on the TTFT of 0.5s, CacheGen reaches the same quality as the quantization baseline with an 60% lower SLO violation rate. Under an SLO of 1s, CacheGen reaches the same quality as the quantization baseline, while reducing the SLO violation rate from 81% to 8%. The reason why CacheGen has a lower SLO violation rate is that when the bandwidth drops, CacheGen can dynamically reduce the quantization level or fall back to the configuration of computing text from scratch, while the quantization baseline and CacheGen without adaptation cannot.

3.7.5 Overheads and microbenchmarks

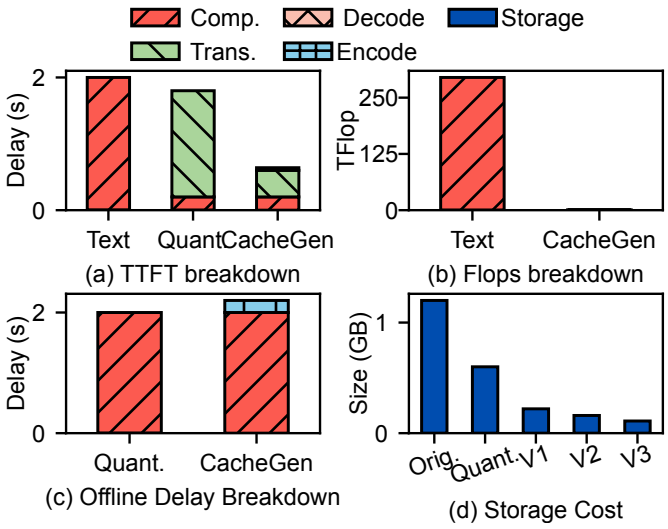


Figure 3.13: *(a) The breakdown of TTFT for text context, quantization baseline, and CacheGen for Llama 13B. (b) Computation overhead of the text baseline and CacheGen. (c) Offline delay breakdown for baseline quantization and CacheGen. (d) The storage cost for CacheGen, quantization baseline and the uncompressed KV cache.*

Decoding overhead: While having a better size-quality and TTFT-quality trade-off, CacheGen requires an extra decoding step compared to the quantization baseline. CacheGen minimizes the decoding overhead by pipelining the decoding of context chunks with the transmission of the context chunks, so as shown in Figure 3.13a, the decoding has minimal impact on the end-to-end delay. It is also important to note that although CacheGen's decoding is performed on GPU (see §3.6), the amount of computation needed by CacheGen's decoding module is negligible compared to the baseline that generates KV cache from text context.

Offline encoding and storage overheads: Unlike prior methods that compress each context only once, CacheGen compresses it into multiple versions (§3.5.3). CacheGen compresses each context almost as fast as the baselines because the encoding delay is very small (200 ms), as shown in Figure 3.13c. Figure 3.13d evaluates the overhead in storage. We can see that despite needing to encode and store multiple bitstream representations, the total storage cost for CacheGen is on par with the quantization baseline.

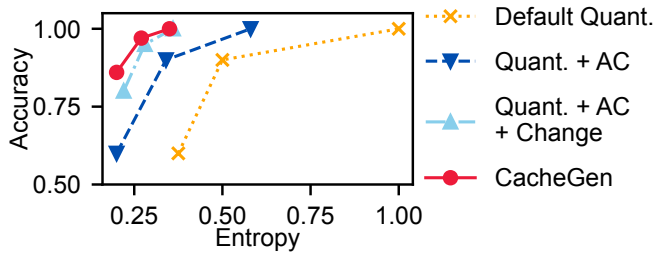


Figure 3.14: *Contributions of individual ideas behind KV encoder: change-based encoding, layer-wise quantization, and AC based on channel-layer grouping.*

Ablation Study: To study the impact of individual components in CacheGen's KV encoder, Figure 3.14 progressively adds each idea into the baseline of uniform quantization and default AC, starting with the use of our AC that uses probability distribution for each channel-layer combination, then change-based encoding, and finally layer-wise quantization. As shown in the figure, CacheGen's AC and change-based encoding significantly improve

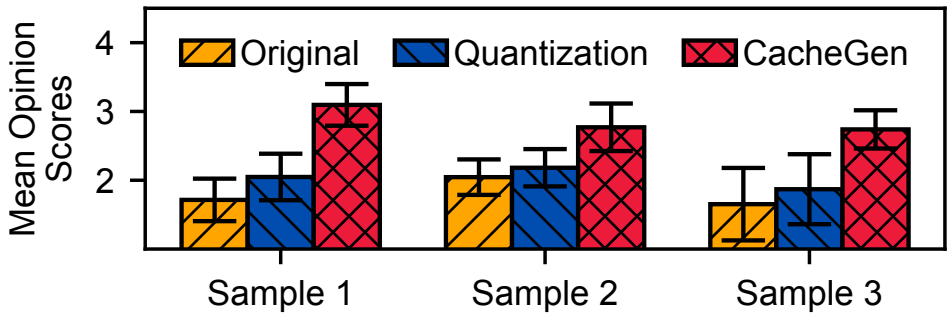


Figure 3.15: *Real user study shows CacheGen improves QoE significantly over other baselines.*

upon the uniform quantization. This indicates that removing the constraint of maintaining the tensor format of KV caches, and encoding them into bitstreams with our change-based encoding and AC can further reduce the size of KV cache after quantization.

Quality of Experience: We performed an IRB-approved user study to validate the effectiveness of CacheGen. We selected three conversation history from the LongChat dataset used in previous evaluations. For each user, we first present the conversation history with ChatGPT. Then we show the same response but produced by different pipelines by adding different TTFT and let users rate the quality of response. With 270 ratings collected from Amazon MTurk mtu, we show that CacheGen consistently outperforms other pipelines in QoE with shorter TTFT in Figure 3.15.

Evaluation results of CacheGen with more baselines is available §A.2, including using a smaller-sized model to speed up TTFT and Gisting, another context-shrinking technique.

3.8 Discussion and future work

CacheGen's Compatibility with KV Cache Compression Techniques: Emerging techniques like smart quantization Kang et al. [2024], Liu et al. [2024f], Hooper et al. [2024] can be integrated with CacheGen. After quantization, CacheGen can still apply delta encoding and arithmetic coding.

Incremental KV Cache Streaming: Future work includes extending CacheGen to stream KV caches incrementally, akin to Scalable Video Coding (SVC) Hellwagner et al. [2011], by initially sending low-quality KV caches and then incrementally improving quality by sending differences.

Context Reuse in Real-World LLM Systems: In §3.2, we explain why contexts are likely reused across requests but lack real-world datasets to illustrate this. Future work includes finding or creating such datasets.

Evaluating on Higher-Power GPUs: In §3.7, we use NVIDIA A40 GPUs to conduct the experiments. We acknowledge that with very high-power GPUs and relatively low bandwidth, CacheGen might not significantly improve over the text context baseline. Evaluating CacheGen on more powerful GPUs is left for future work.

Other System Designs: §3.5 covers CacheGen's encoder and streamer design. Other aspects such as which storage device(s) to store KV cache, caching policies, and locating KV cache quickly are discussed in concurrent works Yao et al. [2024a], Gao et al. [2024a], Jin et al. [2024a]. We leave combining CacheGen with these works for future work.

CHAPTER 4

DROIDSPEAK: KV CACHE SHARING ACROSS MODEL VARIANTS

4.1 Introduction

Nowadays, LLM inference has become one of the most resource-consuming workloads in industry, demanding ever larger clusters of GPU machines vLLM Production Stack Team [2025], ai-dynamo [2025], vLLM Project [2025], Wu et al. [2023b], Patel et al. [2024], Zhong et al. [2024b]. To reduce the computation demand, a common optimization is for GPU machines that run the *same* LLM to share KV caches of reused input prefixes over the network Chen et al. [2024e], Hu et al. [2024a], Li et al. [2024a], Liu et al. [2024d]. However, how to make that optimization work across *different* LLMs is yet to be studied.

Indeed, an emerging trend is hosting multiple *different* LLMs in one GPU cluster. The reason is that with LLMs used in more complex or personalized tasks, multiple LLMs, often fine-tuned from the same foundational model, are needed to serve different users, tasks, or roles. These LLMs work together to perform one complex task or to offer different customized services Fang et al. [2024], Mineiro [2024], Borzunov et al. [2023], Yang et al. [2024], Arslan et al. [2024]. Since standard prefix-caching techniques work only when the KV cache is reused by the same LLM, the use of multiple LLMs poses a direct challenge — how to reuse KV cache efficiently across different LLMs in distributed settings.

To motivate this challenge, we highlight three common use cases of multiple fine-tuned models working together in a system. The first one is *multi-agent systems*, which use different fine-tuned models to serve as the agents and to accomplish collaborative tasks Chen et al. [2023a], Hong et al. [2024b]. The second use case is serving *multi-LoRA or multiple fine-tuned models*, such as models that are continuously updated over time with newer data Huang et al. [2025], or concurrently serving LoRA adapters Chen et al. [2024d], Xia et al.

[2024], Sheng et al. [2024a]. The third use case is *personalized assistant systems*, where a model is fine-tuned for each user (or user type) according to their personal preferences in coding or writing.

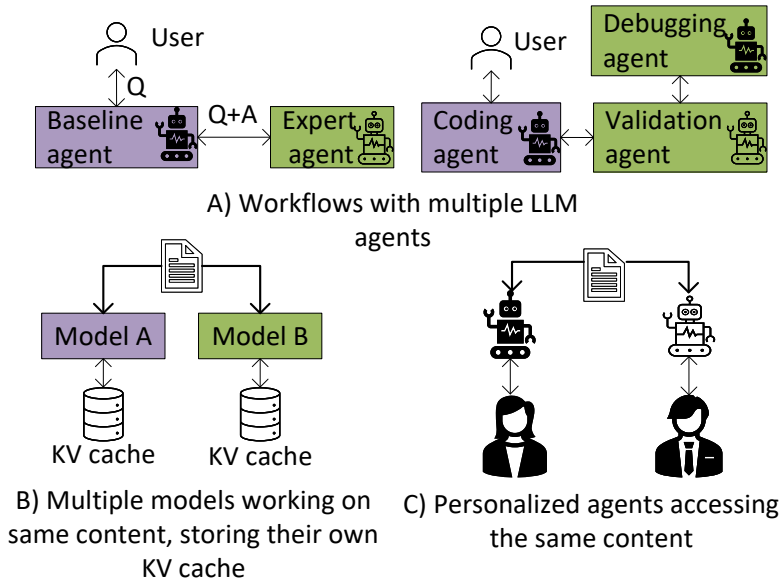


Figure 4.1: *Various scenarios in which same context is shared by multiple LLMs. DroidSpeak brings down the computation latency by up to 3.1×, increases throughput by 4×.*

In all of these scenarios, prefix sharing is common. For example, in multi-agent systems (Figure 4.1A), when the coding agent talks to the validation agent, the conversation history of the coding agent will be prepended to the input of the validation agent, to ensure the coherence of the conversation. In a multi-LLM or multi-LoRA inference system (Figure 4.1B), an updated model in a chatbot application could refer to the same piece of conversation history produced by an older version of the model. In a personalized assistant system (Figure 4.1C), where each assistant is fine-tuned for a different user's preference, the same news (*i.e.,* the query prefix) can be used to answer different users' queries.

Based on the observations above, we pose the question: *can we share the intermediate states (i.e., KV cache) produced by one LLM on a given context to accelerate the prefill for another LLM?* In this paper, we seek to answer this question under the assumption that the two LLMs have different weights but the same architecture.

Intuitively, the potential performance benefit of such intermediate state sharing is huge — previous work on single-LLM KV cache sharing has shown up to 8× latency and throughput improvement by speeding up the expensive prefill phase of LLM inference, particularly for long context workloads Gao et al. [2024b], Gim et al. [2024], Liu et al. [2024d], Jin et al. [2025a]. However, just like a video decoder could not decode a video encoded with another codec, naively sharing the KV cache across different LLMs would cause generation quality to drop greatly.

Our hypothesis is that there should be a way to *re-compute a small portion and reuse a large portion*, although not all, of the KV cache between two models that are *fine-tuned from the same base model* — since the models are only *fine*-tuned, they should share similar understanding of the same input context and hence help accelerate each other's inference. Of course, the challenge is to validate this hypothesis, and to figure out which part to re-compute or re-use without incurring much delay overhead or degradation of quality.

Through a thorough empirical study (§4.3.2), we measured eight representative model pairs and found that only a small subset, often around 10%, of layers are sensitive to the KV cache difference between two models in a pair. The identities of these layers vary with different model pairs, but are largely consistent across different inputs for the same model pair. We refer to these layers as *critical layers* in this paper. Therefore, for each pair of LLMs, we propose to selectively recompute these critical layers in the KV cache.

We build our insights into *DroidSpeak*, the first distributed multiple-LLM inference system that enables efficient sharing of KV caches across different LLMs. First, DroidSpeak identifies the critical layer groups through offline profiling on a held-out training set, ensuring sufficient re-computation for accuracy purposes while reusing as many layers' KV cache as possible. Second, DroidSpeak implements smart KV cache loading, which pipelines the loading of KV cache with the re-computation of critical layers, to hide the loading delay from remote nodes as much as possible.

We should note that the high-level idea of selectively recomputing KV cache is not exactly new. DroidSpeak differs with prior work Gao et al. [2024b], Gim et al. [2024], Liu et al. [2024d], Yao et al. [2025] in *why* and *how* to re-compute KV cache: DroidSpeak is designed for KV cache reuse across *different* LLMs, while prior work assumes a *single* LLM; due to the different purposes, none of the prior work chooses to re-compute or re-use a group of layers like in DroidSpeak. For example, CacheBlend Yao et al. [2025] updates a reused KV cache, but it still feeds the KV cache to the *same* LLM that generated the KV cache. Moreover, it updates the KV cache of certain tokens, rather than certain layers like in DroidSpeak.

We evaluate DroidSpeak on six datasets across *three* different tasks, including *question answering, text summarization, and coding*, across eight different model pairs. We compare DroidSpeak with various baselines, including direct KV reuse Gao et al. [2024b], CacheBlend Yao et al. [2025], and smaller models. Across these setups, we can reduce the latency by up to 3.1×, improve throughput by up to 4× with negligible drop in quality (measured in F1 score, Rouge-L, or code similarity score).

While the concept of "translating" KV caches between models involves a machine-learning challenge, our primary contribution lies in making it practical in a *distributed system* setting, which requires the transfer and computation of such KV-cache translation to be done efficiently. Specifically, we identify a key system-level insight: once the E cache of the transition layer has been transferred, selective re-computation can start independently of the KV cache transfer for reused layers. This decoupling allows transfer and re-computation to be pipelined efficiently.

4.2 Background & Motivation

Throughout this chapter, we follow the terminology of Section 2.1 (*KV cache* and *E cache*); the quality of these E/Q/K/V tensors directly affects the model's ability to understand and process the input context effectively. Furthermore, in this section, we give a more detailed

introduction to the background of the emerging workload of context sharing between different fine-tuned model versions and the motivation for DroidSpeak.

4.2.1 Fine-Tuned LLMs

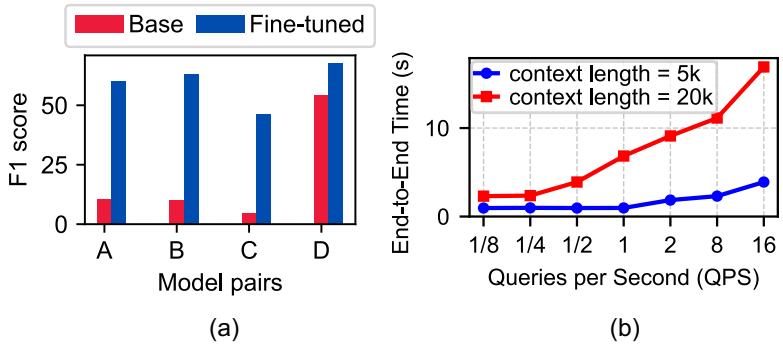


Figure 4.2: *(a) Fine-tuned model produces higher quality than baseline. (b) Shorter input leads to smaller end-to-end time.*

Despite being versatile, foundational LLMs' capabilities on specific tasks can improve through fine-tuning on specialized domain data. For example, one can turn a foundational LLM into a customer-support agent by finetuning on troubleshooting requests Predibase [2023], or into a legal assistant by finetuning on case law and statutes Yue et al. [2023]. Fine-tuning can greatly improve the accuracy of an LLM on a target domain, as shown in Figure 4.2(a), where the fine-tuned models (`Llama-3-70B-Instruct` AI@Meta [2024], `Mistrallite` Song et al. [2023a], `Llama-3-8B-Instruct` AI@Meta [2024], and `MAmmoTH2` Yue et al. [2024]) greatly outperform the foundational models they originate from, on the HotpotQA dataset Yang et al. [2018].

Recent works in parameter efficient fine-tuning, like LoRA Hu et al. [2022] have made fine-tuned models even more accessible by updating part of the model weights, reducing computational and memory resources during fine-tuning.

4.2.2 Context Sharing Across LLMs

As Section 2.1 describes, prefix contexts[1] are increasingly shared across *different* LLMs in compound AI systems: agents in agentic workflows share conversation histories to stay coherent and consistent, personalized assistants answer queries over shared knowledge bases, and updated model versions or concurrently served LoRA adapters re-process the same popular contexts. In all these use cases, the different LLMs are typically fine-tuned versions of a common foundational model or of one another: a single LLM often lacks the diversity in learned data distribution or the specialized capabilities that complex tasks demand, so multiple fine-tuned LLMs collaborate to solve them Feng et al. [2025], Subramaniam et al. [2025], Ye et al. [2025], Li et al. [2025]—in CAMEL, a very popular multi-agent framework, different agents can be models fine-tuned on independent data domains Hu et al. [2025a]. Indeed, compared with using different prompts on the same model for different agents, using fine-tuned models as different agents can better improve output quality Chen et al. [2023a], Song et al. [2024].

In this chapter, for convenience, we use the following terminology:

- ***Sender*** model produces the KV cache of a context;
- ***Receiver*** model reuses the KV cache (with limited recomputation) of the reused context.

For example, in the coding agentic workflow of Section 2.1, the testing agent (receiver model) reuses the context produced by the coding agent (sender model) Hong et al. [2024b], Qian et al. [2024], Wu et al. [2024f].

These emerging workloads fuel the need for efficient context sharing across fine-tuned LLMs and motivate DroidSpeak.

1. Since a shared context is often the prefix of different inputs, we use *prefix* (caching/sharing) and *context* (caching/sharing) interchangeably.

4.2.3 Distributed LLM Inference Systems

As the demand for LLM inference continues to grow, it has become common to serve LLMs in a cluster of GPU nodes. Many companies have developed their own distributed inference systems, such as vLLM Production Stack vLLM Production Stack Team [2025], NVIDIA Dynamo ai-dynamo [2025], and ByteDance AIBrix vLLM Project [2025]. Among all these frameworks, KV cache sharing across nodes is one of the most important features for reducing prefill computation and increasing overall throughput. Specifically, when there are multiple requests to the *same model* querying a common prefix context, these systems can transfer KV cache generated by another user request, either from another GPU node or through a centralized storage backend Jin et al. [2025b], Gao et al. [2024b], Chen et al. [2024e].

However, current distributed inference systems have not optimized for multiple-LLM inference yet–they do not explore the opportunity to share KV cache across GPU nodes when the requests are querying *different models*.

4.2.4 Prefill interference

Since TTFT grows super-linearly with input length and decoding cannot start until the prefill phase completes (Section 2.1), long inputs often turn prefill delays into the end-to-end bottleneck, greatly reducing an inference system's goodput. For example, as shown in Figure 4.2(b)—where Llama-3-8B runs on a single A100 GPU with synthetic input lengths of 5K and 20K tokens—setting a 3-second latency SLO reveals that increasing the input size by only 4× can reduce goodput by as much as 32×.

4.3 Reusing KV cache across LLMs

A simple way to eliminate the overhead of repetitively re-computing the KV cache that another LLM has generated before is to reuse the KV cache produced by another model.

As a concrete example, on an A100 GPU and using `Llama-3.1-8B-Instruct`, reusing the KV cache for a 40K-token input can reduce prefill latency from 4s to 0.08 seconds. This naturally leads to the key question: What effect does directly reusing another LLM's KV cache have on generation quality?

In this section, we present the first empirical study of how reusing another model's KV cache impacts output quality, and we further examine whether these effects vary across individual layers of the cache.

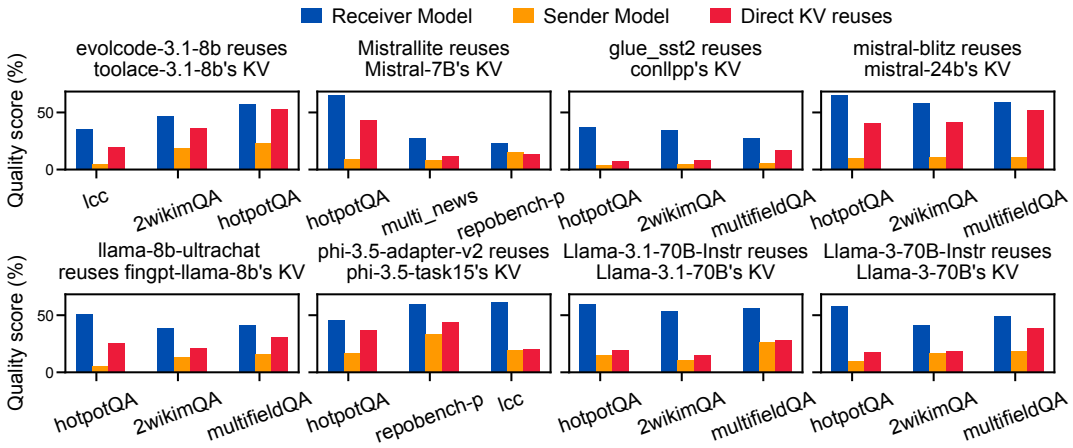


Figure 4.3: *Directly reusing the full KV cache greatly degrades generation quality.*

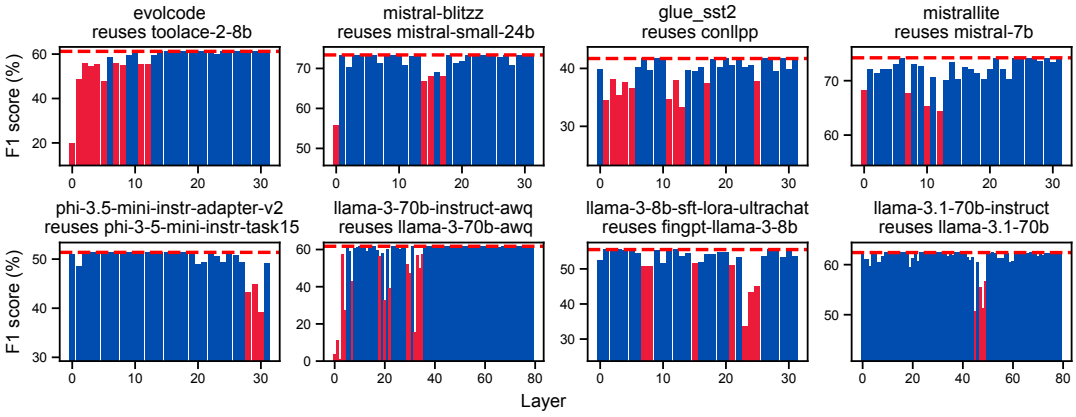


Figure 4.4: *Different layers have different sensitivities to deviation in KV cache. Plotted by reusing only one layer's KV cache from the sender model on the receiver model. The red dashed line is the original accuracy of the receiver model. The bars colored red are those that have an F1 score drop of over 10% compared to the original receiver model.*

4.3.1 Building the benchmarks and datasets

Before getting into the KV cache sharing and patterns, we describe the benchmark set we build for DroidSpeak. The study needs pairs of models that share the context provided by the datasets. The following assumptions are also made when building the benchmark.

- The pair of models should share the same foundational model. Specifically, the pair can either consist of the foundational model and a fine-tuned model based on it, or, two fine-tuned models based on the same foundational model.
- The dataset selected should be related to the task for which one of the models has been fine-tuned. This is important since in any context-sharing scenario, the receiver model is performing the specialized task.
- The receiver model fine-tuned on the task in the corresponding dataset should yield better quality than the sender model in the pair.

Using these assumptions, we formulate the benchmark as shown in Appendix Table B.1. We use 8 pairs of models across 6 datasets (including HotpotQA Yang et al. [2018], multi-fieldQA_en Zhao et al. [2024], 2wikimQA Ho et al. [2020], multi_news Fabbri et al. [2019], lcc Guo et al. [2023], and repobench-p Liu et al. [2024c]). The receiver models are fine-tuned on datasets spanning chat, coding, instruction tuning, and long-context tasks. The quality metric used is taken directly from the dataset.

We focus on the use case where the sender model generates the intermediate states for the context and the receiver model reuses its intermediate states. This is a challenging use case because the sender model has worse accuracy than the receiver model, so achieving high quality requires properly refreshing the KV cache.

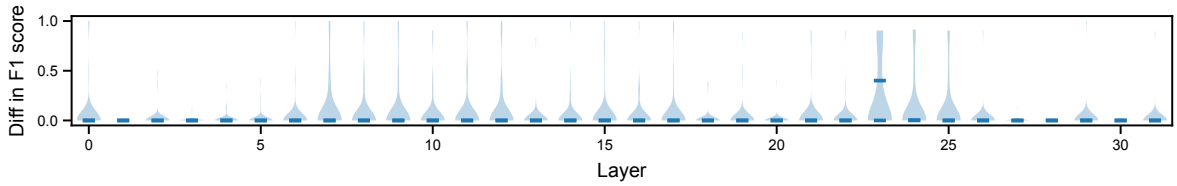


Figure 4.5: *Variation in F1 score per input within a single dataset (HotpotQA) for model pair Llama-3-8B-sft-lora-ultrachat reusing fingpt-llama-3-8B. We plot the 25 and 75 percentiles. Except layer 23, the 25 and 75 percentiles overlap, indicating a low variance of error sensitivity across all layers except 23.*

4.3.2 Empirical insights of KV cache

Naive reusing is suboptimal: The first observation is about naively reusing the sender model's KV cache on the receiver model. Specifically, we observe that:

Insight 4. *Reusing the whole KV cache between models leads to a huge loss in accuracy.*

A naïve way to reuse the intermediate state between models is to reuse the KV cache *as is*. In this case, the receiver model receives the KV cache for the whole input prompt from the sender model. It then uses this to generate the output tokens in the decode phase, thereby skipping the prefill phase.

We show the impact of this on quality in Figure 4.3. For each pair of models and dataset, we show the F1 score (higher is better) of *a)* the receiver model, *b)* the receiver model while reusing the KV cache generated by the sender model, and *c)* the sender model alone.

Although the quality of the receiver model with the sender model's KV cache is still better than the sender model alone, we lose a lot of accuracy. HotpotQA tends to lose more than 50% of the accuracy points across most of the model pairs, while the other datasets show varying amounts of changes across model pairs.

Layer-wise sensitivity to KV cache reuse: Our second observation is about whether KV cache reusing leads to the same impact across all layers.

Insight 5. *Only a small subset of layers are sensitive to KV cache reuse in terms of accuracy.*

Figure 4.4 shows the quality drop by reusing part of KV cache from the sender model. Specifically, each bar represents the quality achieved by the receiver model reusing the KV cache for that corresponding layer from the sender model, with everything else being recomputed.

For most of the model pairs, we find only a small subset of layers are sensitive to the deviation in KV cache (*i.e.,* F1 score drops significantly), and we refer to these layers as *critical layers*, which are colored red. On average across all pairs of models, we identify 11% of layers to be critical.

Similarity of sensitivity across different inputs: Our third observation is about whether different inputs show similar patterns in layer-wise sensitivity.

Insight 6. *The variation in KV cache patterns across inputs is only notable for critical layers.*

Figure 4.5 shows the violin plot of the normalized change in F1 score per input in hotpotQA dataset, when `llama-3-8b-sft-lora-ultrachat` reusing `fingpt-llama-3-8b`'s KV cache of each layer only. Layer 23, which is also marked as the most critical for this model pair in Figure 4.4 (*i.e.,* the largest F1 score change), shows a wider variation across different data points from the dataset, with a lot of them observing F1 score change $> 50\%$. However, for all the non-critical layers, the variance in the F1 score change is insignificant, meaning that such non-critical layers do not change across various inputs.

This phenomenon is also observed across other model pairs. Intuitively, this can be because critical layers are essential for the reasoning capabilities Chen et al. [2020b] or the ability to accomplish specific downstream tasks Chen et al. [2023c]. These reasoning capabilities must remain accurate to interpret any input to the LLMs.

4.4 DroidSpeak Design

Building on the insights in the previous section, we designed DroidSpeak to enhance the context sharing between two LLMs. The central questions that DroidSpeak targets are the following: *how do we determine the layers to re-compute to reduce latency, while keeping the quality loss minimal?*

4.4.1 Challenges with Selective KV Cache Reuse

Insight 5 suggests selectively reusing the KV cache while recomputing it for critical layers *might* preserve quality. However, selecting all critical layers scattered across different parts of the LLM is suboptimal for both efficiency and quality.

The efficiency challenge: Recomputing critical layers that are non-contiguously placed is inefficient.

During the prefill phase, the output of a layer where the KV cache is reused only contains information about the first generated token. In contrast, recomputing the KV cache needs to start from the E cache of this layer for the whole context. While it is possible to obtain the E cache for layer l by performing a full prefill from the context starting from the first layer till layer l, this approach completely defeats the purpose of KV cache reuse.

To address this, we use the E cache from the sender model to start the recomputing at the layer when transitioning from KV cache reuse to recompute. We refer to this layer as a ***transition layer***. As depicted in Figure 4.6, for any transition layer (between reuse and recompute), the sender model must store and transmit the E cache to the receiver model.

The E cache is typically large, reaching up to *twice* the size of the KV cache for the Mistral-7B or Llama-3-8B model families, and up to *four* times larger for Llama-3.1-70B since the KV cache size is optimized by group-query attention Ainslie et al. [2023]. Thus, the overhead of storing the E cache in GPU memory and the delays caused by loading it from remote GPU nodes can be substantial, far exceeding the cost of storing and loading

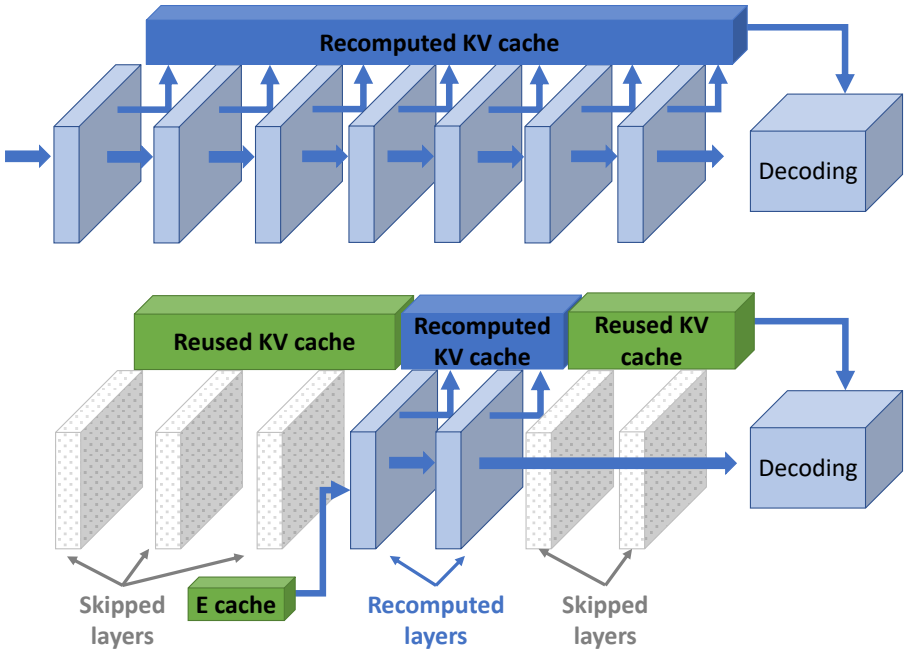


Figure 4.6: *DroidSpeak chooses the critical layer groups (layers 4-5) to re-compute, and reuse KV cache for other layers.*

KV cache alone.

The accuracy challenge: Furthermore, reusing the sender model's E cache at the transition layer also hurts the accuracy of the final output. This is because the E cache loaded from the sender model (starting point of the recomputation) already differs from the receiver model. Such difference eventually will introduce deviation from the point of recomputation and propagate over all later layers. It is crucial to minimize the error caused by such deviation.

If we pick all the critical layers, which are often not appearing in contiguous groups (Figure 4.4), there will be multiple transition layers from reuse to recompute, introducing multiple deviations in E cache.

Figure 4.7(a) illustrates this. If we choose to recompute *only* critical layers (*i.e.,* layers 16–18, 20, and 25–27), we need to load E cache at layers 16, 20, and 25. However, whenever we load E cache, the error from E cache will be propagated to subsequent critical layers (*e.g.,* loading E cache at layer 16 populates errors to 16–18) and eventually to the output. Thus,

even if all the critical layers are recomputed, this will lead to a substantial output error. In contrast, recomputing a contiguous group of layers from 16 to 27 avoids this problem by recomputing the KV cache of non-critical layers that are located between critical layers. As shown in Figure 4.7(a), re-computing a contiguous group of layers has much lower output error than only re-computing the critical layers.

4.4.2 Profiling for re-computation configuration

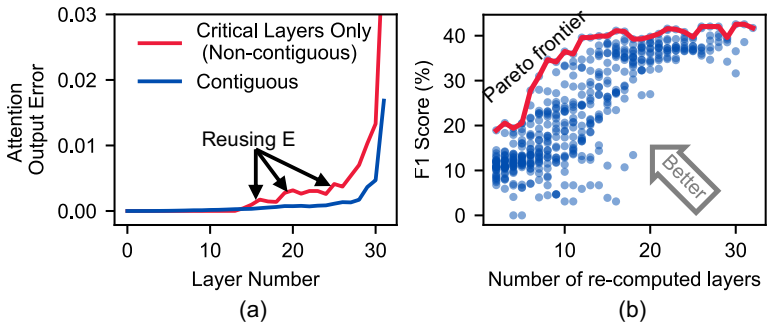


Figure 4.7: *(a) More transition points lead to higher output error. (b) An example result obtained by offline profiling. Each point represents the F1 score achieved when re-computing a specific group of contiguous layers. The Pareto frontier shows the maximum F1 score attainable for each number of re-computed layers.*

A key question still remains: how to determine the critical layer groups to be re-computed?

Based on Insight 6, the critical layers that are sensitive to KV cache differences vary little with the inputs. This motivates our design to choose the critical layer group based on some example inputs, and then apply to other new inputs. Specifically, for each model pair, we use a training dataset to determine the critical layer group. We refer to the critical layer group as the *re-computation configuration*. The goal is to understand the relationship between the critical layer groups to perform re-computation and its impact on generation quality.

Figure 4.7(b) shows an example profiling result for the `glue_sst2` and `conllpp` model pair. Each point in the scatter plot represents the F1 score achieved for a specific number

of re-computed layers. From these points, we obtain the Pareto frontier, which captures the highest F1 score achieved at a specific group of layers. For instance, a configuration with 11 re-computed layers is a good example on the Pareto frontier, since the F1 score drop is within 5% of the original accuracy of the receiver model, while the number of re-computed layers is minimized. Here, 5% is a hyperparameter that users can freely adjust according to their requirements.

Note that although the exact set of layers that need to be re-computed may differ between the training dataset and testing dataset, the key is that the Pareto frontier is similar across different datasets (§4.5.4).

Training data selection: We choose the training dataset to be similar to the domain of the task that the testing datasets target. In practice, we can use the fine-tuning dataset for profiling, as the inference requests of a specialized model should align with the data domain used to fine-tune it.

Profiling Overhead: The profiling overhead has a complexity of $O(l^2)$ where l is the number of layers in the LLMs. For example, for `Llama-3-8B` with 32 layers, it takes three hours on an A100 GPU. This one-time cost is negligible since these models are deployed at a very large scale Patel et al. [2024], Zhong et al. [2024b]. Furthermore, this cost can be substantially reduced by grouping layers when profiling. Profiling at the granularity of 2-layer groups, for instance, reduces the profiling time by about 3x. In the evaluation we show soon in §4.5, we use the 2-layer group profiling granularity.

4.4.3 DroidSpeak runtime design

As shown in Figure 4.8, DroidSpeak consists of two stages. The offline stage (as detailed in §4.4.2) uses a training dataset to profile the relationship between the value change of each layer on the generation quality. This step allows DroidSpeak to dynamically choose the right re-computation configuration based on available resources. Next, we detail the *online*

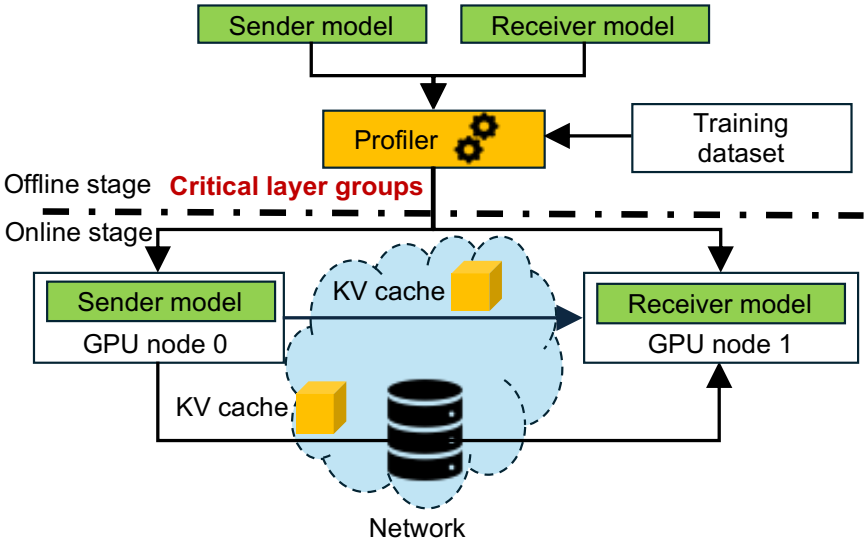


Figure 4.8: *Overall system design of DroidSpeak, including the offline stage (top) that profiles the critical layers of each pair of sender and receiver models (§4.4.2); and the online stage (bottom) that uses the profile to dynamically pick the critical layers and KV-cache loading strategy for each job (§4.4.3).*

stage that uses the offline profile to dynamically decide the critical layers of the KV cache and executes the partial recomputation on these critical layers to preserve high generation quality.

Specifically, in the online stage, DroidSpeak dynamically decides which point on the Pareto frontier should be used based on the latency SLO (details in Appendix§B.3). We assume the E cache and KV cache at each transition point are precomputed and stored, but they can also be generated by the sender model in real-time and sent directly from the sender model. Then, the receiver model selectively recomputes critical layer groups while reusing others, achieving a balance between computational efficiency and quality.

Although storing the E cache introduces extra GPU memory overhead on the sender model's side, only the E cache for *a single layer* – the transition layer – needs to be kept. This overhead is negligible compared to storing the KV cache across all layers by default. For example, in `Llama-3.1-8B-Instruct`, storing the KV cache for all layers for 10K tokens requires 1.2 GB of GPU memory, whereas storing the E cache for just one layer only needs

0.08 GB – about 6% of the total KV cache memory usage.

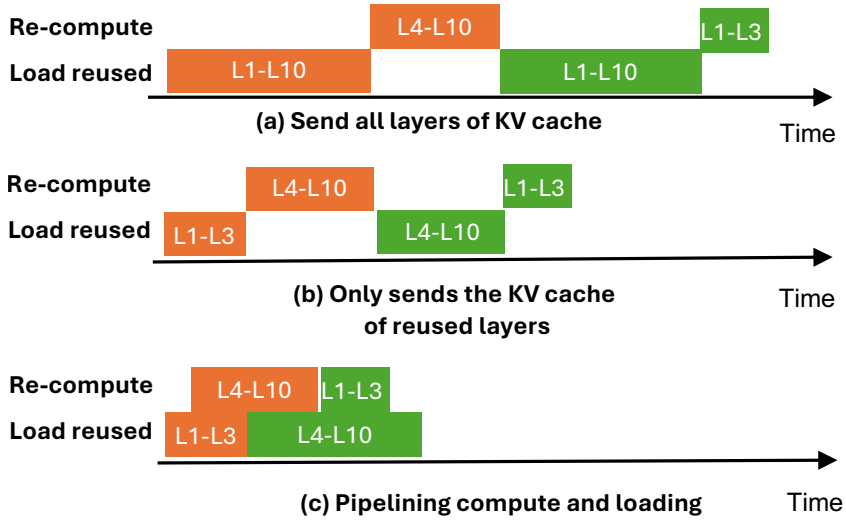


Figure 4.9: *Illustration of different KV cache transferring strategies. Each color represents the work (KV cache recomputation and loading reused KV caches) for the same query for one model.*

Smart KV Cache Loading: Since GPUs hosting different LLMs may reside in separate nodes, DroidSpeak needs to fetch KV cache from remote nodes frequently. Without careful design, this transfer can significantly increase the end-to-end latency, particularly when fewer layers are recomputed (*i.e.,* more KV cache layers need to be transferred).

Consider an example with two receiver models A and B. Each model has ten layers. Model A recomputes layers 4–10 and reuses KV cache for layers 1–3, while model B recomputes layers 1–3 and reuses KV cache for layers 4–10.

Figure 4.9 illustrates the timeline of three different loading and recomputation strategies. We assume that a request arrives at A at time=0, followed by a request to B after 2 time units, and that recomputing or transferring one layer (KV or E cache) takes 1 unit of time. The most naive approach is to load *all layers* of the KV caches before recomputation begins in each job. For example, in model A (orange), this means first loading the E cache for layer 4 and the entire KV cache, followed by 7 units of re-computation time, similarly for B, resulting in a total TTFT of 47 (Figure 4.9(a)).

A slight improvement is to load only the reusable layers' KV cache for A and B, right before recomputation. This reduces the total TTFT to 30, as shown in Figure 4.9(b).

However, both approaches miss the opportunity to overlap recomputation and the loading of reused KV caches. Recomputation can start immediately after receiving the E cache of the transition layer. The optimal solution, shown in Figure 4.9(c), pipelines loading and recomputation. For model A, the E cache for layer 4 is transmitted first, enabling recomputation of layers 4–10 to start while the KV cache for layers 1–3 transfers in parallel. Similarly, for B, the loading of E/KV cache can begin before A finishes recomputation. This pipelined strategy reduces the total TTFT to 17—approximately a 2× improvement over the baseline in (b).

4.4.4 Implementation

We implement DroidSpeak with about 3K lines of code in Python, based on PyTorch v2.0, CUDA 12.0, and LMCache 0.1.4 Authors [2024], Cheng et al. [2024]. DroidSpeak operates the LLM inference serving engines through these interfaces:

- `store(Cache, context, LLM)`: We split the KV or E cache into layers, and store it in a key-value store in GPU memory.
- `fetch(context, LLM, layer_id) -> Cache`: Depending on what was stored previous `store()` call, this loads layer's KV or E cache of the corresponding `LLM`.
- `partial_prefill(recompute_config, context)-> text`: it takes in the recomputation configuration and the context, including which layers to recompute during prefill, and then generates the output text.

We implement these three interfaces in vLLM Kwon et al. [2023a] and LMCache Authors [2024]. For `store_kv`, after an LLM generates the KV cache for a piece of context, we calculate the hash of the context text, and put it into the key-value store if the context

does not exist in the current store. Before we run the inference for any LLM, we obtain the re-computation configuration from the offline profiling (§4.4.2), which includes the layer numbers for recompute and KV cache reuse. During the online inference stage, we call the `partial_prefill` function, which calls `fetch_kv` for the layers for KV cache reusing, and `fetch_e` at the transition layers. Both `fetch_kv` and `fetch_e` are implemented with `torch.distributed` PyTorch Contributors [2024] to fetch KV or E cache from a remote GPU node. All transmission will be placed on a CUDA Stream different from PyTorch's default computation stream PyTorch Team [2024], enabling us to overlap transmission of KV cache with recomputation and hiding the transmission delay.

4.5 Evaluation

The key takeaways from the evaluation are:

- Across three datasets and eight model pairs, DroidSpeak can reduce the prefill latency by 1.7–3.1× without compromising accuracy, with an average prefill speedup of 2.1×.
- In the online serving system, DroidSpeak achieves up to 4× improvement in throughput.
- DroidSpeak's profiling of recomputing layers is robust across different datasets and model types.

We also show more results in Appendix §B.2 including comparison of DroidSpeak with smaller models, and applying DroidSpeak on math tasks.

4.5.1 Experiment Setup

Models: We evaluate DroidSpeak on eight pairs of models (Table B.1) of different sizes, specifically the fine-tuned versions of Mistral-7B, Mistral-24B, Llama-3.1-8B, Llama-3-8B,

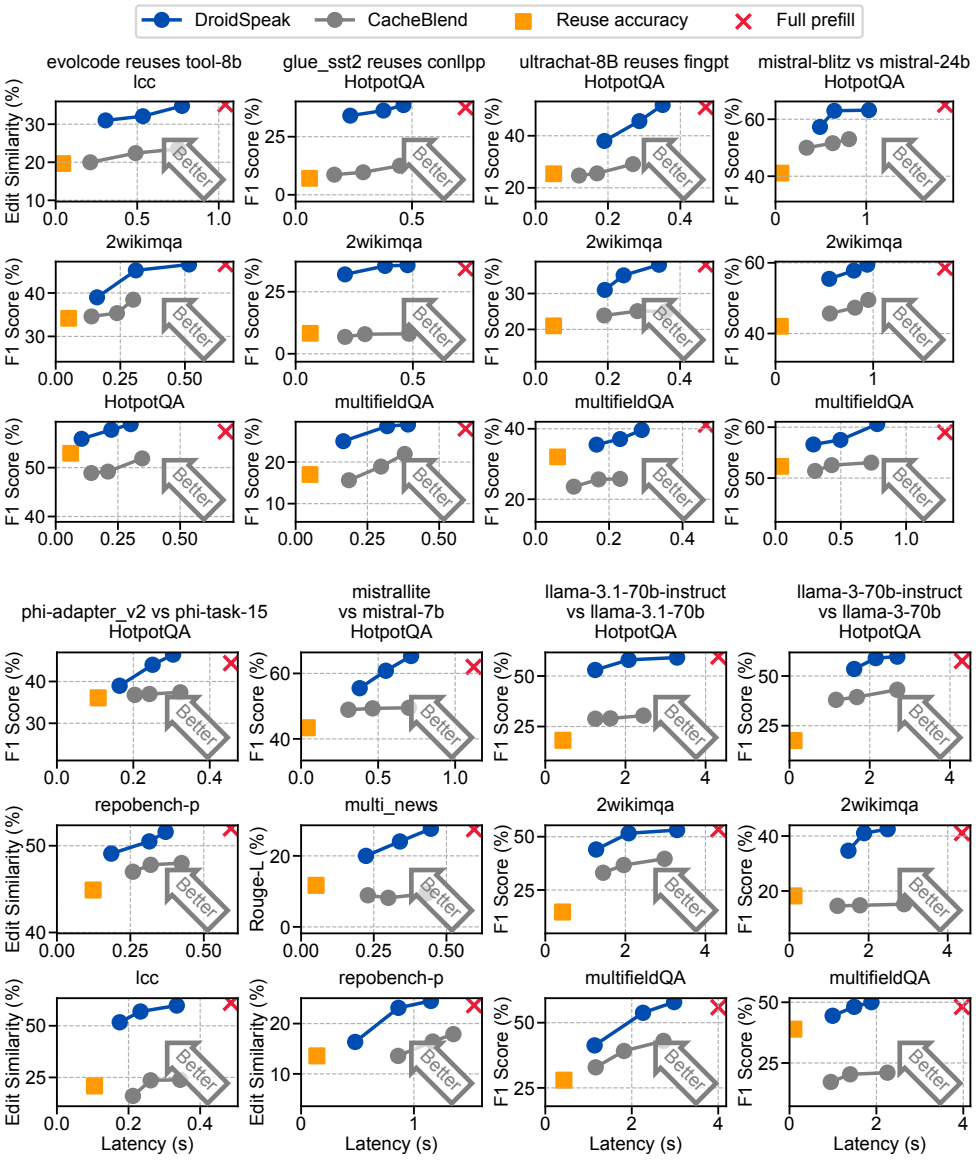


Figure 4.10: *Prefill delay and F1 score trade-off. DroidSpeak greatly reduces prefill latency while maintaining generation quality.*

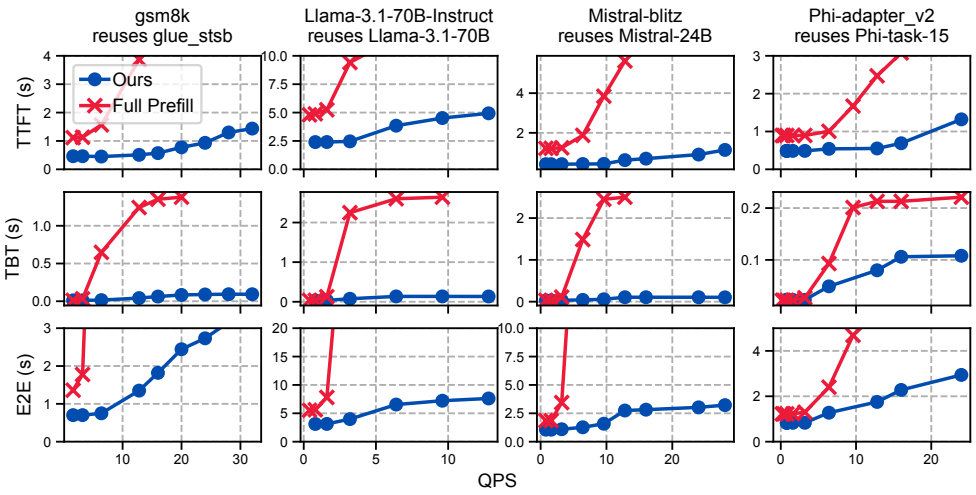


Figure 4.11: *The impact of arrival rate on TTFT, TBT, and E2E, when the DroidSpeak's quality is same as full prefill.*

Phi-3.5-mini-instruct, Llama-3-70B and Llama-3.1-70B, selected with the criteria in §4.3.1. These models are fine-tuned on the base foundation model for chat-enhancing tasks, coding tasks, and long context reasoning *et al.*. For Llama-3.1-70B and Llama-3-70B models, we use 4-bit quantized models with AWQ Lin et al. [2024c] to fit on one A100 GPU.

Note that the receiver models are not all directly fine-tuned from the sender model; they also include *two fine-tuned variants* derived from the same foundation model.

Hardware setting: We run the experiments on two A100 virtual machines connected with 200 Gbps NVIDIA Mellanox HDR InfiniBand links, namely `Standard_ND96amsr_A100_v4`, which contains 8 × 80GB A100 GPUs on each virtual machine.

Datasets: We evaluate DroidSpeak on six datasets, spanning three different tasks including *multi-hop question-answering, summarization, and code completion*, with their context length statistics summarized in Table B.2. For each model pair, we report results on three of these datasets, following the criteria that the receiver model must achieve higher generation quality than the sender model on the chosen datasets.

Train/test split: As discussed in §4.4.2, DroidSpeak profiles the critical layer groups that has quality drop within 5% of the original quality with a training dataset offline. Specifically,

we use 50 contexts from HotpotQA dataset as the training dataset to obtain the critical layer groups with the profiling mechanism mentioned in §4.4.2. We apply the critical layer groups on all the other testing datasets in the benchmark.

Quality metrics: We measure generation quality using the standard metric of each dataset, following prior work Bai et al. [2024], Liu et al. [2024d], Yao et al. [2025]. Specifically, we use F1 score for QA tasks (`hotpotQA, 2wikimQA, multifieldQA_en`), which measures the probability that the generated answer matches the ground-truth answer for the question-answering task; Rouge-L score for summarization tasks (`multi_news`), which measures the longest common subsequence between the generated summarization and the ground-truth answer; and finally the code similarity score for code completion tasks (`lcc, repobench-p`), which measures the edit distance between the generated completed code and the ground-truth code.

System metrics: We evaluate DroidSpeak against the baselines using time-to-first-token (TTFT, defined in Section 2.1), time-between-tokens (TBT), *i.e.,* the average time between two consecutive generated tokens, and end-to-end latency (E2E), *i.e.,* the duration from a query's submission to its last generated token. In §4.5.2 we also measure prefill latency, which includes the prefill computation time on GPU and the loading delay to fetch KV and E cache through InfiniBand bandwidth link across two GPU nodes.

Baselines: We compare with the following baselines:

- Full prefill: the receiver model prefills the text of the context with vLLM Kwon et al. [2023a], representing the baseline of the highest computation overhead but the best quality we can get.
- Full KV cache reuse Gao et al. [2024b]: the receiver model directly reuses the KV cache from the sender model, and the receiver model runs decoding with the transferred KV cache.
- CacheBlend Yao et al. [2025]: we extend CacheBlend's algorithm to determine the

important tokens to re-compute for cross-LLM KV cache sharing, based on the difference between the re-computed KV cache and sender model's original KV cache for the first layer.

- Smaller models: In §B.2.2, we also compare `Llama-3.1-70B-Instruct`'s accuracy and latency trade-off with DroidSpeak with `Llama-3.1-8B-Instruct`, which is fine-tuned with the same instruct-tuning dataset.

4.5.2 Lower Latency with Preserved Accuracy

We first demonstrate DroidSpeak's reduction in prefill delay and accuracy trade-off in Figure 4.10. Across 8 pairs of models on three datasets, DroidSpeak achieves 1.7–3.1× reduction in prefill delay over the full prefill method, without compromising generation quality. On the other hand, when compared with reusing all of sender model's KV cache, DroidSpeak successfully preserves the improved quality of the receiver model despite a slightly higher delay. Compared to CacheBlend, DroidSpeak can achieve much better latency and quality trade-off. Specifically, DroidSpeak has 5–33% (average 16%) higher quality than CacheBlend at a similar prefill latency.

Understanding DroidSpeak's improvement: DroidSpeak outperforms the baselines for various reasons. Compared to the full prefill baseline, DroidSpeak achieves significantly lower prefill delay as only a small fraction of layers are prefilled. In contrast to full KV reuse, DroidSpeak has a longer prefill latency because it does not perform prefill at all. However, it greatly reduces accuracy because it misses the opportunity to leverage layer-wise sensitivity in the KV cache difference. DroidSpeak is better than CacheBlend in quality because DroidSpeak re-computes the critical layer groups that are most sensitive to KV cache deviation between models, while CacheBlend fails to spot the critical layers since it selects the tokens to re-compute based on the first layer.

4.5.3 Inference Throughput and Latency Improvement

To see the impact of DroidSpeak on improving the inference throughput[2] of an online LLM inference system, we emulated an online inference scenario by pairing the datasets with request arrival times following a Poisson distribution under different incoming rates to evaluate the performance of DroidSpeak in more practical workloads.

For the experiment, we deployed a Kubernetes cluster running vLLM Production Stack vLLM Production Stack Team [2025], using DroidSpeak 's customized Docker image. The cluster consisted of two nodes, each equipped with 8 A100 GPUs. We configured eight replicas for each model across these nodes. Model placement followed a simple strategy: for each model pair, we deployed four replicas of both the sender and receiver models on each node. Request routing was handled by vLLM Production Stack's round-robin algorithm, which splits incoming requests evenly across all model replicas.

As demonstrated in Figure 4.11, we compare the TTFT, TBT, and E2E impact under various request rates on the HotpotQA dataset. Due to the limit in space, we only show four pairs of models to illustrate. For DroidSpeak, we chose the configuration within 1% accuracy drop for these pairs of models.

TTFT: Since the full-recompute baseline has much higher prefill latency than DroidSpeak, the queuing delay affects (knee in the curve) its TTFT at a much lower QPS than what DroidSpeak can support.

TBT and E2E: Although we are only reducing the TTFT directly in DroidSpeak, the second-degree effect through less interference and better scheduling brings down the TBT and E2E latency too, as shown in Figure 4.11.

Throughput: Assuming an SLO that avoids the effects of high queuing delays (knee of full prefill) on TTFT, TBT, and E2E latency, DroidSpeak can support 2-4× higher throughput.

2. Here, inference request throughput refers to the number of requests the system can process per second, not network bandwidth throughput.

Although DroidSpeak can only reduce prefill computation, it becomes a bottleneck in the system, especially under long contexts, as long TTFTs cause the subsequent decoding to be queued for a long time. By reducing the TTFT bottleneck, DroidSpeak can thus increase the inference throughput.

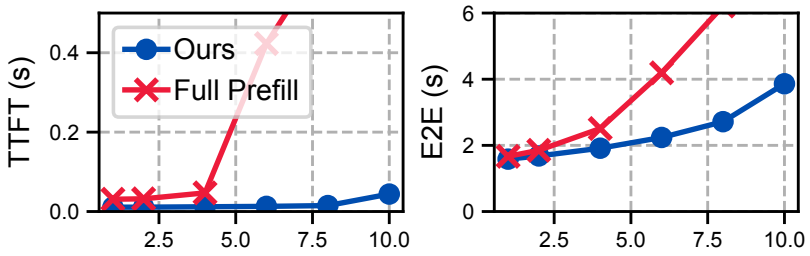


Figure 4.12: *Impact of the arrival rate on the TTFT and E2E delays of code agentic workflow.*

4.5.4 Robustness across datasets

As discussed in §4.4.3, DroidSpeak profiles the KV cache reuse pattern using a single profiling run on a training dataset during the offline stage and then generalizes the profile results to other datasets during the online stage.

Figure 4.13 illustrates whether the profile obtained on one dataset offline generalizes well to other datasets. In each subfigure, we plot the Pareto frontier of the F1 score versus the number of reused layers, obtained through profiling on the original testing dataset vs two other datasets in our benchmark using `glue_sst2` and `conllpp` model pair.

The figure demonstrates that the Pareto frontier obtained using the profile from the training dataset on the testing dataset closely resembles the frontier obtained using the profile directly from the testing dataset. Across all the pertinent configurations, the maximum difference in the score is 4 points, with the average being 2 points. This result further validates the sufficiency and robustness of our profiling strategy.

Case study on coding agentic workflow: Next, we study how DroidSpeak performs under a real agentic workflow by orchestrating a coding agent system using MetaGPT Hong

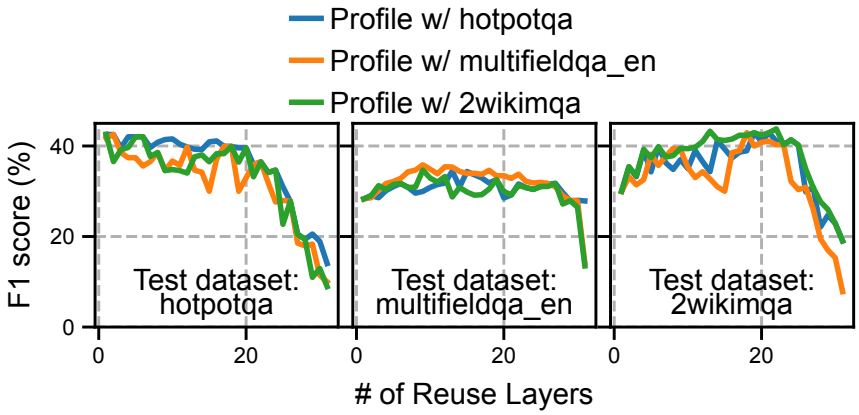


Figure 4.13: *Using the recompute layers profiled on training datasets works well on testing datasets.*

et al. [2024b], a state-of-the-art multi-agent framework. The system consists of two agents, a coder using evolcode model and a tester using tool-8b model. The coder is responsible for implementing Python functions according to the input prompt, and the tester is responsible for testing the coder's code and providing comments to the coder agent for the next round's modification.

We send the problems from the HumanEval dataset at various rates. In Figure 4.12, we plot the TTFT and E2E impact under different QPS, similar to the setup in §4.5.3, where DroidSpeak is plotted with the re-computation configuration that maintains the generation quality (pass@1 score of 52.5). DroidSpeak significantly improves the TTFT by 2.7× and brings down the E2E delay to finish the problem as well.

Mixture-of-Experts Model: We also evaluate DroidSpeak on a Mixture-of-Experts (MoE) model. As shown in Figure 4.14, where the sender model is Mixtral-8x7B and the receiver is Mixtral-8x7B-Instruct, DroidSpeak is able to achieve significant reductions in prefill latency as well. While MoE models may activate different experts for different inputs, this selection occurs only at the linear layers after the attention modules—where the KV cache is used in. Thus, DroidSpeak 's KV cache sharing remains effective for MoE architectures.

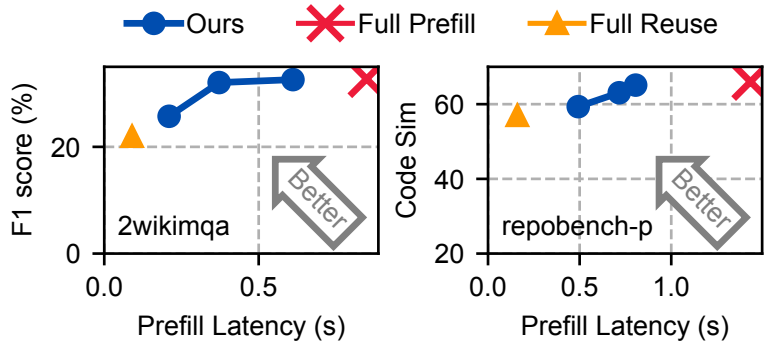


Figure 4.14: *Applying DroidSpeak on Mixtral-8x7B-Instruct sharing Mixtral-8x7B's KV cache, which are MoE models based on Mistral model architecture.*

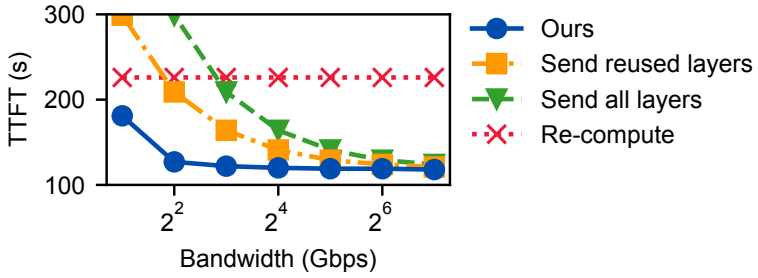


Figure 4.15: *Pipelining re-compute and loading the reused layers greatly reduces the total TTFT compared to the baselines of sequentially send and re-computes the KV cache.*

4.5.5 Impact of different network bandwidth

Figure 4.15 compares DroidSpeak and the baselines where the re-computation and loading of KV cache are not pipelined. We can see that DroidSpeak outperforms baselines under almost all bandwidth situations. Arguably, the absolute reduction in TTFT becomes smaller under high bandwidth, because the transmission delay becomes a smaller amount of the overall delay when the bandwidth is very high. We acknowledge that as networking speed increases, the advantage of loading KV caches in parallel with recomputation decreases compared to simply loading all or reused layers.

4.5.6 Profiling Overhead

Figure 4.16 shows the trade-off between profiling cost and generation quality. Profiling one layer at a time for the `glue_sst2` and `conllpp` model pair with 32 layers takes 3.6 hours,

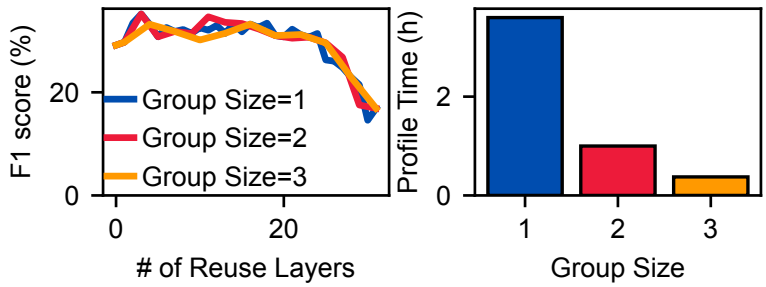


Figure 4.16: *DroidSpeak's profiling delay on glue_sst2 and conllpp model pair, on the multifieldQA dataset. Grouping the profiling layers at the granularity of multiple layers reduces profiling delay, without losing quality.*

while grouping layers cuts this to 1 hour for 2-layer groups and 0.375 hours for 3-layer groups. The left side shows that coarser grouping maintains similar F1 scores and layer reuse trade-off compared to per-layer profiling. We observe the same trend for the `evolcode` and `tool-8b` pair in §B.2.3.

4.6 Limitation and Future Work

KV cache sharing across different foundation models: DroidSpeak as-is only works for KV cache sharing across models originated from the same foundational model. Exploring the possibility and solution of KV cache sharing across different foundational models is a challenging problem that we leave for future work.

Re-computation adaptation with network bandwidth: In §4.4.3, we only consider adjusting the re-computation ratio based on system load. Future work may extend the adaptation algorithm to consider changes in network bandwidth, for example, expanding the range of critical layer groups when network bandwidth is limited.

Data drift in the re-computation configuration: DroidSpeak profiles the re-computation configuration for different models *offline*. We acknowledge that this profiling approach may result in quality degradation if the real test data drifts greatly from the training data used for profiling. To alleviate this issue, periodic re-profiling can be performed to update the re-computation configuration with newer data, which has closer distribution to the testing

dataset. We leave this challenge to future work.

Applicability of DroidSpeak: Since DroidSpeak relies on the empirical insight that only a subset of layers are sensitive to KV cache reuse errors, it may not work for all model pairs, even those sharing the same foundation model. That said, we have validated that for model pairs in Section 4.5 and Section 4.3, DroidSpeak is able to significantly reduce prefill computation with little impact on generation quality.

KV cache reuse across multiple LLMs: Our current evaluation focuses on KV cache reuse between *two models*. Currently, KV cache reuse among *multiple models* needs to treat them as a collection of model pairs. We leave the optimizations of multi-model reuse to future work, such as maximizing shared layers across models with little impact in generation quality.

4.7 Conclusion

In this work, we identified the core challenge of reducing repetitive computation in systems where multiple models work on a shared context. We presented DroidSpeak, a framework for KV cache sharing in compound AI systems. We identified only a subset of layers that require recomputation to maintain quality and show the robustness of our solution for a diverse range of model pairs, model types, and datasets.

CHAPTER 5

CHAMELEONAPI: ENCODING SOFTWARE CONTEXT INTO MODEL BEHAVIOR

5.1 Introduction

The landscape of ML applications has greatly changed, with the rise of ML APIs significantly lowering the barrier of ML application developers. Instead of designing and managing neural network models by themselves via frameworks like TensorFlow and PyTorch, application developers can now simply invoke ML APIs, provided by open-source libraries or commercial cloud service providers, to accomplish common ML tasks like object detection, facial emotion analysis, etc. This convenience thus gives rise to a variety of ML applications on smartphones, tablets, sensors, and personal assistants McEnroe et al. [2022], Chang et al. [2021], Hua et al. [2022], Wan et al. [2021].

Although ML APIs have eased the integration of ML tasks with applications, they are suboptimal by serving different applications with the same neural network models. This issue is particularly striking when applications use the ML API results to make control-flow decisions (also referred to as *application decisions* in this paper). Different applications may check the result of the same ML API using different control-flow code structures and different condition predicates, a process that we refer to as the application's *decision process* (see §5.2 for the formal definition). Due to the heterogeneity across applications' decision processes, we make two observations.

- First, some incorrect ML API outputs may still lead to correct application decisions, with only certain *critical errors* of API output affecting the application's decision.
- Second, among all possible output errors of an ML API, which ones are critical *vary* significantly across applications that use this API. That is, the same API output error may have a much *greater* effect on one application than on another.

Figure 5.1 illustrates the decision process of `Heapsortcypher` Matthew [2019], a garbage-classification application. It first invokes Google's classification API upon a garbage image. Then, based on the returned labels, a simple logic is used to make the *application decision* about which one of the pre-defined categories (`Recycle`, `Compost`, and `Donate`) or `others` the image belongs to. For example, for an input image whose ground-truth label is ``Shirt'', the correct application decision is `Donate`, as shown in Figure 5.1 (b).

For this application, when the classification API fails to return ``Shirt'', the application decision may or may not be wrong. For example, Figure 5.1 (c) and (d) show two possible wrong API output: if the output is ``Paper'', the application will make a wrong decision of `Recycle`; however, if the output is ``Jacket'', the application will make the correct decision of `Donate` despite not matching the ground-truth label. More subtly, if the API returns a list of two labels, ``Shirt'' and ``Paper'', the application would make a correct decision if ``Shirt'' is ordered before ``Paper'' by the API, but would make a wrong decision if ``Paper'' is ordered before ``Shirt''. The reason is that the application logic, the `for` loop in Figure 5.1 (a), checks one API-output label at a time. As we will see later, there are also other ways that applications check the API-output list, which will affect application decision differently.

As we can see, for a specific application, some errors of an ML API may be critical, like mis-classifying the shirt to ``Paper'' in the example above, and yet some errors may be non-critical, like mis-classifying the shirt image as ``Jacket'' or classifying the shirt image as both ``Shirt'' and ``Paper'' in the examples above. Which errors are critical varies, depending on the application's decision process.

These observations regarding the critical errors specific to each application suggest substantial room for improvement by customizing the ML API, essentially the neural network model underneath the API, for individual application's decision process. In particular, for a given application, the customized model can afford having more errors less critical to the application for the benefit of having fewer critical errors that cause wrong application

decisions.

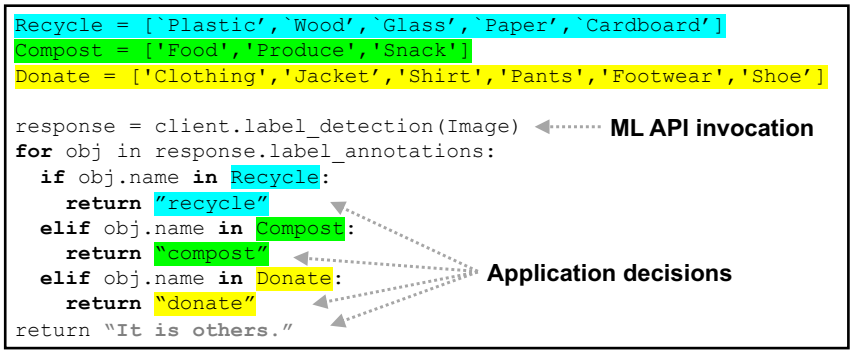
Thus, our goal is to allow ML APIs and their underlying neural network models to be *automatically* customized for a given application, so as to *minimize* incorrect application decisions *without* changing the application's source code or interface between ML API and software exposed to developers. This way, application developers who do not have the expertise to design and train customized ML models can still enjoy the accessibility of generic ML APIs while getting closer to the accuracy of ML models customized for the application.

No prior work shares the same goal as us. The closest line of prior work specializes DNN models for given queries Kang et al. [2018, 2017], Koudas et al. [2020], Cao et al. [2022, 2021], but they require application developers to use a domain specific language (*e.g.,* in SQL Kang et al. [2018]) instead of general programming languages, like Java and Python, and mostly focus on reducing the DNN's size. In contrast, we keep both the ML API interface and the application source code intact while avoiding incorrect decisions for ML applications.

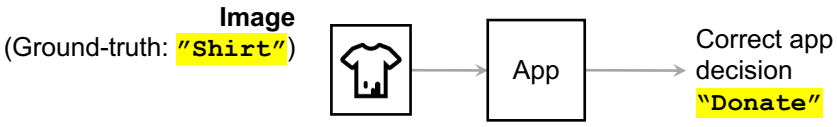
With the aforementioned goal, this paper makes two contributions. *First,* we run an empirical study over 77 real-world applications that collectively use six ML APIs to reveal several common patterns of how the outputs of ML APIs affect the application decisions (§5.2).

Our study identifies two types of ML API output that are used by applications to make control-flow decisions (categorical labels and sentiment scores), and three types of decision types (True-False, Multi-Choice, and Multi-Selection) with different implications regarding which ML API output errors are critical to the application.

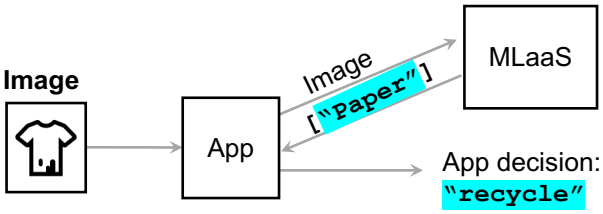
Our study also quantitatively reveals opportunities of model customization. (1) Although popular image-classification models are trained to recognize as many as 19.8K different labels, the largest number used by any one application for decision making is only 54. Consequently, mis-classification among the remaining tens of thousands of labels are completely irrelevant to an application. (2) More importantly, applications tend to treat multiple labels (4.7 on



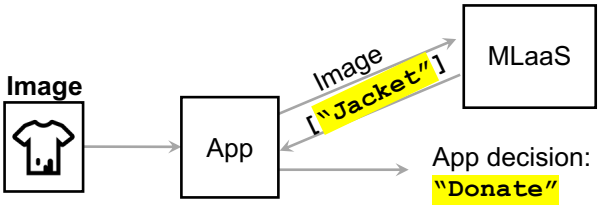
(a) Code snippet of app `Heapsortcypher`



(b) Correct app decision based on input image's ground-truth label



(c) ML API output error leads to wrong app decision



(d) ML API output error still leads to correct app decision

Figure 5.1: *An example ML application whose decision depends on the output of ML API (multi-label classification), but not all errors of ML API output have the same effect.*

average) as one equivalence class in their decision making, such as labels `Plastic`, `Wood`, `Glass`, `Paper`, and `Cardboard` in Figure 5.1(a). Mis-classification among those labels inside one equivalence class does not matter. (3) Which labels are relevant to an application's decision making vary greatly across applications, with only 12% of application pairs share any labels used for their decision making.

Second, inspired by the empirical study, we propose ChameleonAPI, which customizes and serves ML models behind the ML API for each given application's decision process, without any change to the existing ML API or the application source code (§5.3). ChameleonAPI works in three steps. First, it provides a parser that analyzes application source code to extract information about how ML inference results are used in the application's decision process. Based on the analysis result, ChameleonAPI then constructs the loss function to reflect which ML model output is more relevant to the given application as well as the different severity of ML inference errors on the application decisions. The ML model will be retrained accordingly using the new loss function. Finally, when the ML API is invoked by the application at runtime, a customized ML model will be used to serve this query.

We evaluate ChameleonAPI on 57 real-world open-source applications that use Google and Amazon's vision and language APIs. We show that ChameleonAPI's re-trained models reduce 48% of incorrect decisions compared to the off-the-shelf ML models and 50% compared to the commercial ML APIs. Even compared with a baseline that selects the best-of-all commercial ML API, ChameleonAPI reduces 43% of incorrect decisions. ChameleonAPI only takes up to 24 minutes on a GeForce RTX 3080 GPU to re-train the ML model.

Our code is publicly available at `https://github.com/UChi-JCL/chameleonAPI`.

5.2 Understanding Application Decision Process

Section 2.3 gave a brief introduction to ML APIs and how developers invoke them in their applications. Here, we conduct an empirical study to understand how applications make

decisions based on ML APIs (§5.2.3), and how this decision making logic implies the different severity of ML inference errors (§5.2.4). This study will reveal why and how to customize the ML API backend for each application. As a representative sample of ML APIs, this study focuses on cloud AI services due to their popularity.

5.2.1 Definitions

Preliminaries: We begin with basic definitions.

- *Application decision*: the collective control-flow decisions (i.e., which branch(es) are taken) made by the application under the influence of a particular ML API output.
- *Incorrect ML API output:* a situation when the API output differs from the API input's human-labeled ground truth. We refer to such ML API outputs as *API output errors*.
- *Correct decision*: the application decision if the API output is the same as the human-labeled ground-truth of the input.
- *Application decision failure*: a situation when the application decision is different from the correct decision, also referred to as *application failure* for short in this paper.

Software decision process: Given these definitions, an application's *software decision process* (or decision process for short) is the logic that maps an ML API output to an application decision. The code snippet in Figure 5.1 shows an example decision process, which maps the output of a classification ML API on an image to the image's recycling categorization specific to this application.

Critical and non-critical errors: For a given decision process, some API output errors will still lead to a correct decision, whereas some API output errors will lead to an incorrect decision and hence an application failure. We refer to the former as *non-critical errors*, and the latter as *critical errors*.

5.2.2 Methodology

ML API name	ML task	Provider	# of apps
label_detection	Vision::Image classification	Google	29
detect_labels	Vision::Image classification	Amazon	11
object_localization	Vision::Object detection	Google	8
analyze_sentiment	Language::Sentiment analysis	Google	14
analyze_entities	Language::Entity recognition	Google	6
classify_text	Language::Text classification	Google	9

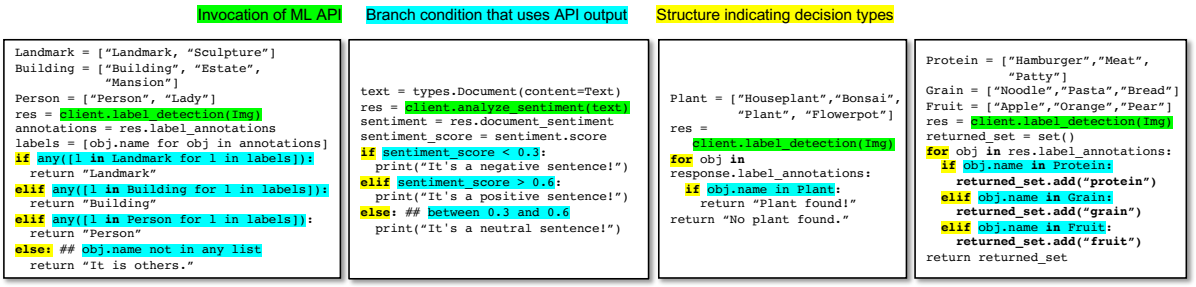
Table 5.1: *Summary of applications used in our empirical study.*

Our work focuses on applications that use ML API output to make control-flow decisions. To this end, we look at 77 open-source applications which collectively use six widely used vision and language APIs Wan et al. [2021], Chen et al. [2021a] offered by two popular cloud AI service providers, as summarized in Table 5.1.

These applications come from two sources. First, we study all 50 applications that use vision and language APIs from a recently published benchmark suite of open-source ML applications Wan et al. [2022]. Second, given the popularity of image classification APIs Chen et al. [2020a, 2022b], we additionally sample 27 applications from GitHub that use Google and Amazon image classification APIs (16 for the former and 11 for the latter). We obtain these 27 by checking close to 100 applications that use image classification APIs and filtering out those that directly print out or store the API output. Every application in our benchmark suite uses exactly one ML API for decision making.

Threats to validity: While many applications use the APIs listed in Table 5.1, there are a few other APIs not covered in our study. A few vision and language-related ML tasks are not as popular and hence are not covered in our study (*e.g.,* face recognition and syntax analysis). Speech APIs are not covered, because their outputs are rarely used to affect application control flow based on our checking of open-source applications. Finally, our study does not cover applications that use ML APIs offered by other cloud or local providers.

5.2.3 *Understanding the decision mechanism*



(a) `Aander-ETL`
(Label output, Multi-Choice, App-Order)

(b) `FoodDelivery`
(Float-point value output, Multi-Choice)

(c) `Plant-watcher`
(Label output, True-False)

(d) `The-Coding-Kid`
(Label output, Multi-Select)

Figure 5.2: *Code snippets from five example applications where ML API output affects control flow decisions in different ways.*

***Q1**: What types of ML API outputs are typically used by applications to make decisions?*

ML APIs produce output of a variety of types. The sentiment analysis API outputs a list of floating-point value pairs (`score` and `magnitude`), describing the sentiment of the whole document and every individual sentence; the other five APIs in Table 5.1 each produces a list of categorical labels ranked in descending order of their confidence scores, which is also part of the output. Some APIs' output also contains other information, like coordinates of bounding boxes, entity names, links to Wikipedia URLs, and so on. Among all these, only two types have been used in application decision processes of our studied application: the floating-point pair (`score` and `magnitude`) and the categorical labels.

For the 63 applications that use categorical-label output from the five APIs (all except `analyze_sentiment` in Table 5.1), they each define one or more *label lists* and check which label list(s) an API output label belongs to. The code snippet of a landmark classification application in Figure 5.2(a) is an example of this. It calls the `label_detection` API with a sight-seeing image and checks the output labels to see if the image might contain `Landmark`, or just ordinary `Building`, or `Person`.

For the 14 applications that use the `analyze_sentiment` API, they each define several

value ranges and check which range the sentiment `score` and/or `magnitude` falls in. The code snippet of `FoodDelivery` Martincu [2020] in Figure 5.2(b) is an example. This application calls `analyze_sentiment` with a restaurant review text, and then checks the returned sentiment `score` to judge if the review is negative, positive, or neutral.

***Q2**: What type of decisions do applications make?*

We observe three categories of ML-based decision making, which we name following common question types in exams:

(1) True-False decision, where a single label list or value range is defined and one selection is allowed: either the ML API output belongs to this list/range or not. This type occurs in about one third of the applications in our study. For example, the plant management application `Plant-watcher` Plant-Watcher (Figure 5.2(c)) checks to see if the image contains plants or not.

(2) Multi-Choice decision, where multiple lists of labels or value ranges are defined, and one selection is allowed. The ML API output will be assigned to *at most one* list or range; the application's decision making logic determines which of these lists/ranges the output belongs to, or determines that the output belongs to none of them. This type of decision is the most common, occurring in about 45% of benchmark applications. The garbage classification application discussed in §5.1 makes such a Multi-Choice decision. It decides which one of the following classes the input image belongs to: `Recycle`, `Compost`, `Donate`, or none of them.

(3) Multi-Select decision, where multiple label lists or value ranges are defined, and *multiple* selections are allowed about which label lists or value ranges the ML API output belongs to. This type of decisions occur in close to a quarter of the applications. Figure 5.2(d) illustrates such an example from the nutrition advisor application `The-Coding-Kid` The-Coding-Kid. This application defines three label lists to represent nutrition types: `Protein`, `Grain`, and `Fruit`, and it checks to find all the nutrition types present in the input image.

In the remainder of the paper, we will use **target class** to refer to a label list (or a

value range) that is used to match against a categorical label (or a value). For instance, the code snippet in Figure 5.2(a) has three label lists as its target classes (`[Landmark, Sculpture]`, `[Building, Estate, Mansion]`, and `[Person, Lady]`), and the code snippet in Figure 5.2(b) has three value ranges as its target classes (`<0.3`, `>0.6`, and in between).

***Q3**: How do applications reach Multi-Choice decisions?*

When the ML API outputs multiple labels, the outcome of a Multi-Choice decision varies depending on which *matching order* is used. First, the matching order can be determined by the API output. For example, the garbage classification application (Figure 5.1) first checks whether the first label in the API output matches any target class. If so, later API output labels will be skipped, even if they might match with a different class. If there is no match for the first label, the second output label is checked, and so on. These labels are ranked by the API in the descending order of their associated confidence scores, so we refer to such a matching order as API-order. It is used by 80% of applications that make Multi-Choice decisions.

The matching order can also be specified by the application, referred to as App-order. For instance, regardless the API output, application `Aander-ETL` Aander-ETL (Figure 5.2(a)) always first checks if the `Landmark` class matches with *any* output label. If there is a match, the decision is made. Only when it fails to match `Landmark`, will it move on to check the next choice, `Building`, and so on. This matching order is used by 20% of applications that make Multi-Choice decisions.

5.2.4 Understanding the decision implication

***Q4**: Does an application need ML APIs that can accurately identify thousands of labels?*

ML models behind popular ML APIs are well trained to support a wide range of applications. For example, Google and Microsoft's image-classification APIs are capable of identifying more than 10000 labels Kuznetsova et al. [2020], while Amazon's image-classification

API can identify 2580 labels Amazon [2022b]. However, for each individual application, its decision making only requires classifying the input image into a handful of target classes: 7 at most in our benchmark applications. The largest number of image-classification labels checked by an application is 54, a tiny portion of all the labels an image-classification API could output.

Clearly, for any application, a customized ML model that focuses on those target classes used by the application's decision process has the potentially to out-perform the big and generic ML model behind ML APIs. How to accomplish the customization without damaging the accessibility of ML APIs will be the goal of ChameleonAPI.

Q5*: Are there equivalence classes among ML API outputs in the context of application decision making?*

For the 63 applications that make decisions based on API output of categorical labels, they present 121 target classes in total, each containing 4.7 labels on average (3 being the median). Only 35 target classes in 22 applications contain a single label. For the 14 applications that make decisions based on floating-point sentiment `score` and `magnitude`, their target classes *all* contain an infinite number of `score` or `magnitude` values. In other word, no class contains just a single value.

Clearly, the wide presence of multi-value target classes creates equivalence classes among output returned by the API—errors within one equivalence class are *not* critical to the corresponding application. This offers another opportunity for ML customization.

Q6*: How much difference is there between different applications' target classes?*

Overall, the difference is significant. We have conducted pair-wise comparison between any two applications in our benchmark suit, and found that 88% of application pairs share *no* common labels in any of their target classes. Similarly, among the 381 labels that appear in at least one application's target classes, 88% of them appear in only one application (*i.e.,* 335 out of 381 labels).

Clearly, there is *little overlap* among the target classes of different applications, again making a case for *per-application* customization of the ML models used by the ML APIs.

Q7*: Do different decision mechanisms imply different sensitivity to output errors of ML APIs?*

Even for two applications that have the same target classes, if they try to make different types of decisions, they will have different sensitivity to ML API output errors—some API errors might be critical to one application, but not to the other. For example, errors that affect the selection of different target classes are equally critical to Multi-Select decisions. However, this is not true for Multi-Choice, where only the first matched target class matters. Furthermore, the matching order of a Multi-Choice decision affects which errors are critical. When the API-output order is used (*e.g.,* `HeapsortCypher` in Figure 5.1), an error on the first label in the API output is more likely to be critical than an error on other labels in the output. However, when App-order order is used (*e.g.,* `Aander-ETL` in Figure 5.2(a)), errors related to labels in the first target class (*e.g.,* `Landmark`) are more likely to be critical than those related to labels in later target classes (*e.g.,* `Person`).

Clearly, to customize ML models for each application, we need to take into account what is the decision type and what is the matching order (for Multi-Choice decisions).

5.3 Design of ChameleonAPI

Inspired by the study of §5.2, we now present ChameleonAPI which automatically customizes ML models for applications.

5.3.1 Problem formulation

Goal: For an application that uses ML APIs, our goal is to **minimize critical errors** in the API outputs for this application by efficiently re-training the original generic neural network models underneath these APIs into customized models; our approach stands in

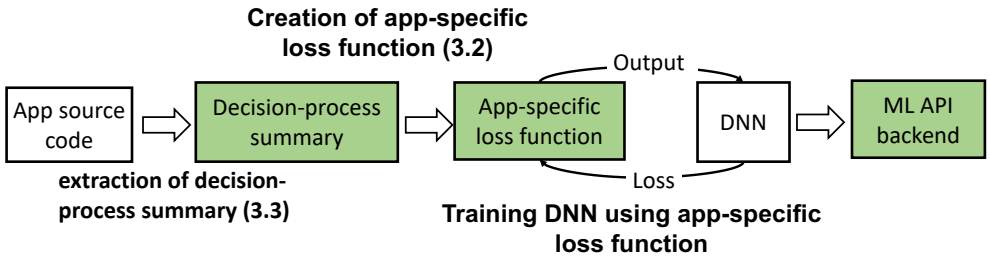


Figure 5.3: *The logical steps of how ChameleonAPI customizes d for individual applications.*

contrast to typical approaches that minimize *all* inference errors. In other words, the new ML model should return outputs that lead the application process to the same decision as if the ground-truth of the input is returned by the ML API.

To formally state this objective, we denote how an application makes a decision by $App(API(\mathbf{x}))$, where $\mathbf{x}$ is the input to the ML API and $API(\mathbf{x})$ is the API output. Then for a given application decision process of $App(\cdot)$ and an input set $\mathbf{X}$[1], our goal is to train an ML model $DNN(\cdot)$ such that

$$\min_{\mathbf{x}_i \in \mathbf{X}} \left| \{\mathbf{x}_i | App(API(\mathbf{x}_i)) \neq App(\widehat{API}(\mathbf{x}_i))\} \right|, \quad \text{where } API(\mathbf{x}_i) = F(DNN(\mathbf{x}_i)) \tag{5.1}$$

Here, $\widehat{API}(\mathbf{x}_i)$ is a hypothetical API function that always returns the ground truth of input $\mathbf{x}_i$, and $F(\cdot)$ represents the postprocessing used by the API to translate a DNN output to an API output. For instance, an image classification model's output is a vector of confidence scores between 0 and 1 (each for a label), but the ML API will use a threshold θ to filter and return only labels with scores higher than θ, or the top k labels with the highest confidence scores.

1. A careful reader might notice that the formulation in Eq. 5.1 also depends on the input set. Though the input set should ideally follow the same distribution of real user inputs of the application, this distribution is hard to obtain in advance and may also vary over time and across users. Instead, we focus our discussion on training the ML model to minimize Eq. 5.1 with an assumed input distribution. Our evaluation (§5.5) will test the resulting model's performance over different input distributions.

Our goal in Eq 5.1 differs from the traditional goal of an ML model, which minimizes any errors in the API output, *i.e.,*

$$\min_{\mathbf{x}_i \in \mathbf{X}} \left| \{\mathbf{x}_i | API(\mathbf{x}_i) \neq \widehat{API}(\mathbf{x}_i)\} \right|. \tag{5.2}$$

Given that it is hard to obtain a DNN with 100% accuracy, the difference between the two formulations is crucial, since not all API output errors in Eq. 5.2 will cause incorrect application decisions in Eq. 5.1. Thus, compared to optimizing Eq. 5.2, optimizing Eq. 5.1 is more likely to focus the DNN training on reducing the critical errors for the application.

To train a DNN that optimizes Eq. 5.1, we need to decide if a DNN inference output $DNN(\mathbf{x})$ is a critical error or not (*i.e.,* $App(DNN(\mathbf{x})) \neq App(\widehat{API}(\mathbf{x}))$) at the end of *every* training iteration. This decision needs to be made automatically and efficiently. For example, repeatedly running the entire ML application after every training iteration would not work, as it may significantly slow down the training procedure.

Logical steps of ChameleonAPI: To customize and deploy the DNN for an application, ChameleonAPI takes three logical steps (Figure 5.3). First, ChameleonAPI extracts from an application's source code a *decision-process summary* (explained shortly), a succinct representation of the application's decision process, which will be used to determine if a DNN inference error is critical (details in §5.3.3). Second, ChameleonAPI converts a decision-process summary to a *loss function*, which can be directly used to train a DNN (details in §5.3.2). This loss function only penalizes DNN outputs that lead to critical errors with respect to a given application. Finally, the loss function will be used to train a customized DNN for this particular application's ML API invocations (§5.3.4).

A decision-process summary is a succinct abstraction of the application that contains enough information to determine if a DNN inference output causes a critical error or not. Specifically, it includes three pieces of information (defined in §5.2.3):

- *Composition of target classes:* the label list or value range of each target class;
- *Decision type:* True-False, Multi-Choice, or Multi-Select;
- *Matching order:* over the target classes, API-order or App-order, if the application makes a Multi-Choice decision.

For a concrete example, the decision-process summary of the garbage classification application in Figure 5.2(a) contains (1) three label lists representing three target classes: `Recycle`, `Compost`, and `Donate`; (2) the Multi-Choice type of decision; and (3) the matching order of API-order.

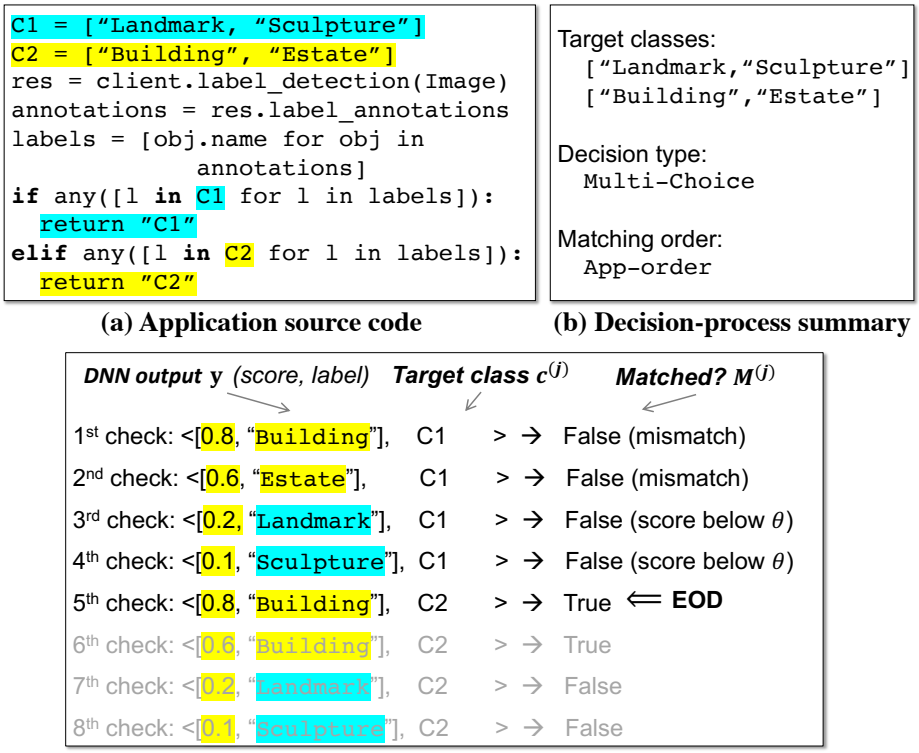
What is changed, what is not: ChameleonAPI does *not* change the ML API or the application source code. Unlike recent work that aims to shrink the size of DNNs or speed them up Mullapudi et al. [2019], Kang et al. [2018, 2017], we do not change the DNN architecture (shape and input/output interface); instead, we train the DNN to minimize critical errors. That said, deploying ChameleonAPI has two requirements. First, the application developers need to run ChameleonAPI's parser script to automatically extract the decision-process summary. Second, an ML model needs to be retrained for each application, instead of serving the same model to all applications.

The remainder of this section will begin with the design of the application-specific loss function based on decision-process summary, followed by how to extract the decision-process summary from the application, and finally, how the customized ML models are used to serve ML API queries.

5.3.2 Application-specific loss function

Given Eq 5.1, ChameleonAPI trains a DNN model with a *new loss function*, which only penalizes critical errors of an application, rather than all DNN inference errors. Since decision processes vary greatly across applications (§5.2.4), we first explain how to conceptually

capture different decision processes in a generic description, which allows us to derive the mathematical form of ChameleonAPI's loss function later.



(a) Application source code **(b) Decision-process summary**

(c) Generic description of the decision process
on DNN output $y = [0.1, 0.2, 0.8, 0.6]$ & score threshold $\theta = 0.3$

Figure 5.4: *The generic description (shown in (c)) of an application (whose source code is shown in (a) and decision-process summary in (b)) on a DNN inference output* $\mathbf{y}$.

Generalization of decision processes: For each application in our study (§5.2.2), our insight is that its decision process can always be viewed as traversing a sequence of conditional *checks* until an *end-of-decision* (*EOD*) occurs:

$$1^{\text{st}} \text{ check: } < \mathbf{y}, c^{(1)} > \rightarrow M^{(1)}$$

$$\ldots$$

$$j^{\text{th}} \text{ check: } < \mathbf{y}, c^{(j)} > \rightarrow M^{(j)} \qquad \Leftarrow \textbf{EOD}$$

$$\ldots$$

where the j-th check takes as input the DNN output $\mathbf{y}$ and one of target classes $c^{(j)}$, and returns a binary $M^{(j)}$ indicating whether $\mathbf{y}^{(j)}$ *matches* the condition of $c^{(j)}$ and a binary decision whether this check happens before the EOD. The set of target classes successfully matched before the EOD will be those selected by the application.

Figure 5.4 shows (a) an example application, (b) the decision-process summary, and (c) the generic description for this application's decision process and a DNN output.

This generic description (*e.g.,* the traversal order of the target classes, how a match is determined in a check, and when the EOD occurs) will depend on the information in the decision-process summary and the DNN output $\mathbf{y}$. We stress that this generic description may *not* apply to all applications, but it does apply to all applications in our study (§5.2.2).

Categorization of critical errors: Importantly, this generic description helps to categorize critical errors:

- *Type-1 Critical Errors:* A correct target class c is not matched before EOD, but will be so if EOD occurs later.
- *Type-2 Critical Errors:* A correct target class c is never matched, before or after the EOD.
- *Type-3 Critical Errors:* An incorrect target class c is matched before EOD.

A useful property of this categorization is that any wrong decision (a correct target class not being picked, or an incorrect target class being picked) falls in a unique category, and non-critical errors do not belong to any category. In other words, as long as the loss function penalizes the occurrences of each category, it will only capture critical errors.

ChameleonAPI's first attempt of a new loss function: To understand why it is difficult to penalize critical errors and critical errors *only*, we first consider the common practice of assigning a higher weight to the loss of a DNN output if the ground-truth of the input will lead to a selection of some target classes (*e.g.,* Han et al. [2016], Jung et al.

[2015], Vasan et al. [2020]). Henceforth, we refer to this basic design of loss function as ChameleonAPI$_{basic}$.

At best, ChameleonAPI$_{basic}$ might improve the DNN's *label-wise* accuracy on inputs whose ground-truth decision selects some target classes. However, as elaborated in §5.2.3, we also need to consider which labels belong to the same target class, the decision type, and the matching order of an application decision process in order to capture the three types of critical errors. For instance, in the garbage-classification application (Figure 5.1), without knowing the label lists of each target class, ChameleonAPI$_{basic}$ will give an equal penalty to a critical error of mis-classifying a `Paper` image to `Wood` and a non-critical error of mis-classifying a `Paper` image to `Shirt`. Similarly, without knowing the matching order, ChameleonAPI$_{basic}$ will equally penalize the output of [`Plastic`, `Jacket`] and [`Jacket`, `Plastic`], but only the latter leads to correct output because `Jacket` is matched first.

ChameleonAPI's loss function: ChameleonAPI leverages the categorization of critical errors to systematically derive a loss function that penalizes each type of critical error. To make it concrete, we explain ChameleonAPI's loss function of "label-based API, Multi-Choice type of decision, and App-order" (*e.g.,* Figure 5.4). Appendix§C.2 will detail the loss functions of other decision processes. The loss function of such applications has three terms, each penalizing one type of critical error:

$$L(\mathbf{y}) = \overbrace{\text{Sigmoid}\left(\min\left(\max_{l\in\cup_{c<\hat{c}}G_c}\mathbf{y}[l], \max_{l\in G_{\hat{c}}}\mathbf{y}[l]\right)-\theta\right)}^{\textbf{Type-1 Critical Errors}} + \overbrace{\text{Sigmoid}\left(\theta-\max_{l\in G_{\hat{c}}}\mathbf{y}[l]\right)}^{\textbf{Type-2 Critical Errors}} + \overbrace{\sum_{c<\hat{c}}\text{Sigmoid}\left(\max_{l\in G_c}\mathbf{y}[l]-\theta\right)}^{\textbf{Type-3 Critical Errors}} \tag{5.3}$$

Here, $\mathbf{y}[l]$ denotes the score of the label l, G_c denotes the set of labels of target class c, $\hat{c}$ denotes the correct (*i.e.,* ground-truth) target class, and the sigmoid function $\text{Sigmoid}(x) = \frac{1}{1+e^x}$ will incur a higher penalty on a greater positive value.

Why does it capture the critical errors? Given this application is Multi-Choice, the EOD will occur right after the first match of a target class, *i.e.,* the first check with a c such that $\max_{l \in G_c} \mathbf{y}[l] \geq \theta$.

- A Type-1 critical error occurs, if (1) the correct target class $\hat{c}$ is matched *and* (2) it is matched after the EOD. First, the correct target class $\hat{c}$ is matched, if and only if at least one of its labels has a score above the confidence threshold, so $\max_{l \in G_{\hat{c}}} \mathbf{y}[l] \geq \theta$). Second, this match happens after the break, if and only if some target class c before $\hat{c}$ (*i.e.,* $c < \hat{c}$) is matched, so $\max_{l \in G_c} \mathbf{y}[l] \geq \theta$). Put together, the first term of Eq 5.3 penalizes any occurrence of these conditions.
- A Type-2 critical error occurs, if no label in the correct target class $\hat{c}$ has a score high enough for $\hat{c}$ to be matched, *i.e.,* $\max_{l \in G_{\hat{c}}} \mathbf{y}[l] < \theta$, so the second term of Eq 5.3 penalizes any occurrence of this condition.
- A Type-3 critical error occurs, if any incorrect target class c before $\hat{c}$ (*i.e.,* $c < \hat{c}$) has a label with a score high enough for c to be matched, *i.e.,* $\max_{l \in G_c} \mathbf{y}[l] - \theta$, so the third term of Eq 5.3 penalizes any occurrence of this condition.

To train a DNN, the loss function must be differentiable with respect to the DNN ouput $\mathbf{y}$. Eq 5.3 uses the max function several times. Though max is not naturally differentiable, it can be closely approximated in well-known differentiable forms provided by PyTorch's differentiable operators Paszke et al. [2017]).

5.3.3 Extracting applications' decision process

The current prototype of ChameleonAPI program analysis supports Python applications that make decisions based on categorical label output or floating point output of ML APIs. We first discuss how it works for ML APIs with categorical label output, like all the APIs in Table 5.1 except for `analyze_sentiment`. We will then discuss a variant of it that works

for most use cases of `analyze_sentiment`.

Given application source code, ChameleonAPI first identifies all the invocations of ML APIs. For every invocation I in a function f, ChameleonAPI then identifies all the branches whose conditions have a data dependency upon the ML API's label output. We will refer to these branches as I-branches. If there is no such branch in f, ChameleonAPI then checks the call graph, and analyzes up to 2 levels of callers and up to 5 levels of callees of f until such a branch is identified. If no such branch is identified after this, ChameleonAPI considers the ML API invocation I to not affect application decisions and hence does not consider any optimization for it. If some I-branches are identified, ChameleonAPI records the top-level function analyzed, F, and moves on to extract the decision-process summary in following steps.

What are the target classes? ChameleonAPI figures out all the target classes and their composition in two steps.

The first step leverages symbolic execution and constraint solving to identify all the labels that belong to *any* target classes. Specifically, ChameleonAPI applies symbolic execution to function F, treating the parameters of F and the label output of I as symbolic (*i.e.,* the symbolic execution skips the ML API invocation I and directly uses I's symbolic output in the remaining execution of F)[2]. Since applications typically match only one label in API output at a time (as observed in §5.2.3), we set the label array returned by I to contain one element (label) and use a symbolic string to represent it. Through symbolic execution, ChameleonAPI obtains constraints for every path that involves an I-branch, solving which tells ChameleonAPI which labels need to be in the output of the ML API in order to execute each unique path, essentially all the labels that belong to any target class.

One potential concern is that a solver may only output one instead of all values that satisfy a constraint. Fortunately, the symbolic execution engine used by ChameleonAPI,

2. Recall that an API output contains several fields not used to influence control flow in any applications. We set them with pre-defined dummy values.

NICE Irlbeck et al. [2015], turns Python code into an intermediate representation where each branch is in a simplest form. Take Figure 5.2(d) as an example, the source-code branch `if obj.name in Protein` is transformed into three branches where `obj.name` is compared with `‘‘Hamburger’’`, `‘‘Meat’’`, and `‘‘Patty’’` separately, allowing us to capture all three labels by solving three separate path constraints.

The second step groups these labels into target classes by comparing their respective paths: if two API output, each with one label, lead the program to follow the same execution path at the source-code level, these two labels belong to the same target class. For example, in Figure 5.2(d), the execution path is exactly the same when the `label_detection` API returns `[‘‘Hamburger’’]`, comparing with when it returns `[‘‘Meat’’]`, with all function parameters and other API output fields being the same. Consequently, we can know that label `Hamburger` and label `Meat` belong to the same target class. To figure out the path, ChameleonAPI simply executes function F using each input produced by the constraint solver and traces the source-code execution path using the Python trace module.

One final challenge is that ChameleonAPI needs to identify and exclude the path where none of the target classes are matched (*e.g.,* the `‘‘It is others.’’` path in Figure 5.2(a)). We achieve this by carefully setting the default solution in the constraint solver to be an empty string, which is impossible to output for any ML APIs in this paper. This way, whenever this default solution is output, ChameleonAPI knows that the corresponding path matches no target class.

What is the type of decision? When only one target class is identified, ChameleonAPI reports a True-False decision type. Otherwise, ChameleonAPI decides whether the decision type is Multi-Choice or Multi-Select by checking the source-code execution path associated with every target class label obtained above. If any execution evaluates an I-branch *after* another I-branch is already evaluated to be true, ChameleonAPI reports a Multi-Select decision type; otherwise, ChameleonAPI reports a Multi-Choice decision type.

What is the matching order over the target classes? To tell whether a Multi-Choice decision is made through API-Order like in Figure 5.1 or App-Order like in Figure 5.2(a), ChameleonAPI first identifies all the `for` loops that iterate through the label array output by the ML API and have control-dependency with I-branches, *e.g.,* the `for l in labels` in Figure 5.2(a) and the `for obj in response.label_annotations` in Figure 5.1.

ChameleonAPI then checks how many such output-iterating loops there are. If there is only one and this loop is not inside another loop, like that in Figure 5.1, ChameleonAPI considers the matching order to be API-Order, as the application only iterates through each output label once, with the matching order determined by the output array arranged by the ML API. Otherwise, ChameleonAPI considers the matching order to be App-Order. This is the case for the example shown in Figure 5.2(a), where three output-iterating loops are identified, each of which matches with one target class in an order determined by the application: the `Landmark` target class, followed by the `Building`, and finally the `Person`.

How to handle floating-point output of ML APIs? Recall in §5.2.3 that some ML APIs, *e.g.,* `analyze_sentiment`, have floating-point output and the application defines several value ranges to put each floating-point output into one category. To handle this type of API, ChameleonAPI needs to identify the value range of each target class, which is not supported by NICE and other popular constraint solvers. Fortunately, many applications directly compare API output with constant values in I-branches, giving ChameleonAPI a chance to infer the value range. For these applications, ChameleonAPI first extracts those constant values that are compared with API output in I-branches, *e.g.,* 0.3 and 0.6 in Figure 5.2(b). ChameleonAPI then forms tentative value ranges using these numbers, like -1 – 0.3, 0.3 – 0.6, and 0.6 – 1 for Figure 5.2(b) (-1 and 1 are the smallest and biggest possible `score` output of `analyze_sentiment` based on the API manual). To confirm these value ranges and figure out the boundary situation, ChameleonAPI then executes function F with all the boundary values, as well as some values in the middle of each range. By comparing which

values lead to the same execution path, ChameleonAPI finalizes the value ranges. For the example in Figure 5.2(b), after executing with `score` set to -0.35, 0.3, 0.45, 0.6, and 0.8, ChameleonAPI settles down on the final value ranges to be: (-1,0.3), [0.3,0.6), and [0.6,1).

Limitation The static analysis in ChameleonAPI does not handle the iterated object of while loops, unfolded loops, and recursive functions. For complexity concerns, ChameleonAPI only checks caller and callee functions with limited levels, and hence may miss some *I*-branches far away from the API invocation. ChameleonAPI's ability of identifying target classes is limited by the constraint solver. ChameleonAPI assumes different source-code paths correspond to different target classes, which in theory could be wrong if the application behaves exactly the same under different execution paths.

5.3.4 Putting them together

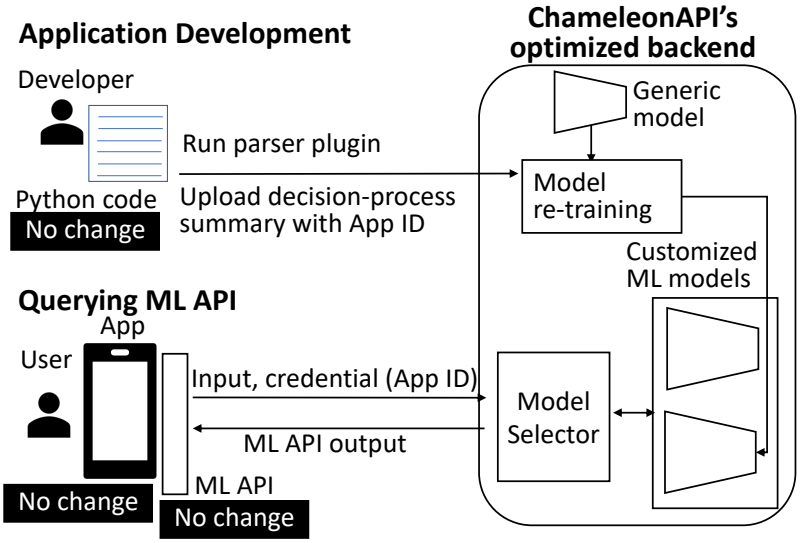


Figure 5.5: *Workflow of ChameleonAPI.*

We put these components together into a ML-as-a-Service workflow shown in Figure 5.5. First, when an application (A) is developed or updated, the developers run a parser (described in §5.3.3) provided by ChameleonAPI on A's source code to extract the decision-

process summary for A. The developers can then upload the decision-process summary to ChameleonAPI's backend together with a unique application ID[3] (which will later be used to identify queries from the same application).

ChameleonAPI's backend then uses the received decision-process summary to construct a new application-specific loss function (described in §5.3.2). When a DNN is trained using the new loss function, its inference results will lead to fewer critical errors (*i.e.,* incorrect application decisions) for application A. In our prototype, ChameleonAPI uses the new loss function to re-train an off-the-shelf pre-trained DNN, a common practice to save training time (see §5.5 for quantification). The DNN re-training uses an application-specific dataset sampled from the dataset used by the pre-trained generic DNN (see Table 5.2 and §5.4), so that each target/non-target class is selected by ground-truth decisions of the same number of inputs.

Finally, ChameleonAPI backend maintains a set of DNN models, each customized for an application and keyed by the application ID. When application A invokes an ML API at run time, the ChameleonAPI backend will use the application ID associated with the API query to identify the DNN model customized for A, run the DNN on the input, and return the inference result of the selected model to the application.

Note that, ChameleonAPI can also be used to customize ML models that run locally behind the ML APIs, instead of those in the cloud through ML service providers. In this case, developers run the ChameleonAPI parser on their application and save the parser's result into a local file. This local file will then be consumed to help re-train an off-the-shelf DNN into a customized DNN to serve the application.

3. In many MLaaS offerings Google [2022], Amazon [2022a], a connection between the application and the MLaaS backend is commonly created before the application issues any queries. Existing MLaaS already allows applications to specify the application ID via the connection between the application and backend.

5.4 Implementation

Extractor of decision-process summary: The current prototype of ChameleonAPI is implemented for Python applications that use Google or Amazon ML APIs. It takes as input the application source code and returns as output the decision-process summary in the JSON format. It uses NICE symbolic execution engine Irlbeck et al. [2015] and CVC5 constraint solver Barbosa et al. [2022] to identify target classes, and uses Python static analysis framework Pyan Marby and Yonskai and Jedi Halter to identify the decision type and the matching order. Particularly, it identifies the object that is iterated through by a `for`-loop through the `iter` expression in each for-loop header, which is used to distinguish Multi-Choice and Multi-Select decisions and the matching order.

ML re-training: The re-training module is implemented in PyTorch v1.10 and CUDA 11.1. It uses a decision-process summary to construct a new loss function (see §5.3.2), and then replaces the builtin loss function in Pytorch with the new loss function, and uses the common forward and backward propagation procedure to re-train an off-the-shelf pre-trained DNN model (explained next).

	Dataset	Generic model
Image Classification	OpenImages Kuznetsova et al. [2020]	TResNet-L Ben-Baruch et al. [2020]
Object Detection	COCO COCO [2017]	Faster-RCNN Ren et al. [2015]
Sentiment Analysis	Amazon review Keung et al. [2020]	BERT Devlin et al. [2018]
Text Classification	Yahoo Huangzhao [2018]	BERT Devlin et al. [2018]
Entity Recognition	conll2003 Tjong Kim Sang and De Meulder [2003]	BERT Devlin et al. [2018]

Table 5.2: *The ML APIs and datasets in evaluation.*

Generic models: Without access to the models and the training data used by commercial ML services, we use open-sourced pre-trained DNNs and their training datasets as a proxy, which are summarized in Table 5.2. These DNNs are trained on the "training" portion of their respective datasets. They are trained to achieve good accuracy over a wide range of labels, and we have confirmed that their accuracies in terms of application decisions are similar to the real ML APIs (§5.5.2).

Training data: We make sure that the labels included in these datasets cover the labels used in the decision processes of the applications in our study. An exception is text classification: to our best knowledge, there is no open-source dataset that covers the classes in Google's text classification API. Instead, we use the Yahoo Question topic classification dataset Huangzhao [2018], whose classes are similar to those used in the applications.

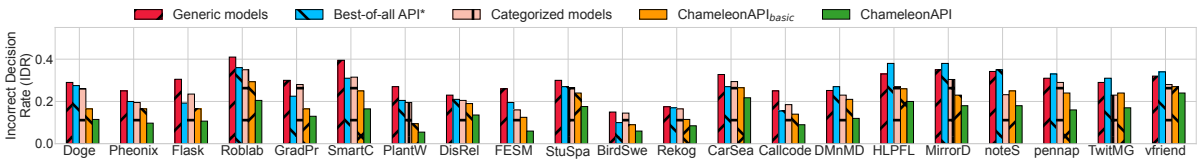
Instead of training DNNs on all training data, most of which do not match any target classes of an application, we create a downsampled training set for ChameleonAPI and ChameleonAPI$_{basic}$. For each application, we randomly sample (without replacement) its training data such that each target class and the non-target class (not matching any target class) is the correct decision for the same number of training inputs, which depending on applications, ranges from 12K to 40K. With such training set, ChameleonAPI$_{basic}$ will be equivalently implemented by training on the downsampled training set using the conventional loss function (*i.e.,* cross-entropy loss for classification tasks). Moreover, the downsampled training set significantly speedups DNN re-training (§5.5.2).

5.5 Evaluation

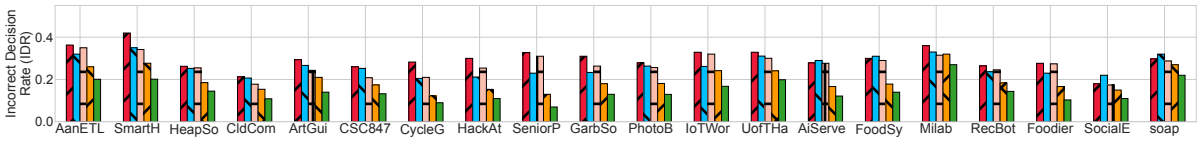
Our evaluation aims to answer following questions: How much can ChameleonAPI reduce incorrect application decisions? How long does it take ChameleonAPI to customize DNN models for applications? and Why does ChameleonAPI reduce incorrect application decisions where ChameleonAPI$_{basic}$ falls short?

5.5.1 Setup

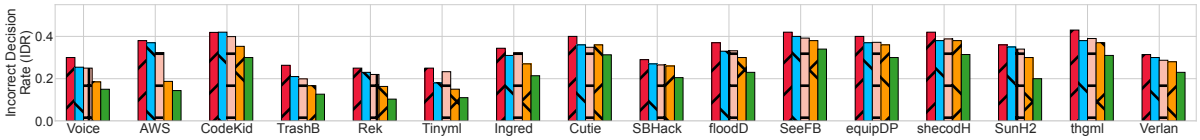
Applications: We have applied ChameleonAPI on all the 77 applications summarized in Table 5.1. Due to space constraints, our discussion below focuses on the 57 applications that involve three popular ML tasks, image-classification, object-detection, and text-classification, and omits the remaining 20 applications that involve sentiment analysis and entity recogni-



(a) Applications making True-False decisions.



(b) Applications making Multi-Choice decisions.



(c) Applications making Multi-Select decisions.

Figure 5.6: *ChameleonAPI reduces the incorrect decision rate (IDR) on the 57 applications that use Google's or Amazon's image-classification, text-classification, and object-detection APIs.*

tion. The results of the latter show similar trends of advantage from ChameleonAPI and are available in Appendix §C.4.

Metrics: For each scheme (explained shortly) and each application, we calculate the *incorrect decision rate* (*IDR*): the fraction of testing inputs whose application decisions do not match the correct application decisions (*i.e.,* decisions based on human-annotated ground truth).

Schemes: We compare the results of these schemes:

- *Various commercial ML APIs*: the results returned by ML APIs of three service providers (Google Google [2022], Amazon Amazon [2022a] and Microsoft Microsoft [2022a]).
- *Best-of-all API**: a hypothetical method that queries ML APIs from those three service providers on each input and picks the best output based on the classic definitions of accuracy: label-wise recall for classification tasks and mean-square-error of floating-point output for sentiment analysis. This serves as an idealized reference of recent

work Chen et al. [2020a, 2022b], which tries to select the best API output with high label-wise accuracy.

- *Generic models*: the open-sourced generic model based on which the next three schemes are re-trained. They serve as a reference without customization and achieve similar accuracy as commercial APIs. Their details are explained in Section 5.4.
- *Categorized models*: This scheme pre-trains a number of specialized models. Each specialized model replaces the last layer of the generic model so that it outputs the confidence scores for a smaller number of labels representing a common category (e.g., "dog", "animal", "person" and a few other labels represent the "natural object" category), and is fine tuned from the generic model accordingly. A simple parser checks which labels are used by an application. If all the labels belong to one category, the corresponding model specialized for this category is used to serve API calls from this application. If the labels belong to multiple categories, multiple specialized models will be used, which we will explain more later. We set up 35 categories for image classification and 7 categories for object detection based on the Wikidata knowledge graph Wikidata [2022], as well as 15 categories for text classification based on the inherent hierarchy in Google text-classification output. More details of how we have designed these categories are available in the Appendix §C.3.

  Note that, we have designed this scheme to represent a middle-point in the design space between the generic model and the ChameleonAPI approach: on one hand, this scheme offers some application customization, but not as much as ChameleonAPI (e.g., which labels belong to the same target class, what is the decision process, and what is the matching order used by the application are all ignored); on the other hand, this scheme requires a simpler parser compared to ChameleonAPI.
- $ChameleonAPI_{basic}$: the model is re-trained with ChameleonAPI's training data, which concentrates on labels used by the application, but with the conventional loss function.

Like Categorized models, this scheme only needs a simple parser that extracts which labels are used by the application, and does not make use of other application information that ChameleonAPI uses. Unlike Categorized models, this scheme prepares a customized model for each application, instead of relying on a small number of categorized models.

- *ChameleonAPI* (our solution): the model re-trained with our training data and loss function.

Testing data: For the same application, all schemes are tested against the same testing input set. The testing set of an application is randomly sampled from the "testing" portion of the dataset associated with the application's generic model (Table 5.2). We make sure that *no* testing input appears in the training data. Like the creation of training data of ChameleonAPI (§5.4), by default, we randomly sample the testing data such that each target class and the non-target class (not matching any target class) appear as the correct decision for the same number of testing inputs, which ranges from 1.2K to 4K. This is similar to the testing sets used in related work on ML API (*e.g.,* Kang et al. [2018, 2017], Wan et al. [2022], Chen et al. [2022b]). Such data downsampling is commonly used in ML ElRafey and Wojtusiak [2017], Liu et al. [2020]. Other than Figure 5.9, we will use this as the default testing dataset.

Hardware setting: We evaluate ChameleonAPI and other approaches on a GeForce RTX 3080 GPU, and an Intel(R) Xeon(R) E5-2667 v4 CPU, with 62GB memory.

5.5.2 Results

Overall gains: Measured by the average incorrect decision rate (IDR) across all applications, the most accurate scheme is ChameleonAPI, with an IDR of 0.16, and the least accurate scheme is Generic models, with an IDR of 0.31. In other words, ChameleonAPI

	True-False	Multi-Choice	Multi-Select
Google API	0.29	0.32	0.35
Microsoft API	0.30	0.33	0.32
Amazon API	0.31	0.33	0.36
Best-of-all API*	0.26	0.27	0.31
Generic models	0.29	0.30	0.34
Categorized models	0.24	0.27	0.31
ChameleonAPI$_{basic}$	0.19	0.22	0.27
ChameleonAPI	**0.13**	**0.16**	**0.21**

Table 5.3: *Average incorrect decision rate (IDR) among apps that make different types of decisions. The lower the better. The top half represents commercial APIs and their idealistic combinations; the bottom half represents open-source models.*

	Single-category	Multi-category
Generic models	0.32	0.28
Categorized models	0.28	0.27
ChameleonAPI$_{basic}$	0.24	0.18
ChameleonAPI	0.17	0.14

Table 5.4: *Average IDR among single-category and multi-category applications. The lower the better.*

successfully reduces the number of incorrect decisions of its baseline model by almost 50%. ChameleonAPI$_{basic}$ (0.22), Categorized models (0.28), and Best-of-all API* (0.28) have IDR rates in between.

The advantage of ChameleonAPI, and even ChameleonAPI$_{basic}$, over the other schemes is consistent across all three types of applications that make different types of decisions, as shown in Table 5.3. In fact, ChameleonAPI offers the highest accuracy by a clear margin for every single application in our evaluation, as shown in Figure 5.6.

To better compare the ChameleonAPI approach with Categorized models, we divide the 57 applications into two types: (1) 39 single-category applications — each application uses labels that belong to one category and hence can benefit from one specialized model in the Categorized models scheme; (2) 18 multi-category applications — each application uses labels that belong to multiple categories. For these applications, the Categorized models scheme

feeds the API input to multiple specialized models and combines these models' output to form the API output. As shown in Table 5.4, the Categorized models scheme does offer improvement from Generic models by considering which labels belong to an application's target classes, particularly for single-category applications. However, both ChameleonAPI and ChameleonAPI$_{basic}$ perform better than Categorized models for both single-category and multi-category applications—the per-application customization in ChameleonAPI and ChameleonAPI$_{basic}$ has paid off.

The above advantage of ChameleonAPI over ChameleonAPI$_{basic}$ and Categorized models shows that the static analysis used in ChameleonAPI to extract not only what labels are used by the application, but also which labels belong to the same target class, the decision type, and the matching order, as described in Section 5.3.3, is worthwhile.

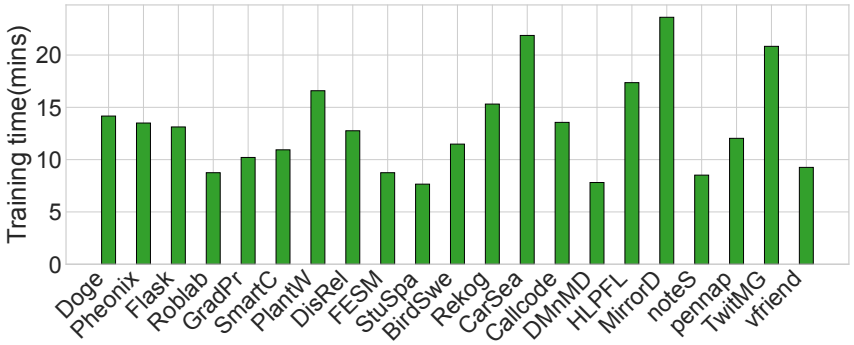


Figure 5.7: *Re-training time for applications in Figure 5.6(a).*

Cost of obtaining customized models: The customization effort of ChameleonAPI includes two parts (1) extracting the decision-process summary from application source code, and (2) re-training the ML model. The first part takes a few seconds: on an Intel(R) Xeon(R) E5-2667 v4 CPU machine, our parser extracts the decision-process summary from every benchmark application within 10 seconds.

The second part takes a few minutes, much faster than training a neural network from scratch. As shown in Figure 5.7, re-training DNNs for the 21 applications in Figure 5.6(a)

on a single RTX 3080 GPU takes 8 to 24 minutes. Focusing on a small portion of all possible labels (§5.2.4), ChameleonAPI fine-tunes pre-trained models using much less training data than the generic models and thus needs fewer iterations to converge.

Considering that a V100 GPU with similar processing GFLOPS as our RTX 3080 GPU only costs $2.38 per hour on Google Cloud Google [2023], re-training an ML model for one application costs less than $1.

Cost of hosting customized models: For cloud providers, ChameleonAPI would incur a higher hosting cost than traditional ML APIs by serving a customized DNN for every application instead of a generic DNN for all applications.

The extra cost includes more disk space to store customized neural network models. For example, each image-classification model in ChameleonAPI uses 115 MB of disk space. So, for n applications, $115 \cdot n$ MB of disk space may be needed to store ChameleonAPI customized models.

The extra cost also involves more GPU resources. A naive design of using one GPU to exclusively serve requests to one customized neural network model will likely lead to underutilization of GPU resources. To serve different applications' customized models on one GPU, we need to pay attention to memory working set and performance isolation issues. In our experiments on an RTX 3080 GPU, loading an image-classification model from CPU to GPU RAM takes 18 to 40 ms (inference itself takes 10 to 35 ms with a batch size of 1). Fortunately, modern GPU has sufficiently large RAMs to host several requests to different customized models simultaneously: in our experiments, the peak memory consumption of one inference request is less than 2GB. Furthermore, the majority of the model inference memory consumption comes from intermediate states, instead of the model itself. Consequently, the memory consumption of multiple inference requests on different models is similar to that on the same model.

Of course, ChameleonAPI can take advantage of recent proposals to improve GPU shar-

ing Zhang et al. [2023b], Corporation [2023], Yu and Chowdhury [2019], Padmanabhan et al. [2023] as well as to reduce the footprint of serving multiple DNNs Jiang et al. [2018]. These techniques could be advantageously employed by ChameleonAPI to determine the optimal degree of sharing among customized DNNs, and we leave them to future work.

Finally, there is also the extra cost of needing more complex software to manage the DNN serving. For instance, ChameleonAPI needs to dynamically route each request to a GPU that serves the DNN of the application (see §5.3.4).

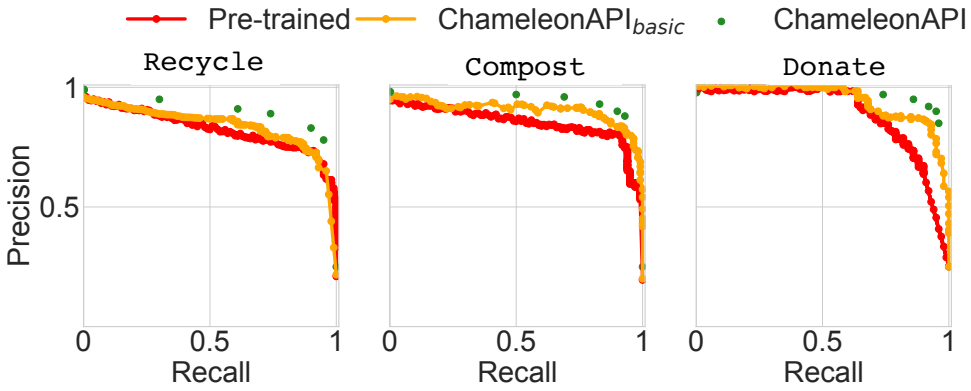


Figure 5.8: *Precision-recall trade-off for* `HeapsortCypher`.

Precision-recall tradeoffs: Traditionally, for a trained ML model, it is common to vary the confidence-score thresholds in order to find the best precision-recall tradeoff of a trained model. Thus, it is important that ChameleonAPI also achieves better precision-recall tradeoffs. Figure 5.8 shows the precision-recall results in each target class of a particular application, by varying the detection threshold θ (defined in §5.3.2) of two baselines (real APIs are excluded, because we cannot change their thresholds and their IDR is not as low as ChameleonAPI$_{basic}$). ChameleonAPI's tradeoffs are better than both baselines (and we observe similar results in other applications). Note that since ChameleonAPI's loss function uses an assumed θ, we do not vary the θ when testing it; instead, we re-train five DNNs of ChameleonAPI, each with a different θ and test them with their respective thresholds.

Understanding the improvement: ChameleonAPI's unique advantage is that it factors

in the decision process of an application, including not only the target classes but also the decision type and the matching order. Next, we use two case studies to further reveal the underlying tradeoffs made by ChameleonAPI to achieve its improvement on application-decision accuracy.

First, ChameleonAPI reduces errors related to different target classes differently depending on their different roles in the application decision process. This effect is particularly striking in Multi-Choice applications with the matching order of App-Order, where the first target class is always matched against API output. Thus, when the correct target class is not the first one, falsely including a label that belongs to the first target class will more likely be a critical error than other mis-classifications, because it will block the match of other target classes. To illustrate this, we consider the Multi-Choice application of `Aander-ETL`. We increase the percentage of testing inputs whose correct action is the first target class or the last target class. Figure 5.9 shows that increasing the portion of inputs of the last target class (`Person`) generally leads to bigger gains of ChameleonAPI, whereas increasing the portion of the first target class (`Landmark`) does the opposite. This shows the application itself already has good tolerance to mis-classification on inputs that belong to the first target class, but not to mis-classification on the inputs that belong to later target classes, which is exactly where ChameleonAPI can help.

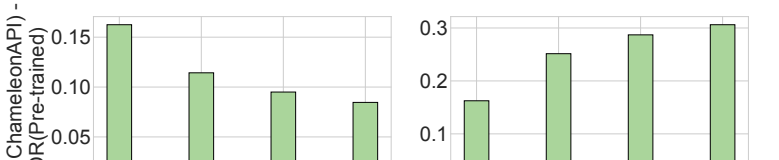


Figure 5.9: *How the accuracy advantage of ChameleonAPI changes with different input distribution.*

Second, recall from §5.3.2 that our loss function helps to minimize critical errors, even at the cost of missing labels that do not affect application decisions (*i.e.,* non-critical errors). To

show this, we define *label error rate* on an image as the fraction of the image's ground-truth labels that are missed by the DNN output (a label list). We consider `IoTWor` (explained in Table C.1), which similar to `Anander-ETL` makes Multi-Choice decisions with App-Order matching order. The average label missing rate of ChameleonAPI on our testing images is 0.21, which is slightly higher than ChameleonAPI$_{basic}$'s 0.18. This means ChameleonAPI makes more label-level mistakes than ChameleonAPI$_{basic}$. However, our IDR (0.17) is 44% lower than ChameleonAPI$_{basic}$, which means ChameleonAPI makes far fewer critical errors.

5.6 Conclusion

ML APIs are popular for its accessibility to application developers who do not have the expertise to design and train their own ML models. In this paper, we study how the generic ML models behind ML APIs might affect different applications' control-flow decisions in different ways, and how some ML API output errors may or may not be critical due to the application decision making logic. Guided by this study, we have designed ChameleonAPI that offers both the accuracy advantage of a custom ML model and the accessibility of the traditional ML API.

CHAPTER 6

RELATED WORK

This chapter surveys the work related to the three systems in this dissertation. Rather than discussing related work separately for each system, I organize it by theme, since the three systems draw on overlapping bodies of work. The first four sections cover the LLM side: techniques for handling long contexts (§6.1), systems that manage, reuse, and compress the KV cache (§6.2)—the line of work closest to this dissertation—general techniques for faster LLM serving (§6.3), and the fine-tuned models and multi-agent systems that motivate cross-model reuse (§6.4). The last section covers the non-LLM side: ML APIs and application-aware model serving (§6.5).

6.1 Handling Long Contexts

Longer LLM contexts: Recent efforts aim at enabling LLMs to process very long contexts Xu et al. [2024a]. The challenge is to fit the large attention matrices of longer contexts into limited GPU memory. This is enabled by offloading parts of the attention matrices Sheng et al. [2023], using external knowledge via KNN Wu et al. [2022b], retraining self-attention to only attend to top-k keys Bertsch et al. [2023], Ainslie et al. [2020], mapping long inputs to smaller latent spaces Hawthorne et al. [2022], and using local windowed, dilated, or sparse Ding et al. [2023a], Beltagy et al. [2020], Zaheer et al. [2021] attention to scale to inputs of $\sim$1 billion tokens. Longer contexts inflate the KV cache, and CacheGen addresses the resulting loading delay by fast remote loading of the KV cache.

Retrieval-augmented generation (RAG): RAG Borgeaud et al. [2022], Izacard et al. [2022], Izacard and Grave [2021], Rubin and Berant [2023a], Lewis et al. [2020], Wang et al. [2023c], Ram et al. [2023] focuses on retrieving documents relevant to the query, via vector-based Chen et al. [2017], Yang et al. [2019], Nie et al. [2019] or DNN-based Lewis et al. [2020],

Karpukhin et al. [2020], Xiong et al. [2021], Yang et al. [2022] similarity search, and feeding them as context to generate the answer. RAG is a fitting use case for the systems in this dissertation: many LLM inference platforms support feeding KV caches instead of text as the retrieved context Wolf et al. [2020a], Chase [2022], and some works define systematic ways to choose which KV cache to reuse Gim et al. [2023a]. Another approach caches a query's generated answers to reduce repetitive query costs Technology [2023], Martinez [2023]; while caching answers is useful, KV cache reuse provides a more generic way to incorporate context reuse and can generate better-quality answers.

6.2 KV Cache Management and Reuse

A common approach for faster inference without modifying the LLM is to cache the KV of previously used inputs within *one* LLM query Sheng et al. [2023], Zhang et al. [2023e], Liu et al. [2023d], Pope et al. [2022], Miao et al. [2023], Wolf et al. [2020b]. Going one step further, many systems support KV cache reuse *across* different requests to accelerate inference Kwon et al. [2023b], Agrawal et al. [2023], Gim et al. [2023b], Zheng et al. [2023b], Gim et al. [2024], Qin et al. [2025], Jin et al. [2025a], Gao et al. [2024b], Chen et al. [2024e, 2025a], Yu et al. [2025], Hu et al. [2024a], for example by building multi-tier caching systems that place the KV cache across storage tiers based on request recency or importance, or by offloading the cache out of GPU memory Hu et al. [2024a], Gao et al. [2024b], Jin et al. [2025a], Lee et al. [2024]. The measurement study in Chapter 2 provides real-world evidence for the reuse patterns that these systems, and this dissertation, build on.

KV cache compression: Since the KV cache of a long context is large, another line of work compresses it, along three main directions. The first *quantizes* the KV cache to lower numeric precision Kang et al. [2024], Liu et al. [2024f], Hooper et al. [2024]. The second *drops tokens*: selecting the most important text segments by their similarity to the user query Borgeaud et al. [2022], keeping only the tokens heavily attended to by the prompt (*i.e.,*

heavy-hitter tokens) Liu et al. [2023d], Zhang et al. [2023e], using hybrid policies that also keep nearby tokens Ge et al. [2023a], or applying query-aware compression with document reordering Jiang et al. [2023b], Ribar et al. [2023]; these methods need to know the query—otherwise they risk dropping potentially important tokens—and some require retraining the LLM to use contexts rewritten by gisting Mu et al. [2023] or auto-encoding Ge et al. [2023b]. The third applies *low-rank* decomposition, a form of general tensor compression Zhao et al. [2016], Oseledets [2011] that has also been used to compress gradient updates in DNN training Wang et al. [2023d], Agarwal et al. [2022, 2020]—although gradient compressors often exploit sparsity Chen et al. [2023e], Zehui et al. [2019], Child et al. [2019], which the KV cache is not known to exhibit in general. Whichever the direction, the compressed KV cache remains a *tensor*—of lower precision, fewer tokens, or lower rank—that the LLM consumes directly in GPU memory.

Yet, all of these systems rest on two implicit while strong assumptions. First, they keep the KV cache in its *tensor* format wherever it is stored—compressed or not—so fetching a cache over a slow link can take longer than recomputing it; CacheGen (Chapter 3) is the first system to abandon the tensor format altogether, encoding the KV cache into compact bitstreams for transmission, and it achieves better delay-quality trade-offs than token dropping, as shown in the evaluation of Chapter 3. Second, they assume the KV cache is reused by *the same model* that produced it; DroidSpeak (Chapter 4) removes this assumption by enabling reuse across different fine-tuned versions of a foundation model.

KV cache blending: Another line of research reduces the prefill delay when blending *non-prefix* KV caches from multiple contexts for the same model Yao et al. [2025], Gim et al. [2024]. Although CacheBlend also performs selective re-computation, DroidSpeak differs in both how and why the KV cache is re-computed: CacheBlend repairs the cross-attention between concatenated contexts for one model, whereas DroidSpeak repairs the layers most sensitive to reuse across two different models.

6.3 Faster LLM Serving

Most LLM systems research aims to speed up LLM training Rasley et al. [2020], Shoeybi et al. [2019] or make serving systems faster. On the serving side, prior work explores approximately parallelizing generation Leviathan et al. [2022], Miao et al. [2023], accelerating inference on edge devices Yi et al. [2023], quantizing LLM weights Aminabadi et al. [2022], reducing the memory I/O of GPU on-chip SRAM Dao et al. [2022], reducing the computation complexity of self-attention Roy et al. [2021], better scheduling strategies Wu et al. [2023a], Yu et al. [2022], Zhong et al. [2024a], Agrawal et al. [2024], Sheng et al. [2024b], Miao et al. [2024], Kwon et al. [2023a], Lin et al. [2024b], Patel et al. [2024], and better GPU memory utilization Kwon et al. [2023b]. Another line of work speeds up the serving of many fine-tuned models by hosting many LoRA adapters in memory at the same time and managing their memory efficiently Wu et al. [2024b], Sheng et al. [2024a], Chen et al. [2024d]. These techniques are orthogonal and complementary to the systems in this dissertation: they accelerate the computation itself, whereas CacheGen (Chapter 3) and DroidSpeak (Chapter 4) reduce TTFT by re-using KV cache to reduce repeated prefill computation in the first place.

Transmitting KV cache between GPUs: A related line of work optimizes the communication delay of transmitting the KV cache between GPUs, either by smart model-parallelism strategies Zhong et al. [2024a] or by implementing a new attention operation that transmits query vectors to the GPUs hosting smaller blocks of KV cache during decoding Lin et al. [2024a]. These systems assume high-bandwidth interconnects between GPUs; CacheGen instead targets the lower-bandwidth links between GPU servers and storage, where the transmission delay dominates.

Trading quality for speed with compact models: Another closely related line of work also trades speed for quality but uses more compact model architectures Liu et al. [2024e], Sarah et al. [2024], Xia et al. [2020]. However, to smoothly adapt the amount of computation,

they need to host multiple models of different sizes in GPU memory at the same time, which degrades the serving capacity of the system. DroidSpeak does not suffer from this, as it simply changes the number of recomputed layers.

6.4 Fine-Tuned Models and Multi-Agent Systems

Fine-tuning: Fine-tuning LLMs for specific tasks has gained importance, but it remains resource-intensive. Parameter-efficient fine-tuning methods, including LoRA and LISA Hu et al. [2022], Pan et al. [2024b], Yao et al. [2024b], Wu et al. [2024b], Sheng et al. [2024a], reduce the memory and computation needed for fine-tuning, making fine-tuned model variants ever more accessible—and ever more numerous in deployment.

Multi-agent systems: Multi-agent systems show promise in areas such as coding Holt et al. [2024], Chen et al. [2024a], Hong et al. [2024b], Team [2024], Islam et al. [2024], Huang et al. [2023], Qian et al. [2024], gaming Zhang et al. [2024c], Gong et al. [2024], Agashe et al. [2025], Zhang et al. [2024a], Chen et al. [2024b], Liu et al. [2024b], Mosquera et al. [2025], and social simulations Park et al. [2023a, 2022]. Using fine-tuned LLMs as agents further improves outcomes in question answering Chen et al. [2023a], tool learning Shen et al. [2024], and personalization Li et al. [2024d].

Together, the two trends mean that a single application increasingly spans multiple fine-tuned models that share the same context—the workload that motivates cross-model KV cache reuse. DroidSpeak (Chapter 4) reduces the repeated prefill delays in such systems by letting one model variant reuse the KV cache produced by another.

6.5 ML APIs and Application-Aware Model Serving

Optimizing storage and throughput of traditional DNN serving: Various techniques have been proposed to optimize the delay, throughput, and storage of ML models via model

distillation Mullapudi et al. [2019], Shen et al. [2017], Kim and Choi [2021b], pruning Han et al. [2016], or cascading Cao et al. [2021], Anderson et al. [2019]. This line of work explores a different design space than ChameleonAPI (Chapter 5): they design ML models with higher inference speed or smaller model size with minimum loss in accuracy, whereas ChameleonAPI re-trains existing ML models such that the rate of incorrect decisions of a given application is reduced. The closest line of prior work specializes DNN models for given queries Kang et al. [2018, 2017], Koudas et al. [2020], Cao et al. [2022, 2021], but it requires application developers to use a domain-specific language (*e.g.,* SQL Kang et al. [2018]) instead of general programming languages, and mostly focuses on reducing the DNN's size. In contrast, ChameleonAPI keeps both the ML API interface and the application source code intact while avoiding incorrect decisions.

Application-side optimization: Recent work also proposes to change the applications to better leverage existing ML APIs. One line of work Chen et al. [2020a, 2022b], Xie et al. [2022] invokes ML APIs from different service providers to achieve high accuracy within a query cost budget. Another aims to eliminate misuse of ML APIs in applications Wan et al. [2021, 2022], which requires changes to the application source code (*e.g.,* changing the API input preparation, or switching from an image-classification API to an object-detection API). These are complementary to ChameleonAPI, which customizes the model behind the API and does not require changes to the application's source code.

Measurement studies of ML services: For their rising popularity, ML-as-a-service platforms have attracted many measurement studies to understand their accuracy Chen et al. [2021a], performance Yao et al. [2017], robustness Hosseini et al. [2017], and fairness Koenecke et al. [2020]. However, they have so far not taken into account the ML applications that use the ML APIs, which differentiates them from the empirical study of application decision processes in Chapter 5. Previous work that studies ML applications Wan et al. [2021] did not look into the decision process or how ML API errors affect different applications differently.

Finally, a myriad of techniques have been studied to better manage and schedule GPU resources in ML training and serving systems (*e.g.,* Crankshaw et al. [2017], Romero et al. [2021], Crankshaw et al. [2020], Weng et al. [2022], Zhang et al. [2018], Holmes et al. [2019], Zhang et al. [2019], Kosaian et al. [2019], Shen et al. [2019b], Gujarati et al. [2020b], Lee et al. [2018], Mohan et al. [2022], Jang et al. [2019], Han et al. [2022], Choi et al. [2022]). They aim for different goals than ChameleonAPI, but these techniques can be used to help ChameleonAPI train and serve the application-specific ML models.

CHAPTER 7

LESSONS AND FUTURE WORK: TOWARD MODEL-NATIVE AI INFRASTRUCTURE

7.1 Summary of the Thesis

Nowadays, AI inference has become one of the most resource-consuming workloads in industry, and the interactions within inference systems have also changed with its rise. As described in Chapter 1, classic model serving largely fit a simple pattern: a model receives an input and returns a prediction to a human in a single turn and stateless manner. Modern AI systems expand this pattern in at least three ways: model-to-user interaction has become multi-turn, repeatedly engaging with long shared contexts; model-to-model interaction has become common, with multiple models share the same long contexts; and model-to-software interaction where application control flow branches on model outputs. In all three, the consumer of a model's output is increasingly another automated component, not a human reader. However, these components still communicate through human-readable media, including text, labels, and scores, which hurts in two ways. First, every consumer must re-process the producer's understanding from scratch, causing redundant computation: the same long context will be prefilled again and again. Second, human-readable inputs (*e.g.,* images, audios) may fail to capture the software context, discarding exactly the information the consuming AI model needs to predict correct outputs.

This dissertation argued that, in these settings, the right media for interacting with AI models are *model-native states*: internal or directly model-digestible states that incorporate a model's understanding of an input more directly than human-readable data. By storing, moving, compressing, translating, and specializing these states, AI systems can reduce redundant computation and communication while preserving or even improving downstream quality. I supported this thesis through a measurement-driven motivation study and three

systems.

Chapter 2 motivated the problem by showing that one concrete form of model-native state, KV cache in LLM inference, has already outgrown its traditional home – GPU memory. The KV cache was long treated as an ephemeral tensor that lives in GPU memory for the duration of a single request and is discarded afterward. Using telemetry from real enterprise deployments of LMCache Liu et al. [2025], Authors [2024], I showed that this single-request view no longer holds: the KV cache produced by one query is increasingly reused by other queries and users, with reuse per token growing week over week; the reusable cache quickly exceeds GPU memory; and a cache is often reused long after it is generated, such as days after it is initially created. In short, the KV cache must be stored *somewhere outside* GPU memory, in CPU DRAM, local disk, or remote storage, in order for it to be efficiently re-used by other requests. However, storing in lower-tier storage devices also comes with a catch: KV cache can easily grow to tens of gigabytes in size, so loading KV cache from lower-tier, such as local SSD, can be even slower than re-computing the KV cache from scratch. Unless loading is made substantially faster, the "shortcut" of reusing the cache loses the race against the very computation it is meant to skip.

Chapter 3 addressed this *loading delay* challenge with CacheGen. The key observation is that LLM inference requires the KV cache to be in its tensor format, but network transmission does not. CacheGen therefore encodes the KV cache into compact *bitstreams*, using delta encoding across nearby tokens, layer-wise quantization that spends more bits on more sensitive layers, and smart arithmetic coding; it then streams the bitstreams in a way that adapts to changes in available bandwidth on token-chunk basis, and fall back to text when the bandwidth is too low. Our evaluation shows that loading a stored KV cache becomes consistently faster than recomputing it, reducing the time-to-first-token by 3.1–4.7× compared to loading the text context and re-compute, with less than 2% drop in response quality.

CacheGen tackles the system challenges in the multi-turn model-to-user interaction: the

same context shared by multiple rounds is persisted in the format of KV cache, so later rounds do not need to re-process that context again and again. The next challenge is that in model-to-model interaction, multiple LLMs often need to read the same piece of shared context. For example, agents of different roles that process the same long context. The natural next question is whether the KV cache produced by one LLM can be reused by another. Chapter 4 addressed this *interoperability* challenge with DroidSpeak. Because the KV cache is a model-specific encoding of the input, the cache produced by one LLM cannot simply be fed to another: naively sharing it across models degrades generation quality by up to 60%. However, an empirical study across eight model pairs revealed that only a small subset of layers, often around 10%, are sensitive to the KV cache difference between two models, and the identities of these *critical layers* are largely consistent across inputs for the same model pair. DroidSpeak therefore profiles the critical layers offline, and at serving time re-computes only those layers while reusing the rest, reducing prefill latency by up to 3.1× and improving throughput by up to 4× with negligible quality drop.

Finally, in chapter 5, I present the last piece of my thesis work which addressed the accuracy challenge in model-to-software interaction, and in doing so showed that model-native state extends beyond the KV cache. In ML API applications, the challenge is that how the AI model is trained and optimized for doing (*i.e.,* labeling images) is often not what the software applications want the model to perform (*i.e.,* going to the right decision branch). This mismatch may lead to unexpectedly low software accuracy. To address this challenge, ChameleonAPI extracts how an application's decision logic consumes model outputs and compiles that software context into an application-specific, differentiable loss function. The resulting model internalizes which prediction errors matter to the application, a model-native representation of the software context, and reduces incorrect application decisions without changing the API of how developers call the AI model.

In short, the motivation study and the three systems together cover the full journey of

model-native state through an AI system: why it should live outside the GPU (Chapter 2), how it moves (Chapter 3), how it crosses model boundaries (Chapter 4), and how it absorbs the context of the software that consumes it (Chapter 5). Each of the central challenges identified in Section 1.6, including size, interoperability, and specialization, is answered by a concrete, evaluated system. The rest of this chapter outlines the lessons of this work (§7.2) and outlines the research agenda it opens (§7.3).

7.2 Lessons from This Dissertation

Beyond the individual systems, this dissertation suggests two broader lessons. They correspond to the two halves of the thesis: how AI inference systems should support model-native state, and how models should be shaped by the software that consumes their outputs with the help of model-native state.

Lesson 1: Model-native state is not a black box, nor merely another "cache": A tempting view is to treat the KV cache the way classic systems treat any cache: an opaque blob whose only useful property is that a hit is faster than a miss. Under this view, the serving system's job ends at storing the bytes and looking them up Kwon et al. [2023b], Zheng et al. [2023b], Yu et al. [2025], Gao et al. [2024a], Authors [2024], or performing higher-level request scheduling based on where or which tier KV cache sits in Qin et al. [2025], Hu et al. [2024a]. Yet, this view leaves most of the opportunity on the table. In my view, KV cache is the model's *encoding* of the input context, and like any other encoding, it has internal structure. The recurring methodology of this dissertation is to study that encoding in detail, token by token and layer by layer, and to let the patterns found there guide the system design.

Studying the KV cache *token by token* and *layer by layer* revealed the structure that CacheGen exploits. Within the same layer and channel, tokens in closer positions have more similar K/V values than tokens far apart, so the deltas between nearby tokens concentrate

around zero and can be encoded into far fewer bits than the original values. Moreover, the tensor format itself is only required at inference time, not during transmission, which is what frees the system to encode the cache into compact bitstreams in the first place. Studying the KV cache *layer by layer* revealed the second pattern. In CacheGen, applying the same quantization error to different layers affects response quality very differently, with the shallower layers far more sensitive to loss; this motivates keeping more bits on the layers that need them. In DroidSpeak, the key observation is also made by studying the KV cache *layer by layer*. Specifically, we found that the differences between two fine-tuned LLMs' KV cache differ greatly by layer. Only a small subset of layers, which are referred to as *critical layers*, are sensitive to the difference between the two LLM's KV cache. This drives our system design of only re-computes the KV cache for the critical layers while re-using other layers' KV cache as much as possible. None of these designs would follow from treating the cache as an opaque blob; each fell out of measuring the encoding first.

Lesson 2: Models should be aligned with their downstream tasks: When a model's output is consumed by software rather than read by a human, a generic training objective leaves quality on the table: it treats all errors alike, while the application does not. ChameleonAPI was an early demonstration of this principle. By parsing how an application's decision logic consumes model outputs and compiling that software context into an application-specific, differentiable loss, the model internalizes which errors matter, reducing incorrect application decisions without changing the external API. What was a contrarian position then has since become mainstream practice: today's models are routinely aligned with their downstream tasks via post-training, *e.g.,* coding models post-trained with reinforcement learning in agentic software-engineering environments, where the reward is derived from how the surrounding software consumes the output—whether the generated patch applies, passes the repository's tests, or matches how real developers resolved the issue Gehring et al. [2024], Pan et al. [2024a], Wei et al. [2025]—rather than from generic next-token accu-

racy alone. The general principle is that the alignment target should be the consumer's task, and a model-digestible encoding of the consumer's context, such as a differentiable objective, is the mechanism for expressing it.

7.3 Future Work

The systems in this dissertation treat model-native state as an optimization inside one serving deployment. Looking ahead, I believe the more consequential question is: *what happens when model-native state becomes the default medium through which networks of models and agents communicate?* I refer to this emerging setting as the *internet of agents*: agents built on different models, operated by different parties, exchanging model-native states rather than only human-readable text. Two layers of that stack are yet to be built: how model-native state is *distributed* between agents (§7.3.1), and how it is *analyzed* (§7.3.2).

7.3.1 Data Distribution for the Internet of Agents

Agentic workloads are quickly becoming networks of models: pipelines, planner–worker hierarchies, and agents that call other agents as services. Today, these agents exchange human-readable data, such as text prompts and tool outputs, so every agent (hop) pays the full prefill cost on largely the same context – the redundancy this dissertation identified within a single deployment, multiplied across every edge of the agent network. The natural endpoint is for the payload itself to become model-native state: an agent that has already processed a context ships its (compressed) KV cache to the next agent instead of re-sending text. The fundamental building blocks exist: CacheGen shows that KV cache can be encoded and streamed under varying bandwidth, and DroidSpeak shows that agents powered by fine-tuned models of the same architecture can be bridged, but the *distribution fabric* in between, a content delivery system whose clients are agents rather than humans, is yet to be built.

Building such a fabric first requires rethinking what "quality of experience" means: clas-

sical content delivery is engineered around human perception, including startup delay, re-buffering rates, perceptual quality, while what an agent experiences during content delivery is yet to be defined and measured. Beyond metrics, the delivery system itself breaks in two instructive ways. The first is whether replication is justified at all. CDNs replicate hot content to edge servers because user interest exhibits strong *regional locality*; whether model-native states exhibit similar locality, *i.e.,* do agents in the same region more likely reuse the same contexts, is yet to be studied. Also, unlike a CDN object, the *value* of a placed cache depends on which models run nearby. The second is the direction of traffic. Traditional CDN traffic is dominated by user-side *reads* from edge servers, but agents also *update* the states they read: each turn of an agentic interaction extends the shared context, appending to and revising the KV cache stored at the edge. The fabric must therefore support efficient in-place updates and replica consistency, not just serve static objects. Once model-native states accumulate in such a network, they also become a data asset worth *mining*: the subject of the next future direction.

7.3.2 Analytics for Model-Native State

The systems in this dissertation make individual decisions well—how much to compress each layer, which layers to re-compute—but each relies on hand-designed heuristics and offline profiling. At fleet scale, these decisions should instead be driven by the measured behavior of the state itself: the analytics layer that classical data systems have long had, but model-native state still lacks. Lesson 1 argued that the leverage comes from studying the encoding; an analytics system makes that study continuous, comprehensive and automatic.

Such a system must answer three questions. *What data to collect*: such as the lifecycle data identified in Chapter 2; but the state is prohibitively large, so the telemetry must record compact sketches of tensors rather than raw dumps. *How to store it*: analysis wants queryable access across many caches at once, unlike storage for reuse, and the retention and

indexing policies of such a "warehouse" remain an open design space. *What to analyze*: does an agent pay more attention to certain parts of the state? How does the perception of the same input differ across agents built on different models? Answering these requires analytics primitives that operate on tensors the way traditional big data analytics operates on strings or videos.

7.4 Closing Vision

Every maturing computing stack has eventually promoted its intermediate artifacts into first-class managed objects. Operating systems came to manage memory pages, not just programs. Databases came to manage indexes and materialized views, not just queries. The web came to manage cached content, not just servers. AI infrastructure today manages models, prompts, and outputs—but not yet the internal states through which AI systems actually communicate.

This dissertation argued that model-native state deserves the same promotion. The argument was motivated by evidence from real deployments —the KV cache already outgrows GPU memory and must live somewhere else (Chapter 2)—and demonstrated through three systems: model-native state can be compressed and streamed like media (CacheGen, Chapter 3), translated and repaired like a protocol (DroidSpeak, Chapter 4), and specialized to its consumer like an interface (ChameleonAPI, Chapter 5).

The stakes of this promotion are growing. As AI shifts from isolated models answering humans to networks of models and software working together, the dominant cost shifts from training to inference, and within inference, from computation to *re*-computation. Managing model-native state is how the infrastructure eliminates that redundancy. It is the difference between an internet of agents that re-reads everything at every hop, and one that remembers.

In short, future AI infrastructure should manage not only what models say, but what they know while saying it.

APPENDIX A

CACHEGEN APPENDIX

A.1 Text Output Examples of CacheGen

Figure A.1 visualizes an example from the LongChat dataset Li* et al. [2023] used in §3.7.2. The context fed into the LLM is a long, multi-round conversation history between the LLM and the user. An abridged context is shown in the upper box, where the first topic is about the role of art in society. The prompt to the LLM asks "What is the first topic we discussed? " CacheGen correctly generates the answer, whereas the default quantization baseline, which has a similar compressed KV cache size as CacheGen, generates the wrong answer.

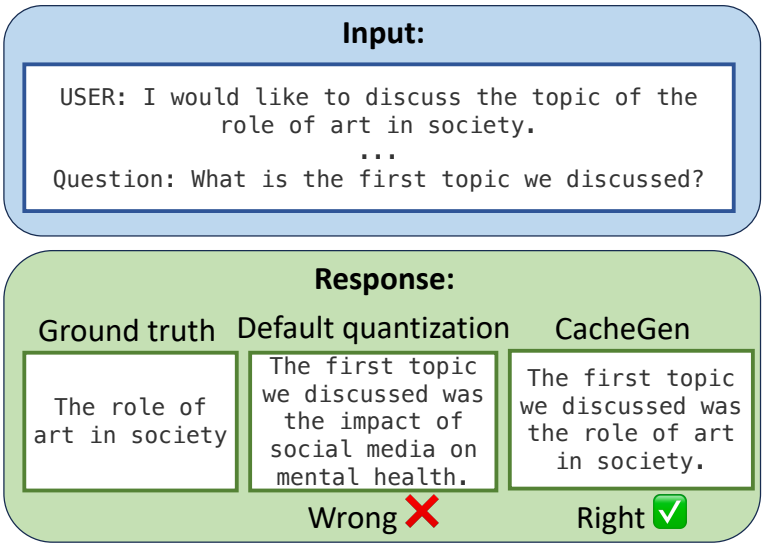


Figure A.1: *An example of CacheGen's output on the LongChat dataset with LongChat-7b-16k model.*

A.2 CacheGen vs. more intrusive methods

So far, all methods we have evaluated, including CacheGen, do not modify the LLM or context. As a complement, Figure A.2 tests CacheGen against recent methods that *change* the context or LLM.

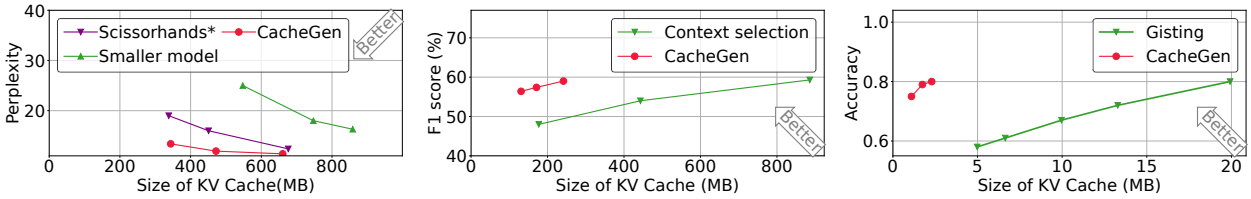


Figure A.2: *Comparing CacheGen and more intrusive methods, including smaller models, token dropping (left), context selection (middle), and gisting (right).*

- *Smaller models:* Replacing the LLM with *smaller models* may speed up the computation. Figure A.2a replaces the Llama-7B model with a smaller Llama-3B and applies different quantization levels.

- *Token selection:* Figure A.2b uses Scissorhands as an example of *removing tokens* with low self-attention scores from the LLM input Liu et al. [2023c]. Since the self-attention scores are only available during the actual generation, it cannot reduce TTFT, but we make an effort to create an idealized version of Scissorhands (Scissorhands*) by running the self-attention offline to determine which tokens to drop and provide this information to Scissorhands* online.

- *Gisting* Finally, we test Gisting as an example of a more advanced method that shortens contexts into gist tokens and changes the LLM to accept the gist tokens Mu et al. [2023]. In Figure A.2c, we test the pre-trained gisting model, which is based on Llama-7B. The gisting model retrains the LLM's attention model in order to run inference on a compressed version of the input prompts. Since the gisting model can compress arbitrary long contexts into *one token*, we vary the compression ratio of the gisting model to obtain a trade-off in size and accuracy. This is done by adapting the fraction of input tokens that are compressed into one token. We apply CacheGen on the original Llama-7B model on the PIQA Bisk et al. [2019] dataset, which is one of the most popular question-answering datasets. We did not apply CacheGen on other datasets in our evaluation because the public pre-trained gisting model can only take up to 512 tokens,

and truncating the dataset into smaller will not be able to preserve the information in the context.

We can see that CacheGen outperforms these baselines, reducing TTFT or KV cache size while achieving similar or better LLM's performance on the respective tasks. In particular, CacheGen is faster than smaller models (which are slowed down by transformer operations), and can reduce KV cache better than context selection or gisting because it compresses the KV features to more compact bitstream representations. We want to stress that even though CacheGen is compared head-to-head with these methods, it makes no assumption about the context and the model, so one can combine CacheGen with these methods to potentially further improve the performance.

A.3 CacheGen System Settings

A.3.1 KV Streamer Adaptation Logic

We present the pseudo-code for the KV streamer logic that adapts to bandwidth here.

Algorithm 1 CacheGen Streaming Adapter Logic

```
chunks_to_send ← context chunks
while chunks_to_send ≠ empty do
  get chunk_data
  throughput ← network throughput
  remaining_time ← SLO − time_elapsed
  if time_recompute ≤ remaining_time then
    cur_chunk ← text of chunk_data
  else
    level ← max(level|size(chunks_to_send, level) ÷ throughput ≤ remaining_time
    cur_chunk ← encode(chunk_data, level)
  end if
  send cur_chunk
  chunks_to_send ← chunks_to_send \ chunk_data
end while
```

A.3.2 Default Encoding Level

By default, CacheGen encoding is done with the following parameters: we partition the layers in the LLM into three groups with equal distance, and set quantization bins to be 0.5, 1, 1.5 respectively.

A.4 CacheGen's improvement under various workloads

Figure A.3 shows CacheGen's improvement over the best baseline (between quantization and text context) over a complete space of workloads characterized along the two dimensions of GPU available cycles (i.e., $1/n$ with n being the number of concurrent requests) and available bandwidth (in log scale). Figure 3.10 and Figure 3.11 can be seen as horizontal/vertical cross-sections of this figure.

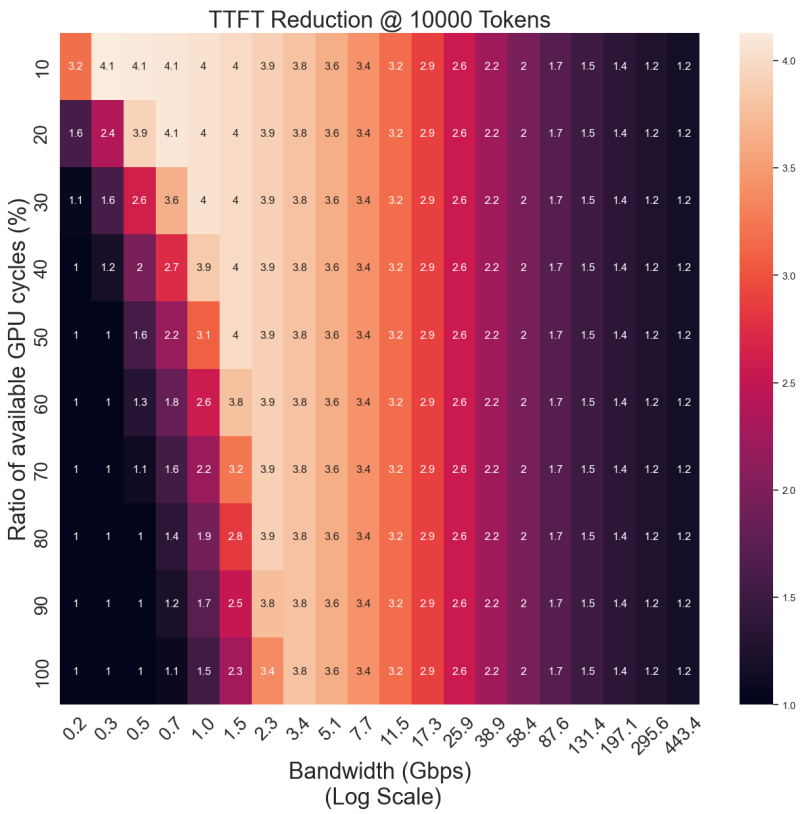


Figure A.3: *Heatmap showing CacheGen's improvement over the best baseline over a complete space of workloads. Brighter cells means more TTFT reduction.*

A.5 Cost of storing KV cache

Our main focus in this paper is to reduce TTFT to achieve service SLO with minimal impact on the generation quality of LLM. However, context loading systems, especially CacheGen, could be an economical choice for LLM service providers as well. For example, one piece of 8.5K-token context in Llama-13B takes roughly 5GB to store different versions compressed with CacheGen. It costs \$0.05 per month to store this data on AWS sto [2024]. On the other hand, recomputing the KV cache from text costs at least \$0.00085 (input only) every time tog [2024], any [2024], ama [2024], rep [2024]. If there are more than 150 requests reusing this piece of context every month, CacheGen will also reduce the inference cost. The calculation here only serves as a rough estimation to highlight CacheGen's potential. We leave the design of such a context loading system targeting cost-saving to future work.

APPENDIX B

DROIDSPEAK APPENDIX

Receiver Model	Sender Model
evolcode-3.1-8b	toolace-3.1-8b
glue_sst2	conllpp
mistral-blitz	mistral-24b
phi3.5-mini-instr-adapter-v2	phi-3.5-mini-instr-task15
llama-3-8b-sft-lora-ultrachat	fingpt-llama-3-8b
llama-3-70b-instruct	llama-3-70b
mistrallite	mistral-7b
llama-3.1-70b-instruct	llama-3.1-70b

Table B.1: *The model pairs used in our paper. For each pair of models, we use the datasets that meet the requirements listed in §4.3.1 (*i.e., *the receiver model has better quality than the sender model).*

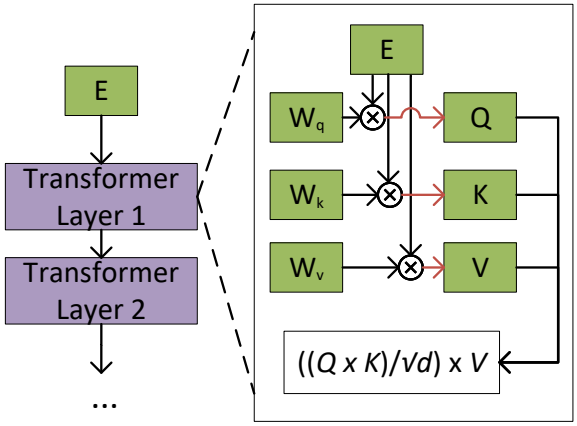


Figure B.1: *Illustration of the use of embedding (E), query (Q), key (K), and value (V) tensors in self-attention in transformer-based LLMs.*

B.1 Detail statistics of the datasets

Table B.2 shows the size of the datasets and the lengths of the contexts for these datasets.

Dataset	Task	Size	Med.	Std.	P95
hotpotQA Yang et al. [2018]	QA	300	10933	5160	18650
2wikimQA Ho et al. [2020]	QA	200	7466	3976	10705
multifieldQA_en Bai et al. [2024]	QA	150	8084	3849	14680
multi_news Fabbri et al. [2019]	Summ.	200	7624	5923	17547
lcc Guo et al. [2023]	Coding	200	12562	9220	26066
repobench-p Liu et al. [2024c]	Coding	200	14285	8665	30376

Table B.2: *Size, context lengths, the tasks the dataset belongs to and evaluation metrics of datasets in the evaluation. Among the tasks, "QA" stands for question-answering, and "Summ." stands for summarization.*

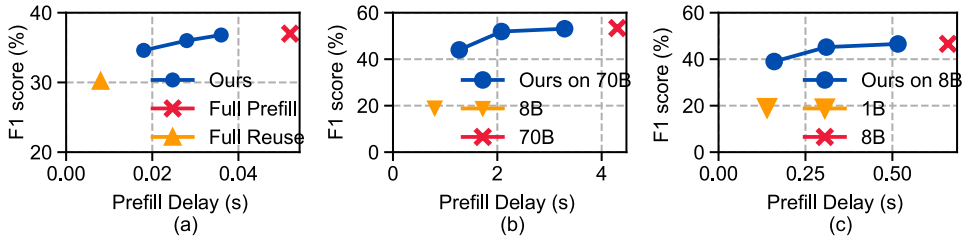


Figure B.2: *(a) Prefill delay and accuracy trade-off for MAmmoTH2 (fine-tuned on math reasoning tasks). (b) DroidSpeak applied on Llama-3.1-70B-Instruct has higher accuracy than Llama-3-8B-Instruct. (c) DroidSpeak applied on Ultrachat-8B has higher accuracy on TinyLlama-1.1B-Chat.*

B.2 More results

B.2.1 Case study of other tasks and other models

Math task: To demonstrate that the mechanisms of DroidSpeak apply to other types of datasets, we apply DroidSpeak on a model pair where the receiver model is fine-tuned on math reasoning, and test on a math reasoning task.

In Figure B.2(a), we run GSM8K Yue et al. [2024] dataset on MAmmoTH2 Yue et al. [2024]. We profile the re-computation configuration on the Math dataset Yue et al. [2024]. Note that the Pareto frontier obtained follows a very similar pattern compared to the LongBench models and dataset, demonstrating the wide applicability of DroidSpeak.

B.2.2 Comparison against a smaller model

Since DroidSpeak trades off minimal accuracy impact for latency, we compare using DroidSpeak on a larger model with a smaller model of the same architecture to show our superior performance in the quality and delay trade-off.

In Figure B.2(b), we compare DroidSpeak on `Llama-3.1-70B-Instruct` and `Llama-3.1-8B-Instruct` in the HotpotQA dataset, which is a smaller version of `Llama-3.1-70B-Instruct` and fine-tuned on the same dataset to enhance the base LLM's ability to follow instructions. As shown, `Llama-3.1-8B` achieves approximately a 4× reduction in prefill delay but suffers a reduction in F1 score of about half compared to the original F1 score of `Llama-3.1-70B-Instruct`.

In Figure B.2(c), we compare DroidSpeak on `Ultrachat-8B` and `TinyLlama-1.1B-Chat` on the 2wikimQA dataset, which is a smaller version of `Ultrachat-8B` while both fine-tuned on ultrachat_200k dataset Ding et al. [2023b]. Similary, we can see that compared with `Ultrachat-8B`, `TinyLlama-1.1B-Chat` reduces the prefill delay by 4.7×, while greatly reduces the F1 score. DroidSpeak, on the other hand, has much higher F1 score when applied to the `Ultrachat-8B` model.

One significant drawback of using a smaller model to achieve speedup is the overhead of switching between small and large models. For example, when additional resources become available, switching back to the larger model to improve serving quality incurs the overhead of loading the larger model back onto the GPU. In contrast, DroidSpeak can easily adapt to the available compute resources by adjusting the number of layers to be recomputed. This enables more possibilities for efficient scaling up or down on demand.

B.2.3 More results on profiling delay

In Figure 4.16, we run the same experiment as in §4.5.6 on the `evolcode` and `tool-8b` model pair. We observe similar pattern as in §4.5.6 that profiling the recomputation configuration at a coarser granularity reduces profiling cost while maintaining generation quality.

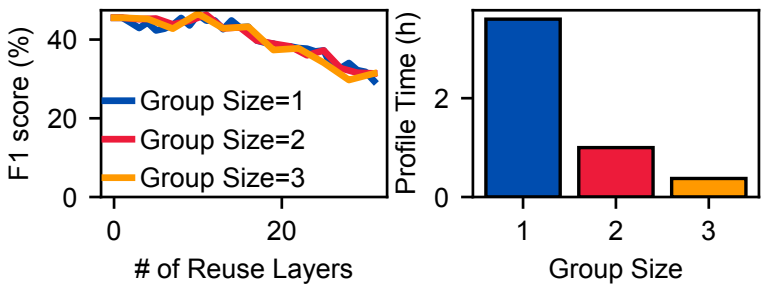


Figure B.3: *DroidSpeak's profiling delay on evolcode and tool-8b model pair, on the 2wikimQA dataset. Grouping the profiling layers at the granularity of multiple layers reduces profiling delay, without losing quality.*

B.3 Dynamic Re-computation Configuration Adaptation

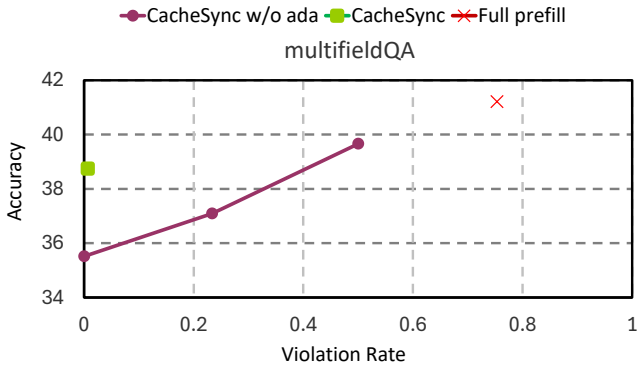


Figure B.4: *DroidSpeak reduces SLO violation rate over DroidSpeak without adaptation and full prefill. Plotted with* `ultrachat-8B` *and* `fingpt` *model pair.*

DroidSpeak initially checks the current workload intensity by monitoring the running and waiting requests in the vLLM engine Kwon et al. [2023a]. If there are requests that are waiting to be executed, it indicates that the current workload is high and triggers an increase in the reuse ratio. In this case, DroidSpeak chooses a re-computation configuration that has the lowest number of re-computed layers above an accuracy target. When there are no queuing requests in the system, DroidSpeak estimates prefill latency based on the number of tokens of each request. The parameters for estimation can be obtained through the profiling phase. DroidSpeak then selects the highest recomputation ratio satisfying the latency SLO for each request to maximize accuracy.

APPENDIX C

CHAMELEONAPI APPENDIX

C.1 Applications

Table C.1: The statistics of 77 applications in empirical study. (Multi-Choice-* refer to Multi-Choice (*-Order).)

Application name (Link to Github repo)	**Decision Type** (Matching Order)	**# of Target Classes** (# of labels per class)	**Branch Conditions** (Class lists or value ranges are separated by semicolons.)
Image Multi-Label Classification (Google `label_detection`, AWS `detect_labels`)			
2019-iot-ai-workshop	Multi-Choice-App	**2** (7, 2)	[Capuchin monkey, ...]; [Wildlife biologist, ...]
Aander-ETL	Multi-Choice-App	**3** (9, 6, 5)	[Landmark, Sculpture, ...]; [Building, Estate, ...]; [Human, ...]
ArtGuide	Multi-Choice-API	**2** (6, 3)	[Painting, Picture frame, ...]; [Building, Architecture, ...]
AWS_CloudComputing	Multi-Select	**2** (1, 1)	[Hot dog]; [Food]
DoorWatch	True-False	**1** (6)	[Clothing, Person, Human, Furniture, Child, Man]
AWSRekognition	Multi-Select	**2** (3, 3)	[Person, People, Human]; [Art, Drawing, Sketch]
GraduateProject	True-False	**1** (5)	[Orator, Professor, Projection Screen, ...]
Voice-Assistant	Multi-Select	**3** (5, 3, 1)	[Highway, Lane, ...]; [Car, ...]; [Classroom]
callforcode	True-False	**1** (5)	[Water, Waste, Bottle, Plastic, Pollution]

(To be continued)

Table C.1: The statistics of 77 applications in empirical study (Continued).

Application	**Decision Type** (Matching Order)	**# of Target Classes** (# of labels per class)	**Branch Conditions** (Class lists or value ranges are separated by semicolons.)
Car-Image-search	True-False	**1** (6)	[Sedan, Mini SUV, Coupe utility, Truck, Van, Convertible]
cloudComputing_project2	Multi-Choice-API	**3** (1, 3, 1)	[Person]; [Dog, Cat, Mammal]; [Flower]
CSC847_GAE_Proj2	Multi-Choice-API	**3** (2, 2, 1)	[Mammal, Livestock]; [Human, People]; [Flower]
cutiehack	Multi-Select	**2** (1, 3)	[Banana]; [Lemon, Citrus fruit, Apple]
CycleGAN-tensorflow_pixie	Multi-Choice-API	**3** (1, 6, 4)	[Food]; [Girl, Boy, Man, ...]; [Room, Living room, House, ...]
DisasterRelief	True-False	**1** (8)	[Hurricane, Flood, Tornado, Landslide, Earthquake, Volcano, ...]
Dogecoin_musk	True-False	**1** (4)	[Dog, Mammal, Carnivore, Wolf]
flaskAPI	True-False	**1** (3)	[Food, Recipe, Ingredient]
food-assessment-system	Multi-Choice-API	**5** (35, 22, 54, 4, 6)	[Dessert, ...]; [Grilling, ...]; [Strawberries, ...]; [Cigarette, ...]; ...
Foodier	Multi-Select	**2** (13, 1)	[Building, Logo, Menu, Person, Vehicle, People, ...]; [Food]
Hack-At-Home-II	Multi-Choice-API	**2** (3, 3)	[Food, Junk food, Plastic]; [Drinkware, Wood, Metal]

(To be continued)

Table C.1: The statistics of 77 applications in empirical study (Continued).

Application	**Decision Type** (Matching Order)	**# of Target Classes** (# of labels per class)	**Branch Conditions** (Class lists or value ranges are separated by semicolons.)
HeapSortCypher	Multi-Choice-API	**3** (8, 5, 11)	[Food, Food grain, ...]; [Clothing, Shirt, ...]; [Paper bag, ...]
IngredientPrediction	Multi-Select	**3** (1, 1, 1)	[Spaghetti]; [Bean]; [Naan]
FESMKMITL	True-False	**1** (1)	[Face]
milab	Multi-Choice-App	**3** (1, 1, 1)	[Sign]; [Nature]; [Car]
BirdSwe	Multi-Choice-API	**1** (5)	[Smoke, Bird, ...]
ai-server-proto	Multi-Choice-API	**3** (3, 14)	[Eye, Eyeball, Eyes]; [Landmark, Sculpture, Monument, ...]
Pheonix	True-False	**1** (1)	[Fire]
photo_book	Multi-Choice-API	**3** (10, 10, 2)	[Mammal, Bird, Insect, ...]; [Skin, Lip, ...]; [Flower, Plant]
Plant-Watcher	True-False	**1** (5)	[Plant, Flowerpot, Houseplant, Bonsai, Wood]
RecBot	Multi-Choice-App	**2** (11, 8)	[Tin, Paper, Magazine, Carton, ...]; [Food, Bread, Pizza, ...]
roblab-hslu	True-False	**1** (7)	[Raincoat, Coat, Jacket, T-shirt, Trousers, Jeans, Shorts]
senior-project	Multi-Choice-API	**3** (2, 3, 1)	[Landscape, Landmark]; [Self-portrait, Portrait, ...]; [Flower]

(To be continued)

Table C.1: The statistics of 77 applications in empirical study (Continued).

Application	**Decision Type** (Matching Order)	**# of Target Classes** (# of labels per class)	**Branch Conditions** (Class lists or value ranges are separated by semicolons.)
smart-can	True-False	**1** (9)	[Paper, Bottle, Plastic, Container, Tin can, Glass, ...]
smart-trash-bin	Multi-Choice-API	**2** (14, 5)	[Aviator sunglass, Beer glass, ...]; [Plastic arts, ...]
smartHamper	Multi-Choice-API	**3** (7, 4, 3)	[Shirt, T-shirt, ...]; [Trousers, Denim, ...]; [Brand, Text, ...]
StudySpaceAvailability	True-False	**1** (4)	[Hardware, Power Drill, Drill, Electronics]
The-Coding-Kid	Multi-Select	**6** (9, 6, 9, 3, 6, 3)	[Noodle, ...]; [Meat, ...]; [Produce, ...]; [Fruit, ...]; [Milk, ...]; ...
Tinyml	Multi-Select	**3** (4, 6, 5)	[Car, Truck, ...]; [Gun, Weapon Violence, ...]; [Cat, Dog, ...]
UofTHacksBackend	Multi-Choice-API	**4** (3, 3, 7, 4)	[T-shirt, ...]; [Outerwear, ...]; [Pants, ...]; [Footwear, ...]
garbage-sort	Multi-Choice-API	**2** (1, 20)	[Food];[Metal, ...];
Image Object Detection (Google `object_localization`)			
equipment-detection-poc	Multi-Select	**1** (1)	[Shoe]
flood_depths	Multi-Select	**1** (5)	[Car, Van, Truck, Boat, Toy vehicle]
SBHacks2021	Multi-Select	**1** (1)	[Person]

(To be continued)

Table C.1: The statistics of 77 applications in empirical study (Continued).

Application	**Decision Type** (Matching Order)	**# of Target Classes** (# of labels per class)	**Branch Conditions** (Class lists or value ranges are separated by semicolons.)
SeeFarBeyond	Multi-Select	**1** (2)	[Spoon, Coin]
shecodes-hack	Multi-Select	**1** (2)	[Dress, Top]
SunHacks2019	Multi-Select	**1** (3)	[Person, Chair, Table]
thgml	Multi-Select	**1** (7)	[Pizza, Food, Sushi, Baked goods, Snack, Cake, Dessert]
Verlan	Multi-Select	**1** (2)	[Dog, Animal]
Text Sentiment Classification (Google `sentiment_detection`)			
animal-analysis	Multi-Choice-API	**4** (1, 1, 1, 1)	[0.5, 1]; [0, 0.5]; [-0.5, 0]; [-1, -0.5]
calhacksv2	Multi-Choice-API	**6** (1, 1, 1, 1, 1, 1)	[0.5, 1]; [0.5, 1]; [0.1, 0.5]; [-0.1, 0.1]; [-0.5, -0.1]; [-1, -0.5]
carbon_hack_sentiment	Multi-Choice-API	**3** (1, 1, 1)	[0.3333, 1]; [-0.3333, 0.3333]; [-1, -0.3333]
FoodDelivery	Multi-Choice-API	**3** (1, 1, 1)	[0.6, 1]; [0.3, 0.6]; [-1, 0.3]
devfest	Multi-Choice-API	**4** (1, 1, 1, 1)	[0.6, 1]; [0.4, 0.6]; [0.2, 0.4]; [-1, 0.2]
EC601_twitter_keyword	Multi-Choice-API	**3** (1, 1, 1)	[0.25, 1]; [-0.25, 0.25]; [0.25, 1]

(To be continued)

Table C.1: The statistics of 77 applications in empirical study (Continued).

Application	**Decision Type** (Matching Order)	**# of Target Classes** (# of labels per class)	**Branch Conditions** (Class lists or value ranges are separated by semicolons.)
ElectionSentimentAnalysis	Multi-Choice-API	**3** (1, 1, 1)	[0.05, 1]; [0, 0.05]; [-1, 0]
Hapi	Multi-Choice-API	**2** (1, 1)	[-1, 0]; [0, 1]
JournalBot	Multi-Choice-API	**3** (1, 1, 1)	[0.5, 1]; [0, 0.5]; [-1, 0]
Mind_Reading_Journal	Multi-Choice-API	**4** (1, 1, 1, 1)	[0.15, 1]; [0.1, 0.15]; [-0.15, 0.1]; [-1, -0.15]
Sarcatchtic-MakeSPP19	Multi-Choice-API	**2** (1, 1)	[-0.5, 1]; [-1, -0.5]
stockmine	Multi-Choice-API	**2** (1, 1)	[-1, 0]; [0, 1]
Tone	Multi-Choice-API	**3** (1, 1, 1)	[-1, -0.5]; [-0.5, 0.5]; [0.5, 1]
UOttaHack_2019	Multi-Choice-API	**3** (1, 1, 1)	[0.25, 1]; [-0.25, 0.25]; [-1, -0.25]
Text Entity Detection (Google `entity_detection`)			
GeoScholar	True-False	**1** (1)	[LOC]
HackThe6ix	Multi-Choice-API	**7** (1, 1, 1, 1, 1, 1, 1)	[PERSON]; [LOC]; [ADD]; [NUM]; [DATE]; [PRICE]; [ORG]

(To be continued)

Table C.1: The statistics of 77 applications in empirical study (Continued).

Application	**Decision Type** (Matching Order)	**# of Target Classes** (# of labels per class)	**Branch Conditions** (Class lists or value ranges are separated by semicolons.)
Klassroom	Multi-Choice-API	**2** (2, 2)	[PERSON, PROPER]; [LOC, ORG]
newsChronicle	True-False	**1** (1)	[OTHER]
ocr-contratos	True-False	**1** (1)	[NUM]
uofthacks6	True-False	**1** (1)	[OTHER]
Text Topic Classification (Google `text_classify`)			
DMnMD	True-False	**1** (1)	[Health]
HLPFL	True-False	**1** (8)	[Public Safety, Law & Government, Emergency Services, News, ...]
MirrorDashboard	True-False	**1** (7)	[Jobs & Education, Law & Government, News, ...]
noteScript	True-False	**1** (1)	[Food]
pennapps_2019f	True-False	**1** (2)	[News/Politics, Investing]
soap	Multi-Choice-API	**2** (2, 2)	[Sensitive Subjects, ...]; [Discrimination & Identity Relations, ...]
SocialEyes	Multi-Choice-API	**2** (2, 1)	[people & society, sensitive subjects]; [adult]
Twitter_Mining_GAE	True-False	**1** (1)	[Sentitive]
vfriendo	True-False	**1** (1)	[Restaurants]

C.2 Loss function for other decision-process summaries

True-False:

$$L(y) = \overbrace{\text{Sigmoid}(\max_{l \in G_{\hat{c}}}(\mathbf{y}) - \theta)}^{\text{Penalize } \textbf{Type-1} \text{ Critical Errors}} + \overbrace{\text{Sigmoid}(\theta - \max_{l \in G_c}(\mathbf{y}))}^{\text{Penalize } \textbf{Type-1} \text{ Critical Errors}} \tag{C.1}$$

Multi-Select:

$$L(y) = \overbrace{\sum_{c \in \hat{T}} \text{Sigmoid}(\theta - \max_{l \in Gc} \mathbf{y}[l])}^{\text{Penalize } \textbf{Type-1} \text{ Critical Errors}} + \overbrace{\sum_{c \in \cup_c G_c \setminus \hat{T}} \text{Sigmoid}(\max_{l \in Gc} \mathbf{y}[l] - \theta)}^{\text{Penalize } \textbf{Type-3} \text{ Critical Errors}} \tag{C.2}$$

Multi-Choice API-order: Here we explain why this loss function captures the critical errors:

- A Type-1 error occurs, if (1) the correct target class is matched, thus at least one of its labels has a score above the confidence threshold ($\max_{l \in G_{\hat{c}}} \mathbf{y}[l] \geq \theta$), and (2) it is matched after the EOD because all of the labels belonging to the correct target class have scores below the maximum score of labels in the incorrect target classes.
- A Type-2 error occurs if the maximum score for labels in a correct target class falls below threshold θ, thus it is never matched (before or after EOD).
- A Type-3 error occurs if any labels belonging ($\max_{l \notin G_{\hat{c}}} \mathbf{y}[l]$) to incorrect target classes appears before labels in the correct target class.

$$
\begin{aligned}
L(\mathbf{y}) = & \overbrace{\text{Sigmoid}\left(\max_{l \in \cup_{c \neq \hat{c}} G_c} \mathbf{y}[l] - \max_{l \in G_{\hat{c}}} \mathbf{y}[l]\right)}^{\textbf{Type-1 Critical Errors}} \\
& + \overbrace{\text{Sigmoid}\left(\max_{l \in \cup_{c \neq \hat{c}} G_c} \mathbf{y}[l] - \theta\right)}^{\textbf{Type-2 Critical Errors}} \\
& + \overbrace{\sum_{c \neq \hat{c}} \text{Sigmoid}\left(\max_{l \in G_c} \mathbf{y}[l] - \max_{l \in G_{\hat{c}}} \mathbf{y}[l]\right)}^{\textbf{Type-3 Critical Errors}}
\end{aligned}
\tag{C.3}
$$

Value ranges: As for APIs that output a score $\mathbf{y}$ to describe the input, applications typically define several value ranges as target classes to make decisions, where the lower bound of the c^{th} target class is denoted as l_c and the upper bound of the c^{th} target class is denoted as u_c.

$$
\begin{aligned}
L(y_i) = & \overbrace{\text{Sigmoid}\left(\mathbf{y} - u_{\hat{c}}\right) + \text{Sigmoid}\left(l_{\hat{c}} - \mathbf{y}\right)}^{\textbf{Type-1 Critical Errors}} \\
& + \overbrace{\sum_{c \neq \hat{c}} \text{Sigmoid}(u_c - \mathbf{y}) + \text{Sigmoid}(\mathbf{y} - l_c)}^{\textbf{Type-3 Critical Errors}}
\end{aligned}
\tag{C.4}
$$

where $\hat{c}$ is the index of the correct target class. A Type-1 error occurs (*i.e.,* a correct target class is matched after EOD) when the output score $\mathbf{y}$ exceeds the upper bound of the ground-truth value range (u_c), or falls below the lower bound of the ground-truth value range (l_c). A Type-3 error occurs when the upper bound of an incorrect value range exceeds $\mathbf{y}$ *and* its lower bound falls below $\mathbf{y}$, leading it to be selected. Type-2 errors are absent in this application because all the target classes span the whole output range, thus a target class must be matched.

C.3 Setup of Categorized models

Here, we describe in detail how we construct the label categories to support the scheme of Categorized models, which is one of the schemes in comparison with ChameleonAPI (Section 5.5).

Image-classification Image-classification APIs typically contains many thousands of labels without providing their categorization or hierarchy. Therefore, we create categories leveraging the Wikidata knowledge graph Wikidata [2022], a widely used knowledge graph database that has been referred to during the creation of many popular ML training datasets Haller et al. [2022], Kuznetsova et al. [2020], Jund et al. [2017]. In this knowledge graph, each node is a named entity, covering *all* the labels used in popular image-classification APIs Google [2022], Amazon [2022a], Microsoft [2022b], and each edge represents a relationship between two entities (e.g., "subclass of", "different from", "said to be the same as").

Extracting "subclass of" edges in Wikidata knowledge graph, we get a directed acyclic graph (DAG) of label hierarchy. We believe it offers a principled foundation to create label categories based on two observations: (1) If an entity/node e is reachable from another entity/node e' through several subclass-of edges, e' is also covered by the *category* of e (e.g., entity "motor vehicle" is directly connected to "land vehicle", entity "land vehicle" is directly connected to "vehicle", so "motor vehicle" is also covered by the "vehicle" category); (2) The distance, measured in the number of subclass-of edges, between an entity/node and the DAG root indicates the specificity of the concept behind this node, with shorter distance representing coarser-grained categories. We will refer to a node that is k edges away from the root as a Level-k node.

Based on these observations, we formally define a set of categories C_k for all the image-classification labels L at a specificity level k as follows: C_k is the minimum set of Level-k nodes such that every label $l \in L$ is covered by at least one category node $c_k \in C_k$. We could

categorize all the applications into single-category and multi-category applications using any level of specificity settings. In this paper, we adopt Level-2 specificity setting, since the number of single-category applications drops a lot when moving from Level-2 to Level-3, indicating Level-3 categories may be too fine-grained.

Under Level-2, we set up 35 categorized models that cover all the image-classification labels. Six of them are used by applications in our benchmark, including *natural object*, *temporal entity*, *artificial entity*, *system*, *phenomenon*, and *continuant*. With this categorization, 27 of the 40 image-classification applications are single-category, and the rest 13 applications are multi-category.

Object-detection Similar as image-classification API, every object-detection API label also corresponds to an entity node in the Wikidata knowledge graph. Therefore, we use the same methodology and the same specificity Level-2 to define categories for object detection labels.

Seven categorized models are set up to cover all object-detection labels. The object-detection labels used by 8 object-detection benchmark applications belong to 3 categories: *natural object*, *artificial entity*, and *system*. Under this setting, there will be 7 single-category applications, and 1 multi-category applications.

Text-classification The Google text-classification API Google [2022] offers the hierarchy tree of all its labels. We simply follow their categorization and get 15 categories to covering all text-classification labels. Nine categories are used by applications in our benchmark, including *business & industrial*, *people & society*, *health*, *food & drink*, *jobs & education*, *news*, *sensitive subjects*, *adult*, and *law & government*. Under this categorization, there are 5 single-category text-classification applications, and 4 multi-category text-classification applications.

Other types of applications There are two other types of applications in our benchmark that are not suitable for designing pre-specialized models: sentiment analysis and named entity recognition.

For sentiment analysis API, the corresponding applications typically define several value-ranges and determine which range the API output (a sentiment score) falls into. Since the API output is a floating point number, there are infinite ways of defining value-ranges. Therefore, it is impracticable to create pre-categorized models.

For named entity recognition API, it only has 6 labels: *person, location, organization, number, date*, and *misc* Google [2022]. They are already high-level categories. There is no need to create pre-categorized models for each category.

C.4 Results of other applications

As mentioned in §5.5.1, the results of the 20 applications that involve sentiment analysis and entity recognition were not included in the evaluation section. Their results are shown in Figure C.1 and C.2. As we can see, the advantage of ChameleonAPI is consistent across these applications, similar to what we presented in§5.5. Note that, the scheme of Categorized models does not apply to applications that involve these two types of ML tasks, and hence is not included in Figure C.1 and C.2.

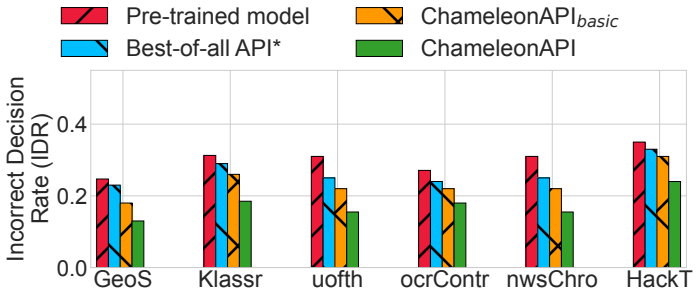


Figure C.1: Results on entity-recognition applications.

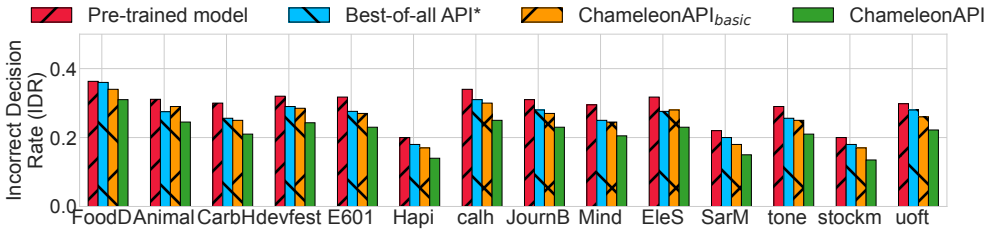


Figure C.2: Results on sentiment-analysis applications.

REFERENCES

2112.04426.pdf. https://arxiv.org/pdf/2112.04426.pdf. (Accessed on 09/21/2023).

Significant-gravitas/auto-gpt: An experimental open-source attempt to make gpt-4 fully autonomous. https://github.com/Significant-Gravitas/Auto-GPT. (Accessed on 09/21/2023).

[2304.03442] generative agents: Interactive simulacra of human behavior. https://arxiv.org/abs/2304.03442. (Accessed on 09/21/2023).

GGML - AI at the edge. https://ggml.ai/.

Bard - chat based ai tool from google, powered by palm 2. https://bard.google.com/. Accessed: September 21st, 2023.

Gpt-4 api general availability and deprecation of older models in the completions api. https://openai.com/blog/gpt-4-api-general-availability. (Accessed on 09/21/2023).

Perplexity in fixed length models. https://huggingface.co/docs/transformers/perplexity, a.

Huggingface Transformers. https://huggingface.co/docs/transformers/index, b.

langchain-ai/langchain:building applications with llms through composability. https://github.com/langchain-ai/langchain, a. (Accessed on 09/21/2023).

Store and reference chat history | langchain. https://python.langchain.com/docs/use_cases/question_answering/how_to/chat_vector_db, b. (Accessed on 09/21/2023).

[2302.13971] llama: Open and efficient foundation language models. https://arxiv.org/abs/2302.13971, a. (Accessed on 09/21/2023).

llama.cpp. https://github.com/ggerganov/llama.cpp/, b.

Applications of large language models - indata labs. https://indatalabs.com/blog/large-language-model-apps, a. (Accessed on 09/21/2023).

12 Practical Large Language Model (LLM) Applications - Techopedia. https://www.techopedia.com/12-practical-large-language-model-llm-applications, b. (Accessed on 09/21/2023).

7 top large language model use cases and applications. https://www.projectpro.io/article/large-language-model-use-cases-and-applications/887, c. (Accessed on 09/21/2023).

Real-world use cases for large language models (llms) | by cellstrat | medium. https://cellstrat.medium.com/real-world-use-cases-for-large-language-models-llms-d71c3a577bf2, d. (Accessed on 09/21/2023).

Anthropic \ introducing 100k context windows. https://www.anthropic.com/index/100k-context-windows. (Accessed on 09/21/2023).

Amazon Mechanical Turk. https://www.mturk.com/.

How latency affects user engagement. https://pusher.com/blog/how-latency-affects-user-engagement/, 2021. (Accessed on 09/21/2023).

How to train generative ai using your company's data. https://hbr.org/2023/07/how-to-train-generative-ai-using-your-companys-data, 2021. (Accessed on 09/21/2023).

Building rag-based llm applications for production. https://www.anyscale.com/blog/a-comprehensive-guide-for-building-rag-based-llm-applications-part-1, 2023. Accessed: 2024-01-25.

Best practices for deploying large language models (llms) in production. https://medium.com/@_aigeek/best-practices-for-deploying-large-language-models-llms-in-production-fdc5bf240d6a, 2023. (Accessed on 09/21/2023).

Llmperf leaderboard. https://github.com/ray-project/llmperf-leaderboard, 2023. Accessed: 2024-01-25.

pathwaycom/llmapp. https://github.com/pathwaycom/llm-app, 2024. Accessed: 2024-01-25.

Amazon bedrock pricing. https://aws.amazon.com/bedrock/pricing/, 2024. Accessed: 2024-01-25.

Anyscale pricing. https://docs.endpoints.anyscale.com/pricing, 2024. Accessed: 2024-01-25.

De-coded: Understanding context windows for transformer models. https://towardsdatascience.com/de-coded-understanding-context-windows-for-transformer-models-cd1baca6427e, 2024.

Chatgpt. https://chat.openai.com/gpts, 2024. Accessed: 2024-01-25.

The Twelfth International Conference on Learning Representations, 2024.

Docuease. https://docuease.com/, 2024. Accessed: 2024-01-25.

Rag-transform. https://huggingface.co/transformers/v4.3.0/model_doc/rag.html, 2024. Accessed: 2024-01-25.

Thompson reuters. https://www.thomsonreuters.com/en.html, 2024. Accessed: 2024-01-25.

Ollama - get up and running with large language models, 2024. URL https://ollama.com/. Accessed: 2024-11-19.

Perplexity. https://www.perplexity.ai/, 2024. Accessed: 2024-01-25.

Replicate pricing. https://replicate.com/pricing, 2024. Accessed: 2024-01-25.

Aws pricing examples. https://aws.amazon.com/s3/pricing/, 2024. Accessed: 2024-01-25.

together.pricing. https://www.together.ai/pricing, 2024. Accessed: 2024-01-25.

Aander-ETL. A smart album application. https://github.com/Grusinator/Aander-ETL.

Daniel Adiwardana, Minh-Thang Luong, David R. So, Jamie Hall, Noah Fiedel, Romal Thoppilan, Zi Yang, Apoorv Kulshreshtha, Gaurav Nemade, Yifeng Lu, and Quoc V. Le. Towards a Human-like Open-Domain Chatbot, 2020.

Megha Agarwal, Asfandyar Qureshi, Nikhil Sardana, Linden Li, Julian Quevedo, and Daya Khudia. Llm inference performance engineering: Best practices, October 2023. URL https://www.databricks.com/blog/llm-inference-performance-engineering-best-practices. Accessed: 2024-06-01.

Saurabh Agarwal, Hongyi Wang, Kangwook Lee, Shivaram Venkataraman, and Dimitris Papailiopoulos. Accordion: Adaptive gradient communication via critical learning regime identification. *arXiv preprint arXiv:2010.16248*, 2020.

Saurabh Agarwal, Hongyi Wang, Shivaram Venkataraman, and Dimitris Papailiopoulos. On the utility of gradient compression in distributed training systems. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 652–672, 2022. URL https://proceedings.mlsys.org/paper_files/paper/2022/file/773862fcc2e29f650d68960ba5bd1101-Paper.pdf.

Saaket Agashe, Yue Fan, Anthony Reyna, and Xin Eric Wang. LLM-coordination: Evaluating and analyzing multi-agent coordination abilities in large language models. In Luis Chiruzzo, Alan Ritter, and Lu Wang, editors, *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 8038–8057, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics. ISBN 979-8-89176-195-7. doi:10.18653/v1/2025.findings-naacl.448. URL https://aclanthology.org/2025.findings-naacl.448/.

Andrea Agostinelli, Timo I. Denk, Zalán Borsos, Jesse Engel, Mauro Verzetti, Antoine Caillon, Qingqing Huang, Aren Jansen, Adam Roberts, Marco Tagliasacchi, Matt Sharifi, Neil Zeghidour, and Christian Frank. MusicLM: Generating Music From Text, 2023.

Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills, 2023.

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming Throughput-Latency tradeoff in LLM inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/agrawal.

Toufique Ahmed and Premkumar Devanbu. Few-shot training llms for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023a. Association for Computing Machinery. ISBN 9781450394758. doi:10.1145/3551349.3559555. URL https://doi.org/10.1145/3551349.3559555.

Toufique Ahmed and Premkumar Devanbu. Few-shot training llms for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023b. Association for Computing Machinery. ISBN 9781450394758. doi:10.1145/3551349.3559555. URL https://doi.org/10.1145/3551349.3559555.

LangChain AI. Open deep research. https://github.com/langchain-ai/open_deep_research, 2025. GitHub repository, accessed September 15, 2025.

Monica (Butterfly Effect AI). Manus ai agent. https://manus.im, 2025. Accessed: 2025-04-23.

Together AI. Finetuning. https://docs.together.ai/docs/fine-tuning-overview, November 2023. Accessed: 2023-11-11.

ai-dynamo. Dynamo: A datacenter scale distributed inference serving framework. https://github.com/ai-dynamo/dynamo, 2025. GitHub repository, release v0.1.1, accessed 21 April 2025.

AI@Meta. Llama 3 model card. 2024. URL https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md.

Joshua Ainslie, Santiago Ontanon, Chris Alberti, Vaclav Cvicek, Zachary Fisher, Philip Pham, Anirudh Ravula, Sumit Sanghai, Qifan Wang, and Li Yang. ETC: Encoding Long and Structured Inputs in Transformers, 2020.

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. In *The 2023 Conference on Empirical Methods in Natural Language Processing*, 2023. URL https://openreview.net/forum?id=hmOwOZWzYE.

Amazon. Amazon artificial intelligence service. Online document https://aws.amazon.com/machine-learning/ai-services, 2022a.

Amazon. Amazon rekognition: detecting labels. Online document https://docs.aws.amazon.com/rekognition/latest/dg/labels.html, 2022b.

Amazon Web Services. Amazon ec2 p5 instances, 2024. URL https://aws.amazon.com/ec2/instance-types/p5/.

Amazon.com Inc. 2023 annual report. Annual report, Amazon.com Inc., 2023. URL https://s2.q4cdn.com/299287126/files/doc_financials/2024/ar/Amazon-com-Inc-2023-Annual-Report.pdf.

Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. DeepSpeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2022.

Sotiris Anagnostidis, Dario Pavllo, Luca Biggio, Lorenzo Noci, Aurelien Lucchi, and Thomas Hofmann. Dynamic Context Pruning for Efficient and Interpretable Autoregressive Transformers, 2023.

Michael R Anderson, Michael Cafarella, German Ros, and Thomas F Wenisch. Physical representation-based predicate optimization for a visual analytics database. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1466–1477. IEEE, 2019.

Anonymous. Chunkattention: Efficient attention on KV cache with chunking sharing and batching, 2024. URL https://openreview.net/forum?id=9k27IITeAZ.

Anyscale Team. Comparing llm performance: Introducing the open source leaderboard for llm apis, December 2023. URL https://www.anyscale.com/blog/comparing-llm-performance-introducing-the-open-source-leaderboard-for-llm. Accessed: 2024-06-01.

Anysphere Inc. Cursor: The ai code editor. https://www.cursor.com/, 2025. Accessed: 2025-04-23.

Samuel Arcadinho, David Oliveira Aparicio, and Mariana S. C. Almeida. Automated test generation to evaluate tool-augmented LLMs as conversational AI agents. In Dieuwke Hupkes, Verna Dankers, Khuyagbaatar Batsuren, Amirhossein Kazemnejad, Christos Christodoulopoulos, Mario Giulianelli, and Ryan Cotterell, editors, *Proceedings of the 2nd GenBench Workshop on Generalisation (Benchmarking) in NLP*, pages 54–68, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi:10.18653/v1/2024.genbench-1.4. URL https://aclanthology.org/2024.genbench-1.4/.

Muhammad Arslan, Saba Munawar, and Christophe Cruz. Sustainable digitalization of business with multi-agent rag and llm. *Procedia Computer Science*, 246:4722–4731, 2024.

Malika Aubakirova, Alex Atallah, Chris Clark, Justin Summerville, and Anjney Midha. State of AI 2025: A 100 trillion token LLM usage study. https://openrouter.ai/state-of-ai, December 2025. (Accessed on 07/16/2026).

AuthorName. Can chatgpt understand context and keep track of conversation history. https://www.quora.com/Can-ChatGPT-understand-context-and-keep-track-of-conversation-history, Year. Quora question.

LMCache Authors. Lmcache: A kv cache sharing layer for fast distributed llm serving. https://github.com/LMCache/LMCache, 2024.

Leif Azzopardi, Mark Girolami, and Keith van Risjbergen. Investigating the relationship between language model perplexity and ir precision-recall measures. In *Proceedings of the 26th Annual International ACM SIGIR Conference on Research and Development in Informaion Retrieval*, SIGIR '03, page 369–370, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581136463. doi:10.1145/860435.860505. URL https://doi.org/10.1145/860435.860505.

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. Longbench: A bilingual, multitask benchmark for long context understanding. *arXiv preprint arXiv:2308.14508*, 2023.

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. LongBench: A bilingual, multitask benchmark for long context understanding. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3119–3137, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi:10.18653/v1/2024.acl-long.172. URL https://aclanthology.org/2024.acl-long.172/.

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. Codeplan: Repository-level coding using llms and planning, 2023.

Raj Balakrishnan. Bring your own data to azure openai chat models. https://techcommunity.microsoft.com/t5/azure-architecture-blog/bring-your-own-data-to-azure-openai-chat-models/ba-p/3852630, Year of Publication. Accessed: 2024-01-04.

Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards predictable datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, page 242–253, New York, NY, USA, 2011a. Association for Computing Machinery. ISBN 9781450307970. doi:10.1145/2018436.2018465. URL https://doi.org/10.1145/2018436.2018465.

Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards predictable datacenter networks. *SIGCOMM Comput. Commun. Rev.*, 41(4):242–253, aug 2011b. ISSN 0146-4833. doi:10.1145/2043164.2018465. URL https://doi.org/10.1145/2043164.2018465.

Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022. doi:10.1007/978-3-030-99524-9_24. URL https://doi.org/10.1007/978-3-030-99524-9_24.

Nabajeet Barman and Maria G. Martini. Qoe modeling for http adaptive video streaming–a survey and open challenges. *IEEE Access*, 7:30831–30859, 2019. doi:10.1109/ACCESS.2019.2901778.

Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The Long-Document Transformer, 2020.

Emanuel Ben-Baruch, Tal Ridnik, Nadav Zamir, Asaf Noy, Itamar Friedman, Matan Protter, and Lihi Zelnik-Manor. Asymmetric loss for multi-label classification. *arXiv preprint arXiv:2009.14119*, 2020.

Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Anima Anandkumar. signSGD: Compressed Optimisation for Non-Convex Problems, 2018.

Amanda Bertsch, Uri Alon, Graham Neubig, and Matthew R Gormley. Unlimiformer: Long-range transformers with unlimited length input. *arXiv preprint arXiv:2305.01625*, 2023.

Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Inc., 1st edition, 2016. ISBN 149192912X.

Romil Bhardwaj, Zhengxu Xia, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Nikolaos Karianakis, Kevin Hsieh, Paramvir Bahl, and Ion Stoica. Ekya: Continuous Learning of Video Analytics Models on Edge Compute Servers. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, Renton, WA, April 2022. USENIX Association. URL https://www.usenix.org/conference/nsdi22/presentation/bhardwaj.

Mohammad Shafiquzzaman Bhuiyan. The role of ai-enhanced personalization in customer experiences. *Journal of Computer Science and Technology Studies*, 6(1):162–169, 2024.

Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. PIQA: Reasoning about Physical Commonsense in Natural Language, 2019.

Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George van den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, Diego de Las Casas, Aurelia Guy, Jacob Menick, Roman Ring, Tom Hennigan, Saffron Huang, Loren Maggiore, Chris Jones, Albin Cassirer, Andy Brock, Michela Paganini, Geoffrey Irving, Oriol Vinyals, Simon Osindero, Karen Simonyan, Jack W. Rae, Erich Elsen, and Laurent Sifre. Improving language models by retrieving from trillions of tokens, 2022.

Alexander Borzunov, Max Ryabinin, Artem Chumachenko, Dmitry Baranchuk, Tim Dettmers, Younes Belkada, Pavel Samygin, and Colin A Raffel. Distributed inference and fine-tuning of large language models over the internet. *Advances in neural information processing systems*, 36:12312–12331, 2023.

Gianni Brauwers and Flavius Frasincar. A general survey on attention mechanisms in deep learning. *IEEE Transactions on Knowledge and Data Engineering*, 35(4):3279–3298, 2021.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners, 2020.

Jiashen Cao, Ramyad Hadidi, Joy Arulraj, and Hyesoon Kim. Thia: Accelerating video analytics using early inference and fine-grained query planning. *arXiv preprint arXiv:2102.08481*, 2021.

Jiashen Cao, Karan Sarkar, Ramyad Hadidi, Joy Arulraj, and Hyesoon Kim. Figo: Fine-grained query optimization in video analytics. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 559–572, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392495. doi:10.1145/3514221.3517857. URL https://doi.org/10.1145/3514221.3517857.

Chi-Min Chan, Weize Chen, Yusheng Su, Jianxuan Yu, Wei Xue, Shanghang Zhang, Jie Fu, and Zhiyuan Liu. Chateval: Towards better llm-based evaluators through multi-agent debate, 2023. URL https://arxiv.org/abs/2308.07201.

Zhuoqing Chang, Shubo Liu, Xingxing Xiong, Zhaohui Cai, and Guoqing Tu. A survey of recent advances in edge-computing-powered artificial intelligence of things. *IEEE Internet of Things Journal*, 8(18):13849–13875, 2021. doi:10.1109/JIOT.2021.3088875.

Harrison Chase. LangChain, October 2022. URL https://github.com/langchain-ai/langchain.

Baian Chen, Chang Shu, Ehsan Shareghi, Nigel Collier, Karthik Narasimhan, and Shunyu Yao. Fireact: Toward language agent fine-tuning. *arXiv preprint arXiv:2310.05915*, 2023a.

Danqi Chen, Adam Fisch, Jason Weston, and Antoine Bordes. Reading wikipedia to answer open-domain questions, 2017.

Dian Chen, Dequan Wang, Trevor Darrell, and Sayna Ebrahimi. Contrastive test-time adaptation. In *CVPR*, pages 295–305, 2022a.

Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, et al. Coder: Issue resolving with multi-agent and task graphs. *arXiv preprint arXiv:2406.01304*, 2024a.

Jiaqi Chen, Yuxian Jiang, Jiachen Lu, and Li Zhang. S-agent: Self-organizing agents in open-ended environment. In *ICLR 2024 Workshop on Large Language Model Agents*, 2024b. URL `https://openreview.net/forum?id=N6RaMCT7pl`.

Jin Chen, Zheng Liu, Xu Huang, Chenwang Wu, Qi Liu, Gangwei Jiang, Yuanhao Pu, Yuxuan Lei, Xiaolong Chen, Xingmei Wang, et al. When large language models meet personalization: Perspectives of challenges and opportunities. *World Wide Web*, 27(4):42, 2024c.

Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. Punica: Multi-tenant lora serving. *Proceedings of Machine Learning and Systems*, 6:1–13, 2024d.

Lingjiao Chen, Matei Zaharia, and James Y Zou. Frugalml: How to use ml prediction apis more accurately and cheaply. *Advances in neural information processing systems*, 33: 10685–10696, 2020a.

Lingjiao Chen, Tracy Cai, Matei Zaharia, and James Zou. Did the model change? efficiently assessing machine learning api shifts. *arXiv preprint arXiv:2107.14203*, 2021a.

Lingjiao Chen, Matei Zaharia, and James Zou. Frugalmct: Efficient online ml api selection for multi-label classification tasks, 2022b. URL `https://openreview.net/forum?id=AypVMhFfuc5`.

Lingjiao Chen, Matei Zaharia, and James Zou. How is chatgpt's behavior changing over time?, 2023b. URL `https://arxiv.org/abs/2307.09009`.

Nuo Chen, Ning Wu, Shining Liang, Ming Gong, Linjun Shou, Dongmei Zhang, and Jia Li. Is bigger and deeper always better? probing llama across scales and layers. *arXiv preprint arXiv:2312.04333*, 2023c.

Shiyang Chen, Rain Jiang, Dezhi Yu, Jinlai Xu, Mengyuan Chao, Fanlong Meng, Chenyu Jiang, Wei Xu, and Hang Liu. Kvdirect: Distributed disaggregated llm inference. *arXiv preprint arXiv:2501.14743*, 2024e.

Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. Extending Context Window of Large Language Models via Positional Interpolation, 2023d.

Stanley F Chen, Douglas Beeferman, and Roni Rosenfeld. Evaluation Metrics For Language Models. 1 2008. doi:10.1184/R1/6605324.v1. URL https://kilthub.cmu.edu/articles/journal_contribution/Evaluation_Metrics_For_Language_Models/6605324.

Tianlong Chen, Zhenyu Zhang, Ajay Jaiswal, Shiwei Liu, and Zhangyang Wang. Sparse moe as the new dropout: Scaling dense and self-slimmable transformers, 2023e.

Weijian Chen, Shuibing He, Haoyang Qu, Ruidong Zhang, Siling Yang, Ping Chen, Yi Zheng, Baoxing Huai, and Gang Chen. IMPRESS: An Importance-Informed Multi-Tier prefix KV storage system for large language model inference. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 187–201, Santa Clara, CA, February 2025a. USENIX Association. ISBN 978-1-939133-45-8. URL https://www.usenix.org/conference/fast25/presentation/chen-weijian-impress.

Wenhu Chen, Ming-Wei Chang, Eva Schlinger, William Wang, and William W. Cohen. Open question answering over tables and text, 2021b.

Xinshi Chen, Yufei Zhang, Christoph Reisinger, and Le Song. Understanding deep architecture with reasoning layer. *Advances in Neural Information Processing Systems*, 33: 1240–1252, 2020b.

Zehui Chen, Kuikun Liu, Qiuchen Wang, Wenwei Zhang, Jiangning Liu, Dahua Lin, Kai Chen, and Feng Zhao. Agent-flan: Designing data and methods of effective agent tuning for large language models, 2024f. URL https://arxiv.org/abs/2403.12881.

Zhixun Chen, Ming Li, Yuxuan Huang, Yali Du, Meng Fang, and Tianyi Zhou. Atlas: Agent tuning via learning critical steps, 2025b. URL https://arxiv.org/abs/2503.02197.

Yihua Cheng, Kuntai Du, Jiayi Yao, and Junchen Jiang. Do large language models need a content delivery network? *arXiv preprint arXiv:2409.13761*, 2024.

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers, 2019.

Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Serving heterogeneous machine learning models on {Multi-GPU} servers with {Spatio-Temporal} sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 199–216, 2022.

Yujeong Choi, Yunseong Kim, and Minsoo Rhu. Lazybatching: An sla-aware batching system for cloud machine learning inference, 2020.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann,

et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

COCO. Coco dataset. https://cocodataset.org/#home, 2017.

Cohere. Command A: An enterprise-ready large language model, 2025. URL https://arxiv.org/abs/2504.00698. Model weights: https://huggingface.co/CohereLabs/c4ai-command-a-03-2025.

Pgvector Contributors. pgvector: Vector similarity search for postgresql. https://github.com/pgvector/pgvector, 2023.

CoreWeave. Hgx h100 | coreweave, 2024. URL https://www.coreweave.com/products/hgx-h100.

CoreWeave Documentation. Hpc interconnect networking | coreweave, 2024. URL https://docs.coreweave.com/networking/hpc-interconnect.

Microsoft Corporation. Bing. https://www.bing.com/, 2009.

Microsoft Corporation. Nd h100 v5 series - azure virtual machines, 2024. URL https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/gpu-accelerated/ndh100v5-series?tabs=sizebasic. Accessed: 2024-10-31.

NVIDIA Corporation. Nvidia multi-process service (mps) documentation, 2023. URL https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf. Accessed: 2023-07-12.

Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A Low-Latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, Boston, MA, March 2017. USENIX Association. ISBN 978-1-931971-37-9. URL https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw.

Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. Inferline: Latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 477–491, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381376. doi:10.1145/3419111.3421285. URL https://doi.org/10.1145/3419111.3421285.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019a.

Zihang Dai, Zhilin Yang, Yiming Yang, William W. Cohen, Jaime Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. Transformer-xl: Language modeling with longer-term dependency, 2019b. URL https://openreview.net/forum?id=HJePno0cYm.

Sumit Kumar Dam, Choong Seon Hong, Yu Qiao, and Chaoning Zhang. A complete survey on llm-based ai chatbots. *arXiv preprint arXiv:2406.16937*, 2024.

Carlos d'Andrea and Andre Mintz. Studying the live cross-platform circulation of images with computer vision api: An experiment based on a sports media event. *International Journal of Communication*, 13:21, 2019.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness, 2022.

Mallesham Dasari, Kumara Kahatapitiya, Samir R. Das, Aruna Balasubramanian, and Dimitris Samaras. Swift: Adaptive video streaming with layered neural codecs. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 103–118, Renton, WA, April 2022. USENIX Association. ISBN 978-1-939133-27-4. URL https://www.usenix.org/conference/nsdi22/presentation/dasari.

Michiel de Jong, Yury Zemlyanskiy, Joshua Ainslie, Nicholas FitzGerald, Sumit Sanghai, Fei Sha, and William Cohen. FiDO: Fusion-in-Decoder optimized for stronger performance and faster inference, 2023.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009. doi:10.1109/CVPR.2009.5206848.

Jiajun Deng, Yingwei Pan, Ting Yao, Wengang Zhou, Houqiang Li, and Tao Mei. Relation distillation networks for video object detection. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 7023–7032, 2019.

Radosvet Desislavov, Fernando Martinez-Plumed, and Jose Hernandez-Orallo. Compute and energy consumption trends in deep learning inference, 2021. URL https://arxiv.org/abs/2109.05472.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm. int8 (): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339*, 2022.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Jiayu Ding, Shuming Ma, Li Dong, Xingxing Zhang, Shaohan Huang, Wenhui Wang, Nanning Zheng, and Furu Wei. LongNet: Scaling Transformers to 1,000,000,000 Tokens, 2023a.

Ning Ding, Yulin Chen, Bokai Xu, Yujia Qin, Shengding Hu, Zhiyuan Liu, Maosong Sun, and Bowen Zhou. Enhancing chat language models by scaling high-quality instructional conversations. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 3029–3051, Singapore, December 2023b. Association for Computational Linguistics. doi:10.18653/v1/2023.emnlp-main.183. URL https://aclanthology.org/2023.emnlp-main.183/.

Shiyao Ding and Takayuki Ito. Self-Agreement: A Framework for Fine-tuning Language Models to Find Agreement among Diverse Opinions, 2023.

Yiran Ding, Li Lyna Zhang, Chengruidong Zhang, Yuanyuan Xu, Ning Shang, Jiahang Xu, Fan Yang, and Mao Yang. Longrope: Extending llm context window beyond 2 million tokens. *arXiv preprint arXiv:2402.13753*, 2024.

Chenpeng Du, Yiwei Guo, Hankun Wang, Yifan Yang, Zhikang Niu, Shuai Wang, Hui Zhang, Xie Chen, and Kai Yu. Vall-t: Decoder-only generative transducer for robust and decoding-controllable text-to-speech. In *ICASSP 2025-2025 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1–5. IEEE, 2025a.

Mingxuan Du, Benfeng Xu, Chiwei Zhu, Xiaorui Wang, and Zhendong Mao. Deepresearch bench: A comprehensive benchmark for deep research agents. *arXiv preprint*, 2025b.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate, 2023. URL https://arxiv.org/abs/2305.14325.

Nader Ebrahimi, Esfandiar Maasoumi, and Ehsan S. Soofi. Ordering univariate distributions by entropy and variance. *Journal of Econometrics*, 90(2):317–336, 1999. ISSN 0304-4076. doi:https://doi.org/10.1016/S0304-4076(98)00046-3. URL https://www.sciencedirect.com/science/article/pii/S0304407698000463.

Elastic. Elasticsearch. https://github.com/elastic/elasticsearch, 2023.

Amr ElRafey and Janusz Wojtusiak. Recent advances in scaling-down sampling methods in machine learning. *Wiley Interdisciplinary Reviews: Computational Statistics*, 9(6):e1414, 2017.

Facebook Engineering. Faiss: A library for efficient similarity search, 2017. URL https://engineering.fb.com/2017/03/29/data-infrastructure/faiss-a-library-for-efficient-similarity-search/. Accessed: 2023-09-29.

Tiannan Wang et al. Weaver: Foundation models for creative writing, 2024. URL https://arxiv.org/abs/2401.17268.

Alexander Fabbri, Irene Li, Tianwei She, Suyi Li, and Dragomir Radev. Multi-news: A large-scale multi-document summarization dataset and abstractive hierarchical model. In Anna

Korhonen, David Traum, and Lluís Màrquez, editors, *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 1074–1084, Florence, Italy, July 2019. Association for Computational Linguistics. doi:10.18653/v1/P19-1102. URL https://aclanthology.org/P19-1102/.

Jiabao Fang, Shen Gao, Pengjie Ren, Xiuying Chen, Suzan Verberne, and Zhaochun Ren. A multi-agent conversational recommender system. *arXiv preprint arXiv:2402.01135*, 2024.

Shangbin Feng, Wenxuan Ding, Alisa Liu, Zifeng Wang, Weijia Shi, Yike Wang, Zejiang Shen, Xiaochuang Han, Hunter Lang, Chen-Yu Lee, et al. When one llm drools, multi-llm collaboration rules. *arXiv preprint arXiv:2502.04506*, 2025.

Javier Ferrando, Gabriele Sarti, Arianna Bisazza, and Marta R Costa-Jussà. A primer on the inner workings of transformer-based language models. *arXiv preprint arXiv:2405.00208*, 2024.

Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S. Wahby, and Keith Winstein. Salsify: low-latency network video through tighter integration between a video codec and a transport protocol. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, page 267–282, USA, 2018. USENIX Association. ISBN 9781931971430.

FriendliAI Tech & Research. Hassle-free llm fine-tuning with friendliai and weights & biases. https://friendli.ai/blog/wandb-friendli-dedicated-endpoints, June 2024. Accessed: 2023-11-11.

Zhenxiao Fu, Fan Chen, Shan Zhou, Haitong Li, and Lei Jiang. LLMCO2: Advancing accurate carbon footprint prediction for LLM inferences, 2024. URL https://arxiv.org/abs/2410.02950.

Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. Attentionstore: Cost-effective attention reuse across multi-turn conversations in large language model serving. *arXiv preprint arXiv:2403.19708*, 2024a.

Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. Cost-efficient large language model serving for multi-turn conversations with cachedattention. In *Proceedings of the 2024 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'24, USA, 2024b. USENIX Association. ISBN 978-1-939133-41-0.

Chen Gao, Xiaochong Lan, Zhihong Lu, Jinzhu Mao, Jinghua Piao, Huandong Wang, Depeng Jin, and Yong Li. S3: Social-network simulation system with large language model-empowered agents. *arXiv preprint arXiv:2307.14984*, 2023.

Hang Gao and Yongfeng Zhang. Memory sharing for large language model based agents, 2024. URL https://arxiv.org/abs/2404.09982.

Shiwei Gao, Youmin Chen, and Jiwu Shu. Fast state restoration in llm serving with hcache, 2024c. URL https://arxiv.org/abs/2410.05004.

Silin Gao, Jane Dwivedi-Yu, Ping Yu, Xiaoqing Ellen Tan, Ramakanth Pasunuru, Olga Golovneva, Koustuv Sinha, Asli Celikyilmaz, Antoine Bosselut, and Tianlu Wang. Efficient tool use with chain-of-abstraction reasoning, 2024d. URL https://arxiv.org/abs/2401.17464.

Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Qianyu Guo, Meng Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey, 2024e.

Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, and Jianfeng Gao. Model tells you what to discard: Adaptive kv cache compression for llms, 2023a.

Tao Ge, Jing Hu, Xun Wang, Si-Qing Chen, and Furu Wei. In-context Autoencoder for Context Compression in a Large Language Model. *arXiv preprint arXiv:2307.06945*, 2023b.

Tao Ge, Hu Jing, Lei Wang, Xun Wang, Si-Qing Chen, and Furu Wei. In-context autoencoder for context compression in a large language model. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=uREj4ZuGJE.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. Rlef: Grounding code llms in execution feedback with reinforcement learning, 2024. URL https://arxiv.org/abs/2410.02089.

In Gim, Guojun Chen, Seung seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt cache: Modular attention reuse for low-latency inference, 2023a.

In Gim, Guojun Chen, Seung seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt cache: Modular attention reuse for low-latency inference, 2023b.

In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt cache: Modular attention reuse for low-latency inference. *Proceedings of Machine Learning and Systems*, 6:325–338, 2024.

GitHub. Github copilot. https://github.com/features/copilot, 2025. Accessed: 2025-04-23.

Tilmann Gneiting and Adrian E Raftery. Strictly proper scoring rules, prediction, and estimation. *Journal of the American Statistical Association*, 102(477):359–378, 2007. doi:10.1198/016214506000001437. URL https://doi.org/10.1198/016214506000001437.

Ran Gong, Qiuyuan Huang, Xiaojian Ma, Yusuke Noda, Zane Durante, Zilong Zheng, Demetri Terzopoulos, Li Fei-Fei, Jianfeng Gao, and Hoi Vo. MindAgent: Emergent gaming interaction. In Kevin Duh, Helena Gomez, and Steven Bethard, editors, *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 3154–3183, Mexico City, Mexico, June 2024. Association for Computational Linguistics. doi:10.18653/v1/2024.findings-naacl.200. URL https://aclanthology.org/2024.findings-naacl.200/.

Google. Google cloud ai. Online document https://cloud.google.com/products/ai, 2022.

Google. Compute engine pricing. Online document https://cloud.google.com/compute/all-pricing, 2023.

Google DeepMind. Gemini api documentation: Long context. https://ai.google.dev/gemini-api/docs/long-context, 2025. (Accessed on 07/16/2026).

Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A {GPU} cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, 2019.

Yuxian Gu, Li Dong, Furu Wei, and Minlie Huang. Minillm: Knowledge distillation of large language models, 2024a. URL https://arxiv.org/abs/2306.08543.

Zhuohan Gu, Jiayi Yao, Kuntai Du, and Junchen Jiang. Llmsteer: Improving long-context llm inference by steering attention on reused contexts, 2024b. URL https://arxiv.org/abs/2411.13009.

Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving {DNNs} like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462, 2020a.

Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving {DNNs} like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462, 2020b.

Daya Guo, Canwen Xu, Nan Duan, Jian Yin, and Julian McAuley. Longcoder: a long-range pre-trained language model for code completion. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. Large language model based multi-agents: a survey of progress and challenges. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, IJCAI '24, 2024. ISBN 978-1-956792-04-1. doi:10.24963/ijcai.2024/890. URL https://doi.org/10.24963/ijcai.2024/890.

Desta Haileselassie Hagos, Rick Battle, and Danda B. Rawat. Recent advances in generative ai and large language models: Current status, challenges, and perspectives. *IEEE Transactions on Artificial Intelligence*, 5(12):5873–5893, 2024. doi:10.1109/TAI.2024.3444742.

Armin Haller, Axel Polleres, Daniil Dobriy, Nicolas Ferranti, and Sergio J Rodríguez Méndez. An analysis of links in wikidata. In *European Semantic Web Conference*, pages 21–38. Springer, 2022.

Dave Halter. Jedi: an awesome auto-completion, static analysis and refactoring library for python. Online document `https://jedi.readthedocs.io`.

Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent {GPU-accelerated}{DNN} inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, 2022.

Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, page 123–136, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342698. doi:10.1145/2906388.2906396. URL `https://doi.org/10.1145/2906388.2906396`.

Curtis Hawthorne, Andrew Jaegle, Cătălina Cangea, Sebastian Borgeaud, Charlie Nash, Mateusz Malinowski, Sander Dieleman, Oriol Vinyals, Matthew Botvinick, Ian Simon, Hannah Sheahan, Neil Zeghidour, Jean-Baptiste Alayrac, Joao Carreira, and Jesse Engel. General-purpose, long-context autoregressive modeling with perceiver AR. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 8535–8558. PMLR, 17–23 Jul 2022. URL `https://proceedings.mlr.press/v162/hawthorne22a.html`.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

Hermann Hellwagner, Ingo Kofler, Michael Eberhard, Robert Kuschnig, Michael Ransburg, and Michael Sablatschan. *Scalable Video Coding: Techniques and Applications for Adaptive Streaming*, pages 1–23. 01 2011. doi:10.4018/978-1-61692-831-5.

Xanh Ho, Anh-Khoa Duong Nguyen, Saku Sugawara, and Akiko Aizawa. Constructing a multi-hop QA dataset for comprehensive evaluation of reasoning steps. In Donia Scott, Nuria Bel, and Chengqing Zong, editors, *Proceedings of the 28th International Conference on Computational Linguistics*, pages 6609–6625, Barcelona, Spain (Online), December 2020. International Committee on Computational Linguistics. doi:10.18653/v1/2020.coling-main.580. URL `https://aclanthology.org/2020.coling-main.580/`.

Connor Holmes, Daniel Mawhirter, Yuxiong He, Feng Yan, and Bo Wu. Grnn: Low-latency and scalable rnn inference on gpus. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362818. doi:10.1145/3302424.3303949. URL https://doi.org/10.1145/3302424.3303949.

Samuel Holt, Max Ruiz Luyten, and Mihaela van der Schaar. L2mac: Large language model automatic computer for extensive code generation. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=EhrzQwsV4K.

Sirui Hong, Yizhang Lin, Bang Liu, Bangbang Liu, Binhao Wu, Danyang Li, Jiaqi Chen, Jiayi Zhang, Jinlin Wang, Li Zhang, Lingyao Zhang, Min Yang, Mingchen Zhuge, Taicheng Guo, Tuo Zhou, Wei Tao, Wenyi Wang, Xiangru Tang, Xiangtao Lu, Xiawu Zheng, Xinbing Liang, Yaying Fei, Yuheng Cheng, Zongze Xu, and Chenglin Wu. Data interpreter: An llm agent for data science, 2024a.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*, 2024b. URL https://openreview.net/forum?id=VtmBAGCN7o.

Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxiao Dong, Ming Ding, et al. Cogagent: A visual language model for gui agents. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14281–14290, 2024c.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *arXiv preprint arXiv:2401.18079*, 2024.

Hossein Hosseini, Baicen Xiao, and Radha Poovendran. Google's cloud vision api is not robust to noise. In *2017 16th IEEE international conference on machine learning and applications (ICMLA)*, pages 101–105. IEEE, 2017.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, and Boris Ginsburg. Ruler: What's the real context size of your long-context language models? *arXiv preprint arXiv:2404.06654*, 2024.

Cunchen Hu, Heyang Huang, Junhao Hu, Jiang Xu, Xusheng Chen, Tao Xie, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, et al. Memserve: Context caching for disaggregated llm serving with elastic memory pool. *arXiv preprint arXiv:2406.17565*, 2024a.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022.

Mengkang Hu, Yuhang Zhou, Wendong Fan, Yuzhou Nie, Bowei Xia, Tao Sun, Ziyu Ye, Zhaoxuan Jin, Yingru Li, Qiguang Chen, et al. Owl: Optimized workforce learning for general multi-agent assistance in real-world task automation. *arXiv preprint arXiv:2505.23885*, 2025a.

Qitian Jason Hu, Jacob Bieker, Xiuyu Li, Nan Jiang, Benjamin Keigwin, Gaurav Ranganath, Kurt Keutzer, and Shriyash Kaustubh Upadhyay. Routerbench: A benchmark for multi-llm routing system, 2024b. URL https://arxiv.org/abs/2403.12031.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *The Thirteenth International Conference on Learning Representations*, 2025b. URL https://openreview.net/forum?id=t9U3LW7JVX.

Haochen Hua, Yutong Li, Tonghe Wang, Nanqing Dong, Wei Li, and Junwei Cao. Edge computing with artificial intelligence: A machine learning perspective. *ACM Comput. Surv.*, aug 2022. ISSN 0360-0300. doi:10.1145/3555802. URL https://doi.org/10.1145/3555802. Just Accepted.

Dong Huang, Jie M Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*, 2023.

Yuyang Huang, Yuhan Liu, Haryadi S Gunawi, Beibin Li, and Changho Hwang. Alchemist: Towards the design of efficient online continual learning system. *arXiv preprint arXiv:2503.01066*, 2025.

Zhang Huangzhao. Yahoo_question_answer. https://github.com/LC-John/Yahoo-Answers-Topic-Classification-Dataset.git, 2018.

Minyoung Huh, Brian Cheung, Tongzhou Wang, and Phillip Isola. The platonic representation hypothesis, 2024. URL https://arxiv.org/abs/2405.07987.

International Energy Agency. Energy and ai. Technical report, International Energy Agency, 2025. URL https://www.iea.org/reports/energy-and-ai.

M Irlbeck et al. Deconstructing dynamic symbolic execution. *Dependable Software Systems Engineering*, 40:26, 2015.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. MapCoder: Multi-agent code generation for competitive problem solving. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4912–4944, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi:10.18653/v1/2024.acl-long.269. URL https://aclanthology.org/2024.acl-long.269/.

Gautier Izacard and Edouard Grave. Leveraging passage retrieval with generative models for open domain question answering. *arXiv preprint arXiv:2007.01282*, 2020.

Gautier Izacard and Edouard Grave. Leveraging Passage Retrieval with Generative Models for Open Domain Question Answering, 2021.

Gautier Izacard, Fabio Petroni, Lucas Hosseini, Nicola De Cao, Sebastian Riedel, and Edouard Grave. A Memory Efficient Baseline for Open Domain Question Answering, 2020.

Gautier Izacard, Patrick Lewis, Maria Lomeli, Lucas Hosseini, Fabio Petroni, Timo Schick, Jane Dwivedi-Yu, Armand Joulin, Sebastian Riedel, and Edouard Grave. Few-shot learning with retrieval augmented language models. *arXiv preprint arXiv:2208.03299*, 2022.

Naman Jain, Tianjun Zhang, Wei-Lin Chiang, Joseph E. Gonzalez, Koushik Sen, and Ion Stoica. Llm-assisted code cleaning for training accurate code generators, 2023a.

Paras Jain, Sam Kumar, Sarah Wooders, Shishir G. Patil, Joseph E. Gonzalez, and Ion Stoica. Skyplane: Optimizing transfer cost and throughput using Cloud-Aware overlays. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1375–1389, Boston, MA, April 2023b. USENIX Association. ISBN 978-1-939133-33-5. URL `https://www.usenix.org/conference/nsdi23/presentation/jain`.

Hanhwi Jang, Joonsung Kim, Jae-Eon Jo, Jaewon Lee, and Jangwoo Kim. Mnnfast: A fast and scalable system architecture for memory-augmented neural networks. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 250–263, 2019.

Debabrata Mukher jee and Makarand V Ratnaparkhi. On the functional relationship between entropy and variance with related applications. *Communications in Statistics - Theory and Methods*, 15(1):291–311, 1986. doi:10.1080/03610928608829122. URL `https://www.tandfonline.com/doi/abs/10.1080/03610928608829122`.

Cave Jeffrey. soap. `https://github.com/jcavejr/soap.git`, 2018.

Cheonsu Jeong. Domain-specialized llm: Financial fine-tuning and utilization method using mistral 7b. *Journal of Intelligence and Information Systems*, 30(1):93–120, March 2024. ISSN 2288-4882. doi:10.13088/jiis.2024.30.1.093. URL `http://dx.doi.org/10.13088/jiis.2024.30.1.093`.

Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Albert Greenberg, and Changhoon Kim. EyeQ: Practical network performance isolation at the edge. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 297–311, Lombard, IL, April 2013. USENIX Association. ISBN 978-1-931971-00-3. URL `https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/jeyakumar`.

Angela H. Jiang, Daniel L.-K. Wong, Christopher Canel, Lilia Tang, Ishan Misra, Michael Kaminsky, Michael A. Kozuch, Padmanabhan Pillai, David G. Andersen, and Gregory R. Ganger. Mainstream: Dynamic Stem-Sharing for Multi-Tenant video processing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 29–42, Boston, MA, July 2018. USENIX Association. ISBN 978-1-931971-44-7. URL https://www.usenix.org/conference/atc18/presentation/jiang.

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Llmlingua: Compressing prompts for accelerated inference of large language models, 2023a.

Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Longllmlingua: Accelerating and enhancing llms in long context scenarios via prompt compression, 2023b.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2023.

Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Xin Liu, Xuanzhe Liu, and Xin Jin. Ragcache: Efficient knowledge caching for retrieval-augmented generation. *arXiv preprint arXiv:2404.12457*, 2024a.

Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Shufan Liu, Xuanzhe Liu, and Xin Jin. Ragcache: Efficient knowledge caching for retrieval-augmented generation. *ACM Trans. Comput. Syst.*, September 2025a. ISSN 0734-2071. doi:10.1145/3768628. URL https://doi.org/10.1145/3768628. Just Accepted.

Shuowei Jin, Xueshen Liu, Qingzhao Zhang, and Zhuoqing Mao. Compute or load kv cache? why not both? In *Forty-second International Conference on Machine Learning*, 2025b. URL https://openreview.net/forum?id=WOyOtaO6lQ.

Yiqiao Jin, Qinlin Zhao, Yiyang Wang, Hao Chen, Kaijie Zhu, Yijia Xiao, and Jindong Wang. Agentreview: Exploring peer review dynamics with llm agents, 2024b. URL https://arxiv.org/abs/2406.12708.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs, 2017.

Mandar Joshi, Eunsol Choi, Daniel S. Weld, and Luke Zettlemoyer. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension, 2017.

Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12, 2017. doi:10.1145/3079856.3080246.

Philipp Jund, Andreas Eitel, Nichola Abdo, and Wolfram Burgard. Optimization beyond the convolution: Generalizing spatial relations with end-to-end metric learning. *CoRR*, abs/1707.00893, 2017. URL http://arxiv.org/abs/1707.00893.

Heechul Jung, Sihaeng Lee, Junho Yim, Sunjeong Park, and Junmo Kim. Joint fine-tuning in deep neural networks for facial expression recognition. In *Proceedings of the IEEE international conference on computer vision*, pages 2983–2991, 2015.

jwatte. How does chatgpt store history of chat. `https://community.openai.com/t/how-does-chatgpt-store-history-of-chat/319608/2`, Aug 2023. OpenAI Community Forum.

Christoph Käding, Erik Rodner, Alexander Freytag, and Joachim Denzler. Fine-tuning deep neural networks in continuous learning scenarios. In *Asian Conference on Computer Vision*, pages 588–605. Springer, 2016.

Waleed Kadous. Numbers every llm developer should know. https://www.anyscale.com/blog/num-every-llm-developer-should-know, 2023.

Waleed Kadous, Kyle Huang, Wendi Ding, Liguang Xie, Avnish Narayan, and Ricky Xu. Reproducible performance metrics for llm inference, November 2023. URL `https://www.anyscale.com/blog/reproducible-performance-metrics-for-llm-inference`. Accessed: 2024-06-01.

Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Noscope: optimizing neural network queries over video at scale. *arXiv preprint arXiv:1703.02529*, 2017.

Daniel Kang, Peter Bailis, and Matei Zaharia. Blazeit: Optimizing declarative aggregation and limit queries for neural network-based video analytics. *arXiv preprint arXiv:1805.01046*, 2018.

Hao Kang, Qingru Zhang, Souvik Kundu, Geonhwa Jeong, Zaoxing Liu, Tushar Krishna, and Tuo Zhao. Gear: An efficient kv cache compression recipefor near-lossless generative inference of llm. *arXiv preprint arXiv:2403.05527*, 2024.

Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. Grandslam: Guaranteeing slas for jobs in microservices execution frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

Shyam Sundar Kannan, Vishnunandan LN Venkatesh, and Byung-Cheol Min. Smart-llm: Smart multi-agent robot task planning using large language models. *arXiv preprint arXiv:2309.10062*, 2023.

Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen tau Yih. Dense passage retrieval for open-domain question answering, 2020.

Enkelejda Kasneci, Kathrin Seßler, Stefan Küchemann, Maria Bannert, Daryna Dementieva, Frank Fischer, Urs Gasser, Georg Groh, Stephan Günnemann, Eyke Hüllermeier, et al. ChatGPT for good? On opportunities and challenges of large language models for education. *Learning and individual differences*, 103:102274, 2023.

Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. Lessons learned from the chameleon testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020a.

Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbah, Alex Rocha, and Joe Stubbs. Lessons learned from the chameleon testbed. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 219–233. USENIX Association, July 2020b. ISBN 978-1-939133-14-4. URL `https://www.usenix.org/conference/atc20/presentation/keahey`.

KelSolaar. fvvt-kels-utilities. `https://github.com/KelSolaar/fvvt-kels-utilities.git`, 2023. GitHub repository.

Phillip Keung, Yichao Lu, György Szarvas, and Noah A. Smith. The multilingual amazon reviews corpus. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*, 2020.

Prathamesh Khade. Multi-agent system for content creation, 2023. URL `https://medium.com/@prathamesh.khade20/multi-agent-system-for-content-creation-aaefa5350012`. Accessed: 2024-10-13.

Hakbin Kim and Dong-Wan Choi. Pool of experts: Realtime querying specialized knowledge in massive neural networks. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2244–2252, 2021a.

Hakbin Kim and Dong-Wan Choi. Pool of experts: Realtime querying specialized knowledge in massive neural networks. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2244–2252, New York, NY, USA, 2021b. Association for Computing Machinery. ISBN 9781450383431. doi:10.1145/3448016.3457326. URL `https://doi.org/10.1145/3448016.3457326`.

Minsu Kim, Seongmin Hong, RyeoWook Ko, Soongyu Choi, Hunjong Lee, Junsoo Kim, Joo-Young Kim, and Jongse Park. Oaken: Fast and efficient llm serving with online-offline hybrid kv cache quantization. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ISCA '25, page 482–497, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400712616. doi:10.1145/3695053.3731019. URL `https://doi.org/10.1145/3695053.3731019`.

Sehoon Kim, Coleman Hooper, Thanakul Wattanawong, Minwoo Kang, Ruohan Yan, Hasan Genc, Grace Dinh, Qijing Huang, Kurt Keutzer, Michael W. Mahoney, Yakun Sophia Shao, and Amir Gholami. Full Stack Optimization of Transformer Inference: a Survey, 2023.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The Efficient Transformer, 2020.

Allison Koenecke, Andrew Nam, Emily Lake, Joe Nudell, Minnie Quartey, Zion Mengesha, Connor Toups, John R Rickford, Dan Jurafsky, and Sharad Goel. Racial disparities in automated speech recognition. *Proceedings of the National Academy of Sciences*, 117(14): 7684–7689, 2020.

Mojtaba Komeili, Kurt Shuster, and Jason Weston. Internet-augmented dialogue generation. *arXiv preprint arXiv:2107.07566*, 2021.

Jack Kosaian, K. V. Rashmi, and Shivaram Venkataraman. Parity models: Erasure-coded resilience for prediction serving systems. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 30–46, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi:10.1145/3341301.3359654. URL `https://doi.org/10.1145/3341301.3359654`.

Nick Koudas, Raymond Li, and Ioannis Xarchakos. Video monitoring queries. *IEEE Transactions on Knowledge and Data Engineering*, 2020.

Tomáš Kočiský, Jonathan Schwarz, Phil Blunsom, Chris Dyer, Karl Moritz Hermann, Gábor Melis, and Edward Grefenstette. The narrativeqa reading comprehension challenge, 2017.

Jason Kuen, Federico Perazzi, Zhe Lin, Jianming Zhang, and Yap-Peng Tan. Scaling object detection by transferring classification weights. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 6044–6053, 2019.

Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Alexander Kolesnikov, et al. The open images dataset v4. *International Journal of Computer Vision*, 128(7):1956–1981, 2020.

Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Matthew Kelcey, Jacob Devlin, Kenton Lee, Kristina N. Toutanova, Llion Jones, Ming-Wei Chang, Andrew Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. Natural questions: a benchmark for question answering research. *Transactions of the Association of Computational Linguistics*, 2019.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA, 2023a. Association for Computing Machinery. ISBN 9798400702297. doi:10.1145/3600006.3613165. URL `https://doi.org/10.1145/3600006.3613165`.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023b.

Modal Labs. Slack fine-tuning with modal, 2024. URL https://modal.com/docs/examples/slack-finetune. Accessed: 2024-11-19.

Ken Lang. Newsweeder: Learning to filter netnews. In Armand Prieditis and Stuart Russell, editors, *Machine Learning Proceedings 1995*, pages 331–339. Morgan Kaufmann, San Francisco (CA), 1995. ISBN 978-1-55860-377-6. doi:https://doi.org/10.1016/B978-1-55860-377-6.50048-7. URL https://www.sciencedirect.com/science/article/pii/B9781558603776500487.

Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. InfiniGen: Efficient generative inference of large language models with dynamic KV cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 155–172, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/lee.

Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. {PRETZEL}: Opening the black box of machine learning prediction serving systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 611–626, 2018.

Benjamin Lefaudeux, Francisco Massa, Diana Liskovich, Wenhan Xiong, Vittorio Caggiano, Sean Naren, Min Xu, Jieru Hu, Marta Tintore, Susan Zhang, Patrick Labatut, Daniel Haziza, Luca Wehrstedt, Jeremy Reizenstein, and Grigory Sizov. xformers: A modular and hackable transformer modelling library. https://github.com/facebookresearch/xformers, 2022.

Letta. *Memory*, 2024. https://docs.letta.com/agents/memory.

Yaniv Leviathan, Matan Kalman, and Y. Matias. Fast Inference from Transformers via Speculative Decoding. In *International Conference on Machine Learning*, 2022. URL https://api.semanticscholar.org/CorpusID:254096365.

Zijian Lew, Joseph B Walther, Augustine Pang, and Wonsun Shin. Interactivity in Online Chat: Conversational Contingency and Response Latency in Computer-mediated Communication. *Journal of Computer-Mediated Communication*, 23(4):201–221, 06 2018. ISSN 1083-6101. doi:10.1093/jcmc/zmy009. URL https://doi.org/10.1093/jcmc/zmy009.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2021.

Baolin Li, Yankai Jiang, Vijay Gadepally, and Devesh Tiwari. Llm inference serving: Survey of recent advances and opportunities. In *2024 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8. IEEE, 2024a.

Dacheng Li*, Rulin Shao*, Anze Xie, Lianmin Zheng Ying Sheng, Joseph E. Gonzalez, Ion Stoica, Xuezhe Ma, and Hao Zhang. How long can open-source llms truly promise on context length?, June 2023. URL https://lmsys.org/blog/2023-06-29-longchat.

Hanchen Li, Yihua Cheng, Ziyi Zhang, Qizheng Zhang, Anton Arapin, Nick Feamster, and Amrita Mazumdar. Optimizing real-time video experience with data scalable codec. In *Proceedings of the 2023 Workshop on Emerging Multimedia Systems*, EMS '23, page 15–21, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400703034. doi:10.1145/3609395.3611108. URL https://doi.org/10.1145/3609395.3611108.

Haoran Li, Jinyu Wang, Yong Su, and Yue Zhang. Marft: Multi-agent reinforcement fine-tuning. *arXiv preprint arXiv:2504.16129*, 2025.

Jie Li, Haifeng Liu, Chuanghua Gui, Jianyu Chen, Zhenyun Ni, and Ning Wang. The design and implementation of a real time visual search system on jd e-commerce platform, 2019.

Jingjing Li, Mengmeng Jing, Hongzu Su, Ke Lu, Lei Zhu, and Heng Tao Shen. Faster domain adaptation networks. *IEEE Transactions on Knowledge and Data Engineering*, 2021.

Junyou Li, Qin Zhang, Yangbin Yu, Qiang Fu, and Deheng Ye. More agents is all you need. *Transactions on Machine Learning Research*, 2024b. URL https://openreview.net/forum?id=bgzUSZ8aeg.

Suyi Li, Hanfeng Lu, Tianyuan Wu, Minchen Yu, Qizhen Weng, Xusheng Chen, Yizhou Shan, Binhang Yuan, and Wei Wang. Caraserve: Cpu-assisted and rank-aware lora serving for generative llm inference, 2024c. URL https://arxiv.org/abs/2401.11240.

Yuanchun Li, Hao Wen, Weijun Wang, Xiangyu Li, Yizhen Yuan, Guohong Liu, Jiacheng Liu, Wenxing Xu, Xiang Wang, Yi Sun, Rui Kong, Yile Wang, Hanfei Geng, Jian Luan, Xuefeng Jin, Zilong Ye, Guanjing Xiong, Fan Zhang, Xiang Li, Mengwei Xu, Zhijun Li, Peng Li, Yang Liu, Ya-Qin Zhang, and Yunxin Liu. Personal llm agents: Insights and survey about the capability, efficiency and security. *arXiv preprint arXiv:2401.05459*, 2024d.

Zonglin Li, Ruiqi Guo, and Sanjiv Kumar. Decoupled context processing for context augmented language modeling. *arXiv preprint arXiv:2210.05758*, 2022.

Tian Liang, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Shuming Shi, and Zhaopeng Tu. Encouraging divergent thinking in large language models through multi-agent debate, 2024. URL https://arxiv.org/abs/2305.19118.

Bin Lin, Tao Peng, Chen Zhang, Minmin Sun, Lanbo Li, Hanyu Zhao, Wencong Xiao, Qi Xu, Xiafei Qiu, Shen Li, Zhigang Ji, Yong Li, and Wei Lin. Infinite-llm: Efficient llm service for long context with distattention and distributed kvcache, 2024a.

Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. Parrot: Efficient serving of LLM-based applications with semantic variable. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 929–945, Santa Clara, CA, July 2024b. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/lin-chaofan.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. In P. Gibbons, G. Pekhimenko, and C. De Sa, editors, *Proceedings of Machine Learning and Systems*, volume 6, pages 87–100, 2024c. URL https://proceedings.mlsys.org/paper_files/paper/2024/file/42a452cbafa9dd64e9ba4aa95cc1ef21-Paper-Conference.pdf.

Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. A survey of transformers, 2021.

Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. A survey of transformers. *AI open*, 3:111–132, 2022.

Yong Lin, Lu Tan, Hangyu Lin, Zeming Zheng, Renjie Pi, Jipeng Zhang, Shizhe Diao, Haoxiang Wang, Han Zhao, Yuan Yao, et al. Speciality vs Generality: An Empirical Study on Catastrophic Forgetting in Fine-tuning Foundation Models. *arXiv preprint arXiv:2309.06256*, 2023.

Jiachen Liu, Zhiyu Wu, Jae-Won Chung, Fan Lai, Myungjin Lee, and Mosharaf Chowdhury. Andes: Defining and enhancing quality-of-experience in llm-based text streaming services. *arXiv preprint arXiv:2404.16283*, 2024a.

Jijia Liu, Chao Yu, Jiaxuan Gao, Yuqing Xie, Qingmin Liao, Yi Wu, and Yu Wang. Llm-powered hierarchical language agent for real-time human-ai coordination. In *Proceedings of the 23rd International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '24, page 1219–1228, Richland, SC, 2024b. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 9798400704864.

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172*, 2023a.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the Middle: How Language Models Use Long Contexts, 2023b.

Shigang Liu, Jun Zhang, Yang Xiang, Wanlei Zhou, and Dongxi Xiang. A study of data preprocessing techniques for imbalanced biomedical data classification. *International Journal of Bioinformatics Research and Applications*, 16(3):290–318, 2020.

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations*, 2024c. URL https://openreview.net/forum?id=pPjZIOuQuF.

Yuhan Liu, Saurabh Agarwal, and Shivaram Venkataraman. Autofreeze: Automatically freezing model blocks to accelerate fine-tuning, 2021. URL https://arxiv.org/abs/2102.01386.

Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. Cachegen: Kv cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 38–56, New York, NY, USA, 2024d. Association for Computing Machinery. ISBN 9798400706141. doi:10.1145/3651890.3672274. URL https://doi.org/10.1145/3651890.3672274.

Yuhan Liu, Jiayi Yao, Yihua Cheng, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Rui Zhang, Kuntai Du, and Junchen Jiang. Lmcache: An efficient kv cache layer for enterprise-scale llm inference. https://arxiv.org/abs/2510.09665, 2025.

Zhenhua Liu, Zhiwei Hao, Kai Han, Yehui Tang, and Yunhe Wang. Ghostnetv3: Exploring the training strategies for compact models. *arXiv preprint arXiv:2404.11202*, 2024e.

Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. Scissorhands: Exploiting the Persistence of Importance Hypothesis for LLM KV Cache Compression at Test Time. *arXiv preprint arXiv:2305.17118*, 2023c.

Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. Scissorhands: Exploiting the Persistence of Importance Hypothesis for LLM KV Cache Compression at Test Time. *arXiv preprint arXiv:2305.17118*, 2023d.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *arXiv preprint arXiv:2402.02750*, 2024f.

LMCache Team. Context engineering & reuse pattern under the hood of claude code. https://blog.lmcache.ai/en/2025/12/23/context-engineering-reuse-pattern-under-the-hood-of-claude-code/, December 2025. (Accessed on 07/16/2026).

Elijah Lo. Introduction to microsoft's new azure openai service. https://www.proserveit.com/blog/introduction-to-microsoft-new-azure-openai-service, Year of Publication. Accessed: 2024-01-04.

Alexandra Sasha Luccioni, Yacine Jernite, and Emma Strubell. Power hungry processing: Watts driving the cost of ai deployment?, 2023. URL https://arxiv.org/abs/2311.16863.

Shengjie Luo, Shanda Li, Tianle Cai, Di He, Dinglan Peng, Shuxin Zheng, Guolin Ke, Liwei Wang, and Tie-Yan Liu. Stable, Fast and Accurate: Kernelized Attention with Relative Positional Encoding, 2021.

Hao Ma, Tianyi Hu, Zhiqiang Pu, Boyin Liu, Xiaolin Ai, Yanyan Liang, and Min Chen. Coevolving with the other you: Fine-tuning llm with sequential cooperative multi-agent reinforcement learning. *arXiv preprint arXiv:2410.06101*, 2024.

Xinyin Ma, Gongfan Fang, and Xinchao Wang. Llm-pruner: On the structural pruning of large language models, 2023.

Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.

Zhao Mandi, Shreeya Jain, and Shuran Song. Roco: Dialectic multi-robot collaboration with large language models, 2023. URL https://arxiv.org/abs/2307.04738.

Sathiya Kumaran Mani, Yajie Zhou, Kevin Hsieh, Santiago Segarra, Trevor Eberl, Eliran Azulai, Ido Frizler, Ranveer Chandra, and Srikanth Kandula. Enhancing network management using code generated by large language models. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, HotNets '23, page 196–204, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400704154. doi:10.1145/3626111.3628183. URL https://doi.org/10.1145/3626111.3628183.

David Marby and Nijiko Yonskai. Pyan3: Offline call graph generator for python 3. https://github.com/davidfraser/pyan.

Petru Martincu. Cloud-computing. https://github.com/Martincu-Petru/Cloud-Computing, 2020.

Ignacio Martinez. privategpt. https://github.com/imartinez/privateGPT, 2023.

Chu Matthew. heapsortcypher. https://github.com/matthew-chu/heapsortcypher.git, 2019.

Patrick McEnroe, Shen Wang, and Madhusanka Liyanage. A survey on the convergence of edge computing and ai for uavs: Opportunities and challenges. *IEEE Internet of Things Journal*, 9(17):15435–15459, 2022. doi:10.1109/JIOT.2022.3176400.

Dave Ver Meer. Number of chatgpt users and key stats (2024). https://www.namepepper.com/chatgpt-users, 2023. Accessed: 2024-01-20.

Yixuan Mei, Yonghao Zhuang, Xupeng Miao, Juncheng Yang, Zhihao Jia, and Rashmi Vinayak. Helix: Serving large language models over heterogeneous gpus and network via max-flow, 2025. URL https://arxiv.org/abs/2406.01566.

Fabian Mentzer, Eirikur Agustsson, Michael Tschannen, Radu Timofte, and Luc Van Gool. Practical full resolution learned lossless image compression. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer Sentinel Mixture Models, 2016.

Meta Engineering Team. Building meta's genai infrastructure, March 2024. URL https://engineering.fb.com/2024/03/12/data-center-engineering/building-metas-genai-infrastructure/.

Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christoforos Nalmpantis, Ram Pasunuru, Roberta Raileanu, Baptiste Rozière, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, et al. Augmented language models: a survey. *arXiv preprint arXiv:2302.07842*, 2023.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Rae Ying Yee Wong, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. SpecInfer: Accelerating Generative LLM Serving with Speculative Inference and Token Tree Verification. *arXiv preprint arXiv:2305.09781*, 2023.

Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. Spotserve: Serving generative large language models on preemptible instances. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 1112–1127, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703850. doi:10.1145/3620665.3640411. URL https://doi.org/10.1145/3620665.3640411.

Microsoft. Microsoft azure cognitive services. Online document https://azure.microsoft.com/en-us/services/cognitive-services, 2022a.

Microsoft. Microsoft azure image tagging. Online document https://docs.microsoft.com/en-us/azure/cognitive-services/computer-vision/concept-tagging-images, 2022b.

Microsoft. Engaging with multimodal models: Gpt-4v in autogen, 2024. URL https://microsoft.github.io/autogen/0.2/docs/notebooks/agentchat_lmm_gpt-4v/. Accessed: 2024-10-25.

Microsoft. What's New in Azure OpenAI Service, 2025. URL https://learn.microsoft.com/en-us/azure/ai-services/openai/whats-new. Accessed: 2025-03-09.

Microsoft AutoGen Project. Agentchat - society of mind, 2024. URL https://microsoft.github.io/autogen/0.2/docs/notebooks/agentchat_society_of_mind/. Accessed: 2024-10-20.

Microsoft Docs. Azure openai service api version lifecycle. https://github.com/MicrosoftDocs/azure-ai-docs/blob/main/articles/ai-services/openai/api-version-deprecation.md, 2025. Accessed: 2025-04-23.

Microsoft Documentation. Enable infiniband on azure virtual machines, 2024. URL https://learn.microsoft.com/en-us/azure/virtual-machines/extensions/enable-infiniband.

Jeremiah Milbauer, Annie Louis, Mohammad Javad Hosseini, Alex Fabrikant, Donald Metzler, and Tal Schuster. LAIT: Efficient Multi-Segment Encoding in Transformers with Layer-Adjustable Interaction. *arXiv preprint arXiv:2305.19585*, 2023.

Paul Mineiro. Online joint fine-tuning of multi-agent flows. *arXiv preprint arXiv:2406.04516*, 2024.

Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. Looking beyond {GPUs} for {DNN} scheduling on {Multi-Tenant} clusters. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 579–596, 2022.

Amirkeivan Mohtashami and Martin Jaggi. Landmark Attention: Random-Access Infinite Context Length for Transformers, 2023.

Manuel Mosquera, Juan Sebastian Pinzón, Yesid Fonseca, Manuel Ríos, Nicanor Quijano, Luis Felipe Giraldo, and Rubén Manrique. Can llm-augmented autonomous agents cooperate? an evaluation of their cooperative capabilities through melting pot. *IEEE Transactions on Artificial Intelligence*, pages 1–10, 2025. doi:10.1109/TAI.2025.3569192.

Jesse Mu, Xiang Lisa Li, and Noah Goodman. Learning to compress prompts with gist tokens. *arXiv preprint arXiv:2304.08467*, 2023.

Ravi Teja Mullapudi, Steven Chen, Keyi Zhang, Deva Ramanan, and Kayvon Fatahalian. Online model distillation for efficient video inference. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 3573–3582, 2019.

Author's Name. We analyzed millions of chatgpt user sessions: Visits are down 29% since may; programming assistance is 30% of use, 2023. URL https://sparktoro.com/blog/we-analyzed-millions-of-chatgpt-user-sessions-visits-are-down-29-since-may-programming-assistance-is-30-of-use/. SparkToro Blog.

Author's Name. Gpt-3 use case examples: Get inspired and go big with gpt-3. https://neoteric.eu/blog/gpt-3-use-case-examples-get-inspired-and-go-big-with-gpt-3/, Year of Publicationa. Accessed: 2024-01-04.

Author's Name. Llms in finance: Bloomberggpt and fingpt - what you need to know. Medium, Year of Publicationb. URL https://12gunika.medium.com/llms-in-finance-bloomberggpt-and-fingpt-what-you-need-to-know-2fdf3af29217.

Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 1–15, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi:10.1145/3341301.3359646. URL https://doi.org/10.1145/3341301.3359646.

Deepak Narayanan, Keshav Santhanam, Peter Henderson, Rishi Bommasani, Tony Lee, and Percy Liang. Cheaply Evaluating Inference Efficiency Metrics for Autoregressive Transformer APIs, 2023.

Crow Nick. sbhacks. https://github.com/connorasmith/SBHacks, 2016.

Armand Nicolicioiu, Eugenia Iofinova, Eldar Kurtic, Mahdi Nikdan, Andrei Panferov, Ilia Markov, Nir Shavit, and Dan Alistarh. Panza: A personalized text writing assistant via data playback and local fine-tuning, 2024. URL https://arxiv.org/abs/2407.10994.

Yixin Nie, Songhe Wang, and Mohit Bansal. Revealing the importance of semantic retrieval for machine reading at scale, 2019.

Rishabh Nimje. Langchain: A powerful tool for local llm execution. https://medium.com/@rishabhnimje123/langchain-a-powerful-tool-for-local-llm-execution-441df81e2ab, 2023. Accessed: 2024-01-04.

Zhaoyang Niu, Guoqiang Zhong, and Hui Yu. A review on the attention mechanism of deep learning. *Neurocomputing*, 452:48–62, 2021.

Antonio Nucci. Large language models in financial services & banking, 2024. URL https://aisera.com/blog/large-language-models-in-financial-services-banking/.

NVIDIA. Multi-process service. https://docs.nvidia.com/deploy/mps/index.html.

NVIDIA. Fastertransformer. https://github.com/NVIDIA/FasterTransformer.git, 2019.

NVIDIA Corporation. Nvidia infiniband solutions, 2024. URL https://www.nvidia.com/en-us/networking/products/infiniband/.

NVIDIA Newsroom. Nvidia spectrum x: The ethernet networking colossus for ai and accelerated computing, 2024. URL https://nvidianews.nvidia.com/news/spectrum-x-ethernet-networking-xai-colossus.

OpenAI. GPT-4 Technical Report, 2023.

OpenAI. Gpt-4 technical report, 2024. URL https://arxiv.org/abs/2303.08774.

Author Name or Organization. Retrieval-augmented generation: Streamlining the creation of intelligent natural language processing models, 2020. URL https://ai.meta.com/blog/retrieval-augmented-generation-streamlining-the-creation-of-intelligent-natural-language-processing-models/. Accessed: 2023.

Matanel Oren, Michael Hassid, Nir Yarden, Yossi Adi, and Roy Schwartz. Transformers are multi-state rnns, 2024. URL https://arxiv.org/abs/2401.06104.

I. V. Oseledets. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33(5): 2295–2317, 2011. doi:10.1137/090752286. URL https://doi.org/10.1137/090752286.

Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling. *arXiv preprint arXiv:1904.01038*, 2019.

Arthi Padmanabhan, Neil Agarwal, Anand Iyer, Ganesh Ananthanarayanan, Yuanchao Shu, Nikolaos Karianakis, Guoqing Harry Xu, and Ravi Netravali. Gemel: Model merging for Memory-Efficient, Real-Time video analytics at the edge. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 973–994, Boston, MA, April 2023. USENIX Association. ISBN 978-1-939133-33-5. URL https://www.usenix.org/conference/nsdi23/presentation/padmanabhan.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym, 2024a. URL https://arxiv.org/abs/2412.21139.

Rui Pan, Xiang Liu, Shizhe Diao, Renjie Pi, Jipeng Zhang, Chi Han, and Tong Zhang. Lisa: Layerwise importance sampling for memory-efficient large language model fine-tuning. *Advances in Neural Information Processing Systems*, 37:57018–57049, 2024b.

Keivalya Pandya and Mehfuza Holia. Automating customer service using langchain: Building custom open-source gpt chatbot for organizations, 2023. URL https://arxiv.org/abs/2310.05421.

Jay H Park, Gyeongchan Yun, M Yi Chang, Nguyen T Nguyen, Seungmin Lee, Jaesik Choi, Sam H Noh, and Young-ri Choi. {HetPipe}: Enabling large {DNN} training on (whimpy) heterogeneous {GPU} clusters through integration of pipelined model parallelism and data parallelism. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 307–321, 2020.

Joon Sung Park, Lindsay Popowski, Carrie Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Social simulacra: Creating populated prototypes for social computing systems. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*, UIST '22, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393201. doi:10.1145/3526113.3545616. URL https://doi.org/10.1145/3526113.3545616.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, UIST '23, New York, NY, USA, 2023a. Association for Computing Machinery. ISBN 9798400701320. doi:10.1145/3586183.3606763. URL https://doi.org/10.1145/3586183.3606763.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior, 2023b.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior, 2023c. URL https://arxiv.org/abs/2304.03442.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. *arXiv preprint arXiv:2311.18677*, 2023a.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Brijesh Warrier, Nithish Mahalingam, and Ricardo Bianchini. Polca: Power oversubscription in llm cloud providers, 2023b. URL https://arxiv.org/abs/2308.12908.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 118–132, 2024. doi:10.1109/ISCA59077.2024.00019.

Timothy Pecoraro. Character ai: A revolution in writing, 2023. URL https://medium.com/@timothypecoraro/character-ai-a-revolution-in-writing-18451d521d7e. Accessed: 2024-11-13.

Andrew Peng, John Allard, and Steven Heidel. Fine-tuning now available for gpt-4o. https://openai.com/index/gpt-4o-fine-tuning/, August 2024. Accessed: 2023-11-11.

Chau Pham, Boyi Liu, Yingxiang Yang, Zhengyu Chen, Tianyi Liu, Jianbo Yuan, Bryan A. Plummer, Zhaoran Wang, and Hongxia Yang. Let models speak ciphers: Multiagent debate through embeddings, 2024. URL https://arxiv.org/abs/2310.06272.

Plant-Watcher. A plant management application. https://github.com/siwasu17/plant-watcher.

Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C. Mogul, Yoshio Turner, and Jose Renato Santos. Elasticswitch: Practical work-conserving bandwidth guarantees for cloud computing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 351–362, New York, NY, USA, 2013a. Association for Computing Machinery. ISBN 9781450320566. doi:10.1145/2486001.2486027. URL https://doi.org/10.1145/2486001.2486027.

Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C. Mogul, Yoshio Turner, and Jose Renato Santos. Elasticswitch: Practical work-conserving bandwidth guarantees for cloud computing. *SIGCOMM Comput. Commun. Rev.*, 43(4):351–362, aug 2013b. ISSN 0146-4833. doi:10.1145/2534169.2486027. URL https://doi.org/10.1145/2534169.2486027.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently Scaling Transformer Inference, 2022.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5, 2023a.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5, 2023b.

Predibase. Predibase finetuning for customer service. https://predibase.com/customer-service-automation, Dec 2023. Accessed: 2024-12-09.

Pranav Putta, Edmund Mills, Naman Garg, Sumeet Motwani, Chelsea Finn, Divyansh Garg, and Rafael Rafailov. Agent q: Advanced reasoning and learning for autonomous ai agents, 2024. URL https://arxiv.org/abs/2408.07199.

PyTorch Contributors. *Distributed Communication Package - torch.distributed*, 2024. URL https://pytorch.org/docs/stable/distributed.html. Accessed: 2024-12-10.

PyTorch Team. torch.cuda.stream — pytorch 2.2 documentation. https://pytorch.org/docs/stable/generated/torch.cuda.Stream.html, 2024. Accessed: 2025-04-25.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative agents for software development. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15174–15186, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi:10.18653/v1/2024.acl-long.810. URL https://aclanthology.org/2024.acl-long.810/.

Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021.

Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation — a KVCache-centric architecture for serving LLM chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 155–170, Santa Clara, CA, February 2025. USENIX Association. ISBN 978-1-939133-45-8. URL https://www.usenix.org/conference/fast25/presentation/qin.

Jiezhong Qiu, Hao Ma, Omer Levy, Scott Wen tau Yih, Sinong Wang, and Jie Tang. Blockwise self-attention for long document understanding, 2020.

Raf. What are tokens and how to count them? https://help.openai.com/en/articles/4936856-what-are-tokens-and-how-to-count-them, 2024.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv e-prints*, 2019.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer, 2023.

Ori Ram, Yoav Levine, Itay Dalmedigos, Dor Muhlgay, Amnon Shashua, Kevin Leyton-Brown, and Yoav Shoham. In-Context Retrieval-Augmented Language Models, 2023.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, page 3505–3506, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379984. doi:10.1145/3394486.3406703. URL https://doi.org/10.1145/3394486.3406703.

Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, et al. MLPerf inference benchmark, 2020. URL https://arxiv.org/abs/1911.02549.

Arsénio Reis, Dennis Paulino, Vitor Filipe, and João Barroso. Using online artificial vision services to assist the blind-an assessment of microsoft cognitive services and google cloud vision. In *World Conference on Information Systems and Technologies*, pages 174–184. Springer, 2018.

Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015. URL https://proceedings.neurips.cc/paper/2015/file/14bfa6bb14875e45bba028a21ed38046-Paper.pdf.

Luka Ribar, Ivan Chelombiev, Luke Hudlass-Galley, Charlie Blake, Carlo Luschi, and Douglas Orr. Sparq attention: Bandwidth-efficient llm inference, 2023.

Jorma Rissanen. Generalized Kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, 20(3):198–203, 1976.

Jorma Rissanen. Arithmetic coding. *IBM Journal of Research and Development*, 23(2): 149–162, 1979.

Jorma Rissanen and G. G. Jr Langdon. Efficient arithmetic coding for data compression. *IEEE transactions on Communications*, 29(6):858–865, 1981.

Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. INFaaS: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411. USENIX Association, July 2021. ISBN 978-1-939133-23-6. URL https://www.usenix.org/conference/atc21/presentation/romero.

Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. Efficient content-based sparse attention with routing transformers, 2020.

Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. Efficient content-based sparse attention with routing transformers. *Transactions of the Association for Computational Linguistics*, 9:53–68, 2021.

Amelie Royer and Christoph H Lampert. Classifier adaptation at prediction time. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1401–1409, 2015.

Ohad Rubin and Jonathan Berant. Long-range Language Modeling with Self-retrieval. *arXiv preprint arXiv:2306.13421*, 2023a.

Ohad Rubin and Jonathan Berant. Long-range language modeling with self-retrieval, 2023b.

Ayesha Saleem. Llm for lawyers, enrich your precedents with the use of ai. Data Science Dojo, July 2023. URL https://datasciencedojo.com/blog/llm-for-lawyers/.

Mansi Sampat. -voice-assistant-for-visually-impaired. https://github.com/Sarveshtg/-Voice-Assistant-for-Visually-Impaired.git, 2019.

Siddharth Samsi, Dan Zhao, Joseph McDonald, Baolin Li, Adam Michaleas, Michael Jones, William Bergeron, Jeremy Kepner, Devesh Tiwari, and Vijay Gadepally. From words to watts: Benchmarking the energy costs of large language model inference, 2023. URL https://arxiv.org/abs/2310.03003.

Anthony Sarah, Sharath Nittur Sridhar, Maciej Szankin, and Sairam Sundaresan. Llama-nas: Efficient neural architecture search for large language models. In *European Conference on Computer Vision*, pages 67–74. Springer, 2024.

Jacob Schmitt. Build & evaluate llm apps with langchain. `https://circleci.com/blog/build-evaluate-llm-apps-with-langchain/`, Year of Publication. Accessed: 2024-01-04.

Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948. doi:10.1002/j.1538-7305.1948.tb01338.x.

Hang Shao, Bei Liu, and Yanmin Qian. One-shot sensitivity-aware mixed sparsity pruning for large language models, 2024.

Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. Cnn features off-the-shelf: an astounding baseline for recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 806–813, 2014.

Haichen Shen, Seungyeop Han, Matthai Philipose, and Arvind Krishnamurthy. Fast video classification via adaptive cascading of deep models. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3646–3654, 2017.

Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 322–337, New York, NY, USA, 2019a. Association for Computing Machinery. ISBN 9781450368735. doi:10.1145/3341301.3359658. URL `https://doi.org/10.1145/3341301.3359658`.

Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 322–337, 2019b.

Weizhou Shen, Chenliang Li, Hongzhan Chen, Ming Yan, Xiaojun Quan, Hehong Chen, Ji Zhang, and Fei Huang. Small LLMs are weak tool learners: A multi-LLM agent. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 16658–16680, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi:10.18653/v1/2024.emnlp-main.929. URL `https://aclanthology.org/2024.emnlp-main.929/`.

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E Gonzalez, et al. High-throughput generative inference of large language models with a single gpu. *arXiv preprint arXiv:2303.06865*, 2023.

Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph E. Gonzalez, and Ion Stoica. Slora: Scalable serving of thousands of lora adapters. In P. Gibbons, G. Pekhimenko, and C. De Sa, editors, *Proceedings of Machine Learning and Systems*, volume 6, pages 296–311, 2024a. URL https://proceedings.mlsys.org/paper_files/paper/2024/file/906419cd502575b617cc489a1a696a67-Paper-Conference.pdf.

Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. Fairness in serving large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 965–988, Santa Clara, CA, July 2024b. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/sheng.

Weijia Shi, Sewon Min, Michihiro Yasunaga, Minjoon Seo, Rich James, Mike Lewis, Luke Zettlemoyer, and Wen-tau Yih. Replug: Retrieval-augmented black-box language models. *arXiv preprint arXiv:2301.12652*, 2023a.

Zijing Shi, Meng Fang, Shunfeng Zheng, Shilong Deng, Ling Chen, and Yali Du. Cooperation on the fly: Exploring language agents for ad hoc teamwork in the avalon game, 2023b.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023. URL https://arxiv.org/abs/2303.11366.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Kurt Shuster, Spencer Poff, Moya Chen, Douwe Kiela, and Jason Weston. Retrieval augmentation reduces hallucination in conversation. *arXiv preprint arXiv:2104.07567*, 2021.

Yifan Song, Weimin Xiong, Xiutian Zhao, Dawei Zhu, Wenhao Wu, Ke Wang, Cheng Li, Wei Peng, and Sujian Li. AgentBank: Towards generalized LLM agents via fine-tuning on 50000+ interaction trajectories. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 2124–2141, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi:10.18653/v1/2024.findings-emnlp.116. URL https://aclanthology.org/2024.findings-emnlp.116/.

Yin Song, Chen Wu, and Eden Duthie. amazon/MistralLite, 2023a. URL https://huggingface.co/amazon/MistralLite.

Yisheng Song, Ting Wang, Puyu Cai, Subrota K. Mondal, and Jyoti Prakash Sahoo. A comprehensive survey of few-shot learning: Evolution, applications, challenges, and opportunities. *ACM Comput. Surv.*, 55(13s), jul 2023b. ISSN 0360-0300. doi:10.1145/3582688. URL https://doi.org/10.1145/3582688.

Krishna Prasad Varadarajan Srinivasan, Prasanth Gumpena, Madhusudhana Yattapu, and Vishal H Brahmbhatt. Comparative analysis of different efficient fine tuning methods of large language models (llms) in low-resource setting. *arXiv preprint arXiv:2405.13181*, 2024.

Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. Dynamollm: Designing llm inference clusters for performance and energy efficiency, 2024. URL https://arxiv.org/abs/2408.00741.

Vighnesh Subramaniam, Yilun Du, Joshua B. Tenenbaum, Antonio Torralba, Shuang Li, and Igor Mordatch. Multiagent finetuning: Self improvement with diverse reasoning chains. In *The Thirteenth International Conference on Learning Representations*, July 2025. URL https://openreview.net/forum?id=JtGPIZpOrz.

Sainbayar Sukhbaatar, Da Ju, Spencer Poff, Stephen Roller, Arthur Szlam, Jason Weston, and Angela Fan. Not all memories are created equal: Learning to forget by expiring, 2021.

624 sun. Dogecoin_musk. https://github.com/sun624/Dogecoin_musk.git, 2019.

Chi Sun, Xipeng Qiu, Yige Xu, and Xuanjing Huang. How to fine-tune bert for text classification? In *China national conference on Chinese computational linguistics*, pages 194–206. Springer, 2019.

Simeng Sun, Kalpesh Krishna, Andrew Mattarella-Micke, and Mohit Iyyer. Do long-range language models actually use long-range context? *arXiv preprint arXiv:2109.09115*, 2021a.

Simeng Sun, Kalpesh Krishna, Andrew Mattarella-Micke, and Mohit Iyyer. Do Long-Range Language Models Actually Use Long-Range Context?, 2021b.

Pavlo Sydorenko. Top 5 applications of large language models (llms) in legal practice. Medium, 2023. URL https://medium.com/jurdep/top-5-applications-of-large-language-models-llms-in-legal-practice-d29cde9c38ef.

Vivienne Sze and Madhukar Budagavi. High Throughput CABAC Entropy Coding in HEVC. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1778–1791, 2012. doi:10.1109/TCSVT.2012.2221526.

Mohammad Reza Taesiri, Finlay Macklon, Yihe Wang, Hengshuo Shen, and Cor-Paul Bezemer. Large language models are pretty good zero-shot video game bug detectors, 2022. URL https://arxiv.org/abs/2210.02506.

Nima Tajbakhsh, Jae Y Shin, Suryakanth R Gurudu, R Todd Hurst, Christopher B Kendall, Michael B Gotway, and Jianming Liang. Convolutional neural networks for medical image analysis: Full training or fine tuning? *IEEE transactions on medical imaging*, 35(5): 1299–1312, 2016.

Microsoft AutoGen Team. Autogen 0.2 documentation - agentchat auto feedback from code execution. https://microsoft.github.io/autogen/0.2/docs/notebooks/agentchat_auto_feedback_from_code_execution, 2024. Accessed: 2024-10-14.

MosaicML NLP Team. Introducing mpt-7b: A new standard for open-source, commercially usable llms, 2023. URL www.mosaicml.com/blog/mpt-7b. Accessed: 2023-05-05.

Zilliz Technology. Gptcache. https://github.com/zilliztech/GPTCache, 2023.

The-Coding-Kid. A smart album application. https://github.com/The-Coding-Kid/888hacks-flask.

Robert Tinn, Hao Cheng, Yu Gu, Naoto Usuyama, Xiaodong Liu, Tristan Naumann, Jianfeng Gao, and Hoifung Poon. Fine-tuning large neural language models for biomedical natural language processing. *Patterns*, 4(4):100729, 2023. ISSN 2666-3899. doi:https://doi.org/10.1016/j.patter.2023.100729. URL https://www.sciencedirect.com/science/article/pii/S2666389923000697.

Erik F. Tjong Kim Sang and Fien De Meulder. Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition. In *Proceedings of the Seventh Conference on Natural Language Learning at HLT-NAACL 2003*, pages 142–147, 2003. URL https://aclanthology.org/W03-0419.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and Efficient Foundation Language Models, 2023.

Szymon Tworkowski, Konrad Staniszewski, Mikołaj Pacek, Yuhuai Wu, Henryk Michalewski, and Piotr Miłoś. Focused transformer: Contrastive training for context scaling. *arXiv preprint arXiv:2307.03170*, 2023.

Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, et al. Unity: Accelerating {DNN} training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 267–284, 2022.

Athanasios Valavanidis. Artificial intelligence (ai) applications. *Department of Chemistry, National and Kapodistrian University of Athens, University Campus Zografou*, 15784, 2023.

Danish Vasan, Mamoun Alazab, Sobia Wassan, Hamad Naeem, Babak Safaei, and Qin Zheng. Imcfn: Image-based malware classification using fine-tuned convolutional neural network architecture. *Computer Networks*, 171:107138, 2020.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

vLLM Production Stack Team. vLLM Production Stack: reference system for k8s-native cluster-wide deployment with community-driven performance optimization. `https://github.com/vllm-project/production-stack`, 2025. GitHub repository, release `vllm-stack-0.1.1`, accessed 21 April 2025.

vLLM Project. AIBrix: Cost-efficient and pluggable infrastructure components for genai inference. `https://github.com/vllm-project/aibrix`, 2025. GitHub repository, release `v0.2.1`, accessed 21 April 2025.

Chengcheng Wan, Shicheng Liu, Sophie Xie, Yifan Liu, Henry Hoffmann, Michael Maire, and Shan Lu. Project Webpage: Accurate Learning for EneRgy and Timeliness in Software System. `https://alert.cs.uchicago.edu/#release`.

Chengcheng Wan, Shicheng Liu, Henry Hoffmann, Michael Maire, and Shan Lu. Are machine learning cloud apis used correctly? In *43th International Conference on Software Engineering (ICSE'21)*, 2021.

Chengcheng Wan, Shicheng Liu, Sophie Xie, Yifan Liu, Henry Hoffmann, Michael Maire, and Shan Lu. Automated testing of software that uses machine learning apis. In *44th International Conference on Software Engineering (ICSE'22)*, 2022.

Dequan Wang, Evan Shelhamer, Shaoteng Liu, Bruno Olshausen, and Trevor Darrell. Tent: Fully test-time adaptation by entropy minimization. In *ICLR*, 2021a.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models, 2023a. URL `https://arxiv.org/abs/2305.16291`.

Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2614–2627, New York, NY, USA, 2021b. Association for Computing Machinery. ISBN 9781450383431. doi:10.1145/3448016.3457550. URL `https://doi.org/10.1145/3448016.3457550`.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents, 2024a. URL `https://arxiv.org/abs/2402.01030`.

Ya Wang, Dongliang He, Fu Li, Xiang Long, Zhichao Zhou, Jinwen Ma, and Shilei Wen. Multi-label classification with label graph superimposing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 12265–12272, 2020.

Yiding Wang, Decang Sun, Kai Chen, Fan Lai, and Mosharaf Chowdhury. Egeria: Efficient dnn training with knowledge-guided layer freezing. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 851–866, New York, NY, USA, 2023b. Association for Computing Machinery. ISBN 9781450394871. doi:10.1145/3552326.3587451. URL https://doi.org/10.1145/3552326.3587451.

Zhenhailong Wang, Xiaoman Pan, Dian Yu, Dong Yu, Jianshu Chen, and Heng Ji. Zemi: Learning zero-shot semi-parametric language models from multiple tasks, 2023c.

Zhuang Wang, Haibin Lin, Yibo Zhu, and T. S. Eugene Ng. Hi-speed dnn training with espresso: Unleashing the full potential of gradient compression with near-optimal usage strategies. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 867–882, New York, NY, USA, 2023d. Association for Computing Machinery. ISBN 9781450394871. doi:10.1145/3552326.3567505. URL https://doi.org/10.1145/3552326.3567505.

Zihao Wang, Bin Cui, and Shaoduo Gan. Squeezeattention: 2d management of kv-cache in llm inference via layer-wise optimal budget, 2024b. URL https://arxiv.org/abs/2404.04793.

Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. Analyticdb-v: A hybrid analytical engine towards query fusion for structured and unstructured data. *Proc. VLDB Endow.*, 13(12):3152–3165, aug 2020. ISSN 2150-8097. doi:10.14778/3415478.3415541. URL https://doi.org/10.14778/3415478.3415541.

Kevin Wei. Retrieval-augmented generation (rag). https://writer.com/blog/retrieval-augmented-generation-rag/, Year of Publication. Accessed: 2024-01-04.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I. Wang. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution, 2025. URL https://arxiv.org/abs/2502.18449.

Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 945–960, Renton, WA, April 2022. USENIX Association. ISBN 978-1-939133-27-4. URL https://www.usenix.org/conference/nsdi22/presentation/weng.

Wikidata. A free and open knowledge base. Online document https://www.wikidata.org/, 2022.

Hansen William. etl. https://github.com/Grusinator/Aander-ETL.git, 2019a.

Hansen William. etl. https://github.com/The-Coding-Kid/888hacks-flask, 2019b.

Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *Commun. ACM*, 30(6):520–540, jun 1987. ISSN 0001-0782. doi:10.1145/214762.214771. URL https://doi.org/10.1145/214762.214771.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Perric Cistac, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-Art Natural Language Processing. pages 38–45. Association for Computational Linguistics, October 2020a. URL https://www.aclweb.org/anthology/2020.emnlp-demos.6.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020b. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/2020.emnlp-demos.6.

BigScience Workshop, :, Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, Jonathan Tow, Alexander M. Rush, Stella Biderman, Albert Webson, Pawan Sasanka Ammanamanchi, Thomas Wang, Benoît Sagot, Niklas Muennighoff, Albert Villanova del Moral, Olatunji Ruwase, Rachel Bawden, Stas Bekman, Angelina McMillan-Major, Iz Beltagy, Huu Nguyen, Lucile Saulnier, Samson Tan, Pedro Ortiz Suarez, Victor Sanh, Hugo Laurençon, Yacine Jernite, Julien Launay, Margaret Mitchell, Colin Raffel, Aaron Gokaslan, Adi Simhi, Aitor Soroa, Alham Fikri Aji, Amit Alfassy, Anna Rogers, Ariel Kreisberg Nitzav, Canwen Xu, Chenghao Mou, Chris Emezue, Christopher Klamm, Colin Leong, Daniel van Strien, David Ifeoluwa Adelani, Dragomir Radev, Eduardo González Ponferrada, Efrat Levkovizh, Ethan Kim, Eyal Bar Natan, Francesco De Toni, Gérard Dupont, Germán Kruszewski, Giada Pistilli, Hady Elsahar, Hamza Benyamina, Hieu Tran, Ian Yu, Idris Abdulmumin, Isaac Johnson, Itziar Gonzalez-Dios, Javier de la Rosa, Jenny Chim, Jesse Dodge, Jian Zhu, Jonathan Chang, Jörg Frohberg, Joseph Tobing, Joydeep Bhattacharjee, Khalid Almubarak, Kimbo Chen, Kyle Lo, Leandro Von Werra, Leon Weber, Long Phan, Loubna Ben allal, Ludovic Tanguy, Manan Dey, Manuel Romero Muñoz, Maraim Masoud, María Grandury, Mario Šaško, Max Huang, Maximin Coavoux, Mayank Singh, Mike Tian-Jian Jiang, Minh Chien Vu, Mohammad A. Jauhar, Mustafa Ghaleb, Nishant Subramani, Nora Kassner, Nurulaqilla Khamis, Olivier Nguyen, Omar Espejel, Ona de Gibert, Paulo Villegas, Peter Henderson, Pierre Colombo, Priscilla Amuok, Quentin Lhoest, Rheza Harliman, Rishi Bommasani, Roberto Luis López, Rui Ribeiro, Salomey Osei, Sampo Pyysalo, Sebastian Nagel,

Shamik Bose, Shamsuddeen Hassan Muhammad, Shanya Sharma, Shayne Longpre, Somaieh Nikpoor, Stanislav Silberberg, Suhas Pai, Sydney Zink, Tiago Timponi Torrent, Timo Schick, Tristan Thrush, Valentin Danchev, Vassilina Nikoulina, Veronika Laippala, Violette Lepercq, Vrinda Prabhu, Zaid Alyafeai, Zeerak Talat, Arun Raja, Benjamin Heinzerling, Chenglei Si, Davut Emre Taşar, Elizabeth Salesky, Sabrina J. Mielke, Wilson Y. Lee, Abheesht Sharma, Andrea Santilli, Antoine Chaffin, Arnaud Stiegler, Debajyoti Datta, Eliza Szczechla, Gunjan Chhablani, Han Wang, Harshit Pandey, Hendrik Strobelt, Jason Alan Fries, Jos Rozen, Leo Gao, Lintang Sutawika, M Saiful Bari, Maged S. Alshaibani, Matteo Manica, Nihal Nayak, Ryan Teehan, Samuel Albanie, Sheng Shen, Srulik Ben-David, Stephen H. Bach, Taewoon Kim, Tali Bers, Thibault Fevry, Trishala Neeraj, Urmish Thakker, Vikas Raunak, Xiangru Tang, Zheng-Xin Yong, Zhiqing Sun, Shaked Brody, Yallow Uri, Hadar Tojarieh, Adam Roberts, Hyung Won Chung, Jaesung Tae, Jason Phang, Ofir Press, Conglong Li, Deepak Narayanan, Hatim Bourfoune, Jared Casper, Jeff Rasley, Max Ryabinin, Mayank Mishra, Minjia Zhang, Mohammad Shoeybi, Myriam Peyrounette, Nicolas Patry, Nouamane Tazi, Omar Sanseviero, Patrick von Platen, Pierre Cornette, Pierre François Lavallée, Rémi Lacroix, Samyam Rajbhandari, Sanchit Gandhi, Shaden Smith, Stéphane Requena, Suraj Patil, Tim Dettmers, Ahmed Baruwa, Amanpreet Singh, Anastasia Cheveleva, Anne-Laure Ligozat, Arjun Subramonian, Aurélie Névéol, Charles Lovering, Dan Garrette, Deepak Tunuguntla, Ehud Reiter, Ekaterina Taktasheva, Ekaterina Voloshina, Eli Bogdanov, Genta Indra Winata, Hailey Schoelkopf, Jan-Christoph Kalo, Jekaterina Novikova, Jessica Zosa Forde, Jordan Clive, Jungo Kasai, Ken Kawamura, Liam Hazan, Marine Carpuat, Miruna Clinciu, Najoung Kim, Newton Cheng, Oleg Serikov, Omer Antverg, Oskar van der Wal, Rui Zhang, Ruochen Zhang, Sebastian Gehrmann, Shachar Mirkin, Shani Pais, Tatiana Shavrina, Thomas Scialom, Tian Yun, Tomasz Limisiewicz, Verena Rieser, Vitaly Protasov, Vladislav Mikhailov, Yada Pruksachatkun, Yonatan Belinkov, Zachary Bamberger, Zdeněk Kasner, Alice Rueda, Amanda Pestana, Amir Feizpour, Ammar Khan, Amy Faranak, Ana Santos, Anthony Hevia, Antigona Unldreaj, Arash Aghagol, Arezoo Abdollahi, Aycha Tammour, Azadeh HajiHosseini, Bahareh Behroozi, Benjamin Ajibade, Bharat Saxena, Carlos Muñoz Ferrandis, Daniel McDuff, Danish Contractor, David Lansky, Davis David, Douwe Kiela, Duong A. Nguyen, Edward Tan, Emi Baylor, Ezinwanne Ozoani, Fatima Mirza, Frankline Ononiwu, Habib Rezanejad, Hessie Jones, Indrani Bhattacharya, Irene Solaiman, Irina Sedenko, Isar Nejadgholi, Jesse Passmore, Josh Seltzer, Julio Bonis Sanz, Livia Dutra, Mairon Samagaio, Maraim Elbadri, Margot Mieskes, Marissa Gerchick, Martha Akinlolu, Michael McKenna, Mike Qiu, Muhammed Ghauri, Mykola Burynok, Nafis Abrar, Nazneen Rajani, Nour Elkott, Nour Fahmy, Olanrewaju Samuel, Ran An, Rasmus Kromann, Ryan Hao, Samira Alizadeh, Sarmad Shubber, Silas Wang, Sourav Roy, Sylvain Viguier, Thanh Le, Tobi Oyebade, Trieu Le, Yoyo Yang, Zach Nguyen, Abhinav Ramesh Kashyap, Alfredo Palasciano, Alison Callahan, Anima Shukla, Antonio Miranda-Escalada, Ayush Singh, Benjamin Beilharz, Bo Wang, Caio Brito, Chenxi Zhou, Chirag Jain, Chuxin Xu, Clémentine Fourrier, Daniel León Periñán, Daniel Molano, Dian Yu, Enrique Manjavacas, Fabio Barth, Florian Fuhrimann, Gabriel Altay, Giyaseddin Bayrak, Gully Burns, Helena U. Vrabec, Imane Bello, Ishani Dash, Jihyun Kang, John Giorgi, Jonas Golde,

Jose David Posada, Karthik Rangasai Sivaraman, Lokesh Bulchandani, Lu Liu, Luisa Shinzato, Madeleine Hahn de Bykhovetz, Maiko Takeuchi, Marc Pàmies, Maria A Castillo, Marianna Nezhurina, Mario Sänger, Matthias Samwald, Michael Cullan, Michael Weinberg, Michiel De Wolf, Mina Mihaljcic, Minna Liu, Moritz Freidank, Myungsun Kang, Natasha Seelam, Nathan Dahlberg, Nicholas Michio Broad, Nikolaus Muellner, Pascale Fung, Patrick Haller, Ramya Chandrasekhar, Renata Eisenberg, Robert Martin, Rodrigo Canalli, Rosaline Su, Ruisi Su, Samuel Cahyawijaya, Samuele Garda, Shlok S Deshmukh, Shubhanshu Mishra, Sid Kiblawi, Simon Ott, Sinee Sang-aroonsiri, Srishti Kumar, Stefan Schweter, Sushil Bharati, Tanmay Laud, Théo Gigant, Tomoya Kainuma, Wojciech Kusa, Yanis Labrak, Yash Shailesh Bajaj, Yash Venkatraman, Yifan Xu, Yingxin Xu, Yu Xu, Zhe Tan, Zhongli Xie, Zifan Ye, Mathilde Bras, Younes Belkada, and Thomas Wolf. BLOOM: A 176B-Parameter Open-Access Multilingual Language Model, 2023.

Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast Distributed Inference Serving for Large Language Models, 2023a.

Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920*, 2023b.

Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. Loongserve: Efficiently serving long-context large language models with elastic sequence parallelism. *arXiv preprint arXiv:2404.09526*, 2024a.

Bingyang Wu, Ruidong Zhu, Zili Zhang, Peng Sun, Xuanzhe Liu, and Xin Jin. dLoRA: Dynamically orchestrating requests and adapters for LoRA LLM serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 911–927, Santa Clara, CA, July 2024b. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/wu-bingyang.

Dekun Wu, Haochen Shi, Zhiyuan Sun, and Bang Liu. Deciphering digital detectives: Understanding llm behaviors and capabilities in multi-agent mystery games, 2023c.

Feijie Wu, Zitao Li, Yaliang Li, Bolin Ding, and Jing Gao. Fedbiot: Llm local fine-tuning in federated learning without full model. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 3345–3355, 2024c.

Haijie Wu, Junhwi Kim, and Vyas Sekar. Improving llm output by combining rag and fine-tuning, May 2024d. URL https://www.conviva.com/blog/improving-llm-output-by-combining-rag-and-finetuning/. Accessed: 2024-11-19.

Pengying Wu, Yao Mu, Kangjie Zhou, Ji Ma, Junting Chen, and Chang Liu. Camon: Cooperative agents for multi-object navigation with llm-based conversations, 2024e. URL https://arxiv.org/abs/2407.00632.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White,

Doug Burger, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *First Conference on Language Modeling*, 2024f. URL https://openreview.net/forum?id=BAakY1hNKS.

Yongji Wu, Matthew Lentz, Danyang Zhuo, and Yao Lu. Serving and optimizing machine learning workflows on heterogeneous infrastructures, 2022a.

Yuhuai Wu, Markus Norman Rabe, DeLesley Hutchins, and Christian Szegedy. Memorizing transformers. In *International Conference on Learning Representations*, 2022b. URL https://openreview.net/forum?id=TrjbxzRcnf-.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. *Science China Information Sciences*, 68(2):121101, 2025.

Wenhan Xia, Hongxu Yin, and Niraj K. Jha. Efficient synthesis of compact deep neural networks: invited. In *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*, DAC '20. IEEE Press, 2020. ISBN 9781450367257.

Yifei Xia, Fangcheng Fu, Wentao Zhang, Jiawei Jiang, and Bin Cui. Efficient multi-task llm quantization and serving for multiple lora adapters. *Advances in Neural Information Processing Systems*, 37:63686–63714, 2024.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*, pages 38087–38099. PMLR, 2023.

Guangxuan Xiao, Jiaming Tang, Jingwei Zuo, Junxian Guo, Shang Yang, Haotian Tang, Yao Fu, and Song Han. Duoattention: Efficient long-context llm inference with retrieval and streaming heads, 2024a. URL https://arxiv.org/abs/2410.10819.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks, 2024b. URL https://arxiv.org/abs/2309.17453.

Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, 2018.

Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. {AntMan}: Dynamic scaling on {GPU} clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548, 2020.

Shuzhao Xie, Yuan Xue, Yifei Zhu, and Zhi Wang. Cost effective mlaas federation: A combinatorial reinforcement learning approach. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2022.

Wenhan Xiong, Hong Wang, and William Yang Wang. Progressively pretrained dense corpus index for open-domain question answering. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 2803–2815, Online, April 2021. Association for Computational Linguistics. doi:10.18653/v1/2021.eacl-main.244. URL https://aclanthology.org/2021.eacl-main.244.

Peng Xu, Wei Ping, Xianchao Wu, Lawrence McAfee, Chen Zhu, Zihan Liu, Sandeep Subramanian, Evelina Bakhturina, Mohammad Shoeybi, and Bryan Catanzaro. Retrieval meets long context large language models, 2024a.

Xinrun Xu, Yuxin Wang, Chaoyi Xu, Ziluo Ding, Jiechuan Jiang, Zhiming Ding, and Börje F. Karlsson. A survey on game playing agents and large models: Methods, applications, and challenges, 2024b. URL https://arxiv.org/abs/2403.10249.

Keiji Yanai and Yoshiyuki Kawano. Food image recognition using deep convolutional network with pre-training and fine-tuning. In *2015 IEEE International Conference on Multimedia & Expo Workshops (ICMEW)*, pages 1–6. IEEE, 2015.

Wei Yang, Yuqing Xie, Aileen Lin, Xingyu Li, Luchen Tan, Kun Xiong, Ming Li, and Jimmy Lin. End-to-end open-domain question answering with. In *Proceedings of the 2019 Conference of the North*. Association for Computational Linguistics, 2019. doi:10.18653/v1/n19-4013. URL https://doi.org/10.18653%2Fv1%2Fn19-4013.

Yingrui Yang, Yifan Qiao, Jinjin Shao, Xifeng Yan, and Tao Yang. Lightweight composite re-ranking for efficient keyword search with bert. In *Proceedings of the Fifteenth ACM International Conference on Web Search and Data Mining*, WSDM '22, page 1234–1244, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391320. doi:10.1145/3488560.3498495. URL https://doi.org/10.1145/3488560.3498495.

Yingxuan Yang, Qiuying Peng, Jun Wang, Ying Wen, and Weinan Zhang. Llm-based multi-agent systems: Techniques and business perspectives. *arXiv preprint arXiv:2411.14033*, 2024.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii, editors, *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2369–2380, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi:10.18653/v1/D18-1259. URL https://aclanthology.org/D18-1259/.

Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. Cacheblend: Fast large language model serving with cached knowledge fusion. *arXiv preprint arXiv:2405.16444*, 2024a.

Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. Cacheblend: Fast large language model serving for rag with cached knowledge fusion. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 94–109, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400711961. doi:10.1145/3689031.3696098. URL https://doi.org/10.1145/3689031.3696098.

Kai Yao, Penglei Gao, Lichun Li, Yuan Zhao, Xiaofeng Wang, Wei Wang, and Jianke Zhu. Layer-wise importance matters: Less memory for better performance in parameter-efficient fine-tuning of large language models. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 1977–1992, Miami, Florida, USA, November 2024b. Association for Computational Linguistics. doi:10.18653/v1/2024.findings-emnlp.109. URL https://aclanthology.org/2024.findings-emnlp.109/.

Yuanshun Yao, Zhujun Xiao, Bolun Wang, Bimal Viswanath, Haitao Zheng, and Ben Y Zhao. Complexity vs. performance: empirical analysis of machine learning as a service. In *Proceedings of the 2017 Internet Measurement Conference*, pages 384–397, 2017.

Rui Ye, Xiangrui Liu, Qimin Wu, Xianghe Pang, Zhenfei Yin, Lei Bai, and Siheng Chen. X-mas: Towards building multi-agent systems with heterogeneous llms. *arXiv preprint arXiv:2505.16997*, 2025.

Tong Ye, Lingfei Wu, Tengfei Ma, Xuhong Zhang, Yangkai Du, Peiyu Liu, Wenhai Wang, and Shouling Ji. Tram: A token-level retrieval-augmented mechanism for source code summarization, 2023.

Catherine Yeh, Yida Chen, Aoyu Wu, Cynthia Chen, Fernanda Viégas, and Martin Wattenberg. Attentionviz: A global view of transformer attention. *IEEE Transactions on Visualization and Computer Graphics*, 30(1):262–272, January 2024. ISSN 1077-2626. doi:10.1109/TVCG.2023.3327163. URL https://doi.org/10.1109/TVCG.2023.3327163.

Rongjie Yi, Liwei Guo, Shiyun Wei, Ao Zhou, Shangguang Wang, and Mengwei Xu. EdgeMoE: Fast On-Device Inference of MoE-based Large Language Models. *arXiv preprint arXiv:2308.14352*, 2023.

Chunxing Yin, Bilge Acun, Carole-Jean Wu, and Xing Liu. Tt-rec: Tensor train compression for deep learning recommendation models. *Proceedings of Machine Learning and Systems*, 3:448–462, 2021.

Da Yin, Faeze Brahman, Abhilasha Ravichander, Khyathi Chandu, Kai-Wei Chang, Yejin Choi, and Bill Yuchen Lin. Agent lumos: Unified and modular training for open-source language agents. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12380–12403, Bangkok, Thailand, August 2024. Association for

Computational Linguistics. doi:10.18653/v1/2024.acl-long.670. URL https://aclanthology.org/2024.acl-long.670.

Jaehong Yoon, Eunho Yang, Jeongtae Lee, and Sung Ju Hwang. Lifelong learning with dynamically expandable networks. *arXiv preprint arXiv:1708.01547*, 2017.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022.

Lingfan Yu, Jinkun Lin, and Jinyang Li. Stateful large language model serving with pensieve. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 144–158, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400711961. doi:10.1145/3689031.3696086. URL https://doi.org/10.1145/3689031.3696086.

Peifeng Yu and Mosharaf Chowdhury. Salus: Fine-grained GPU sharing primitives for deep learning applications. *CoRR*, abs/1902.04610, 2019. URL http://arxiv.org/abs/1902.04610.

Shengbin Yue, Wei Chen, Siyuan Wang, Bingxuan Li, Chenchen Shen, Shujun Liu, Yuxuan Zhou, Yao Xiao, Song Yun, Xuanjing Huang, et al. Disc-lawllm: Fine-tuning large language models for intelligent legal services. *arXiv preprint arXiv:2309.11325*, 2023.

Xiang Yue, Tianyu Zheng, Ge Zhang, and Wenhu Chen. Mammoth2: Scaling instructions from the web. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=yVu5dnPlqA.

Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big Bird: Transformers for Longer Sequences, 2021.

Lin Zehui, Pengfei Liu, Luyao Huang, Junkun Chen, Xipeng Qiu, and Xuanjing Huang. Dropattention: A regularization method for fully-connected self-attention networks, 2019.

Xiaohua Zhai, Alexander Kolesnikov, Neil Houlsby, and Lucas Beyer. Scaling vision transformers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12104–12113, 2022.

Ceyao Zhang, Kaijie Yang, Siyi Hu, Zihao Wang, Guanghe Li, Yihang Sun, Cheng Zhang, Zhaowei Zhang, Anji Liu, Song-Chun Zhu, Xiaojun Chang, Junge Zhang, Feng Yin, Yitao Liang, and Yaodong Yang. Proagent: Building proactive cooperative agents with large language models. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(16): 17591–17599, Mar. 2024a. doi:10.1609/aaai.v38i16.29710. URL https://ojs.aaai.org/index.php/AAAI/article/view/29710.

Chaoyun Zhang, Zicheng Ma, Yuhao Wu, Shilin He, Si Qin, Minghua Ma, Xiaoting Qin, Yu Kang, Yuyi Liang, Xiaoyu Gou, Yajie Xue, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Qi Zhang. Allhands: Ask me anything on large-scale verbatim feedback via large language models, 2024b. URL https://arxiv.org/abs/2403.15157.

Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. {MArk}: Exploiting cloud services for {Cost-Effective},{SLO-Aware} machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1049–1062, 2019.

Feiyu Zhang, Liangzhi Li, Junhao Chen, Zhouqiang Jiang, Bowen Wang, and Yiming Qian. Increlora: Incremental parameter allocation method for parameter-efficient fine-tuning, 2023a. URL https://arxiv.org/abs/2308.12043.

Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. {SHEPHERD}: Serving {DNNs} in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 787–808, 2023b.

Hongxin Zhang, Weihua Du, Jiaming Shan, Qinhong Zhou, Yilun Du, Joshua B. Tenenbaum, Tianmin Shu, and Chuang Gan. Building cooperative embodied agents modularly with large language models. In *ICLR* con [2024]. URL https://openreview.net/forum?id=EnXJfQqy0K.

Mingyang Zhang, Hao Chen, Chunhua Shen, Zhen Yang, Linlin Ou, Xinyi Yu, and Bohan Zhuang. Loraprune: Structured pruning meets low-rank parameter-efficient fine-tuning, 2024d. URL https://arxiv.org/abs/2305.18403.

Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. {DeepCPU}: Serving {RNN-based} deep learning models 10x faster. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 951–965, 2018.

Qingru Zhang, Minshuo Chen, Alexander Bukharin, Nikos Karampatziakis, Pengcheng He, Yu Cheng, Weizhu Chen, and Tuo Zhao. Adalora: Adaptive budget allocation for parameter-efficient fine-tuning, 2023c. URL https://arxiv.org/abs/2303.10512.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. OPT: Open Pre-trained Transformer Language Models, 2022.

Xuan Zhang, Cunxiao Du, Chao Du, Tianyu Pang, Wei Gao, and Min Lin. Simlayerkv: A simple framework for layer-level kv cache reduction, 2024e. URL https://arxiv.org/abs/2410.13846.

Xuchao Zhang, Menglin Xia, Camille Couturier, Guoqing Zheng, Saravan Rajmohan, and Victor Ruhle. Hybrid retrieval-augmented generation for real-time composition assistance, 2023d.

Yang Zhang, Shixin Yang, Chenjia Bai, Fei Wu, Xiu Li, Zhen Wang, and Xuelong Li. Towards efficient llm grounding for embodied multi-agent collaboration. *arXiv preprint arXiv:2405.14314*, 2024f.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Re, Clark Barrett, Zhangyang Wang, and Beidi Chen. H2o: Heavy-hitter oracle for efficient generative inference of large language models. In *Workshop on Efficient Systems for Foundation Models at ICML*, 2023e. URL `https://openreview.net/forum?id=ctPizehA9D`.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. H_2o: Heavy-hitter oracle for efficient generative inference of large language models, 2023f.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. H_2o: Heavy-hitter oracle for efficient generative inference of large language models, 2023g. URL `https://arxiv.org/abs/2306.14048`.

Qibin Zhao, Guoxu Zhou, Shengli Xie, Liqing Zhang, and Andrzej Cichocki. Tensor Ring Decomposition, 2016.

Qingfei Zhao, Ruobing Wang, Yukuo Cen, Daren Zha, Shicheng Tan, Yuxiao Dong, and Jie Tang. LongRAG: A dual-perspective retrieval-augmented generation paradigm for long-context question answering. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 22600–22632, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi:10.18653/v1/2024.emnlp-main.1259. URL `https://aclanthology.org/2024.emnlp-main.1259/`.

Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Joseph E Gonzalez, et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. *arXiv preprint arXiv:2201.12023*, 2022.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric. P Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena, 2023a.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Efficiently programming large language models using sglang. *arXiv preprint arXiv:2312.07104*, 2023b.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs, 2024. URL `https://arxiv.org/abs/2312.07104`.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving, 2024a.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, Santa Clara, CA, July 2024b. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin.

Xizhou Zhu, Yuntao Chen, Hao Tian, Chenxin Tao, Weijie Su, Chenyu Yang, Gao Huang, Bin Li, Lewei Lu, Xiaogang Wang, Yu Qiao, Zhaoxiang Zhang, and Jifeng Dai. Ghost in the minecraft: Generally capable agents for open-world environments via large language models with text-based knowledge and memory, 2023. URL https://arxiv.org/abs/2305.17144.

Yuqi Zhu, Shuofei Qiao, Yixin Ou, Shumin Deng, Ningyu Zhang, Shiwei Lyu, Yue Shen, Lei Liang, Jinjie Gu, and Huajun Chen. Knowagent: Knowledge-augmented planning for llm-based agents, 2024. URL https://arxiv.org/abs/2403.03101.

Luisa Zintgraf, Kyriacos Shiarli, Vitaly Kurin, Katja Hofmann, and Shimon Whiteson. Fast context adaptation via meta-learning. In *International Conference on Machine Learning*, pages 7693–7702. PMLR, 2019.