

# LLM as a System Service on Mobile Devices

Wangsong Yin<sup>◆</sup>, Mengwei Xu<sup>◇</sup>, Yuanchun Li<sup>★</sup>, Xuanzhe Liu<sup>◆</sup>

<sup>◆</sup>Key Lab of High Confidence Software Technologies (Peking University), Beijing, China

<sup>★</sup>Institute for AI Industry Research (AIR), Tsinghua University, Beijing, China

<sup>◇</sup>State Key Laboratory of Networking and Switching Technology (BUPT), Beijing, China

yws@stu.pku.edu.cn

mwx@bupt.edu.cn

liyuanchn@air.tsinghua.edu.cn

liuxuanzhe@pku.edu.cn

## ABSTRACT

Being more powerful and intrusive into user-device interactions, LLMs are eager for on-device execution to better preserve user privacy. In this work, we propose a new paradigm of mobile AI: LLM as a system service on mobile devices (LLMaaS). Unlike traditional DNNs that execute in a stateless manner, such a system service is stateful: LLMs execution often needs to maintain persistent states (mainly KV cache) across multiple invocations. To minimize the LLM context switching overhead under tight device memory budget, this work presents LLMS, which decouples the memory management of app and LLM contexts with a key idea of fine-grained, chunk-wise, globally-optimized KV cache compression and swapping. By fully leveraging KV cache’s unique characteristics, it proposes three novel techniques: (1) Tolerance-Aware Compression: it compresses chunks based on their measured accuracy tolerance to compression. (2) IO-Recompute Pipelined Loading: it introduces recompute to swapping-in for acceleration. (3) Chunk Lifecycle Management: it optimizes the memory activities of chunks with an ahead-of-time swapping-out and an LCTRU (Least Compression-Tolerable and Recently-Used) queue based eviction. In evaluations conducted on well-established traces and various edge devices, LLMS reduces context switching latency by up to 2 orders of magnitude when compared to competitive baseline solutions.

## 1 INTRODUCTION

The recent progress of Large Language Models (LLMs) is reshaping the mobile AI landscape. LLMs can comprehend human language and handle most (if not all) language-based ML tasks with a huge knowledge base, e.g., language translation [27, 63], Q&A [32, 57], and smart reply [3]. More importantly, it catalyzes novel mobile applications such as UI automation tasks based on user instructions, e.g., “forward the recent 5 emails from Bob to Alice” [66]. In a nutshell, LLM marks a giant step for mobile devices towards more intelligent and personalized assistive agent [47].

Being more powerful and intrusive into user-device interactions, LLMs are eager for on-device execution to better

preserve user privacy. For instance, mobile UI automation task takes in the screen information (either in view hierarchy [64, 66] or pixels [42, 58, 76]), which could contain highly privacy-sensitive information like chatting history, photos, and textual input. Beyond privacy, on-device LLM also alleviates the huge resource intensity on datacenters and guarantees functionality availability with weak network.

Indeed, there have been tremendous progress made towards on-device LLM in the past year. On the one hand, more compact and resource-efficient LLMs (e.g., Gemma-2B and Falcon-1B [25, 35]) are released, and various compression algorithms (e.g., 4-bit or even 1-bit quantization [34, 49]) are proposed to effectively cut down the resource consumption of LLMs. On the other hand, system- and hardware-level support for LLMs are maturing [10, 59, 73, 75]. For instance, Qualcomm claims Snapdragon 8gen3’s NPU to be “meticulously designed with generative AI in mind”, capable of executing LLM at 20 tokens/second [20]. Consequently, smartphone vendors like Google [9] are exploring to built-in LLM into their off-the-shelf devices.

**LLM-as-a-Service** (LLMaaS). In this work, we propose a new paradigm of mobile AI: *LLM as a system service on mobile devices* (LLMaaS). It indicates that, the mobile OS exposes an LLM and its inference infrastructure as a system feature to mobile apps, akin to the location or notification services. The interface between apps and LLM Service is based on prompts in nature language. This paradigm fundamentally differs from prior arts that apps own their models separately, into which the OS has no visibility. Such a paradigm shift is natural in LLM era, motivated by following observations: (1) LLM has world knowledge and can support generic ML tasks [23, 29, 40, 62] through properly curated or even learned prompts. (2) LLMaaS needs only one copy of LLM weights in memory, regardless of how intensively the LLM is used across apps; otherwise, the LLMs owned by different apps easily blow up the device memory; (3) A system-level LLM can be better customized for on-device accelerator and enjoy the performance gain over commodity hardware. An exemplification of LLMaaS paradigm is the

recently released Android AICore [1], a standalone LLM system service that is already in use by several Google apps for on-device summarization and Gboard smart reply.

To fulfill the vision of LLMAaaS, this work identifies and tackles a unique system challenge: *LLM context management*. Specifically, unlike traditional DNNs that execute in a stateless manner, LLMs execution often needs to maintain persistent states (mainly KV cache [56]) across multiple invocations. For example, a smart reply app [3] needs to remember its historical conversation text to generate more accurate reply suggestions compared to using only the last message. According to our preliminary experiments in §2, a single LLM context could consume significant device memory (e.g., 2GB for Llama2-7B with 4k context window size); more of such LLM contexts soon dominate the memory usage of LLM Service. Consequently, how to properly manage the persistent LLM contexts across apps becomes crucial to improve the quality-of-LLM-service. One might treat the LLM context memory as part of the app memory and reuse the mobile memory manager (e.g., low memory killer [2]) to manage them in a unified manner. This approach, however, is inefficient due to the unique characteristics of LLM contexts as will be demonstrated in §2.3.

**LLMS** This work presents a first-of-its-kind system towards LLMAaaS on mobile devices, named LLMS, that decouples the memory management of LLM contexts from the app. LLMS aims to minimize the LLM context switching overhead under tight memory budget, akin to the traditional mobile memory mechanisms that focus on reducing the app cold-start latency [26, 55, 77]. To alleviate the limited device memory, LLMS introduces novel techniques on fine-grained, chunk-wise, globally-optimized KV cache compression and swapping. LLMS splits KV cache into series of chunks – memory blocks covering the same number of tokens (all layers in one token). Each chunk is compressed and swapped out/in independently. In practice, the chunk size is determined empirically based on the configurations of system and LLM, e.g., 16 tokens. Similar to the OS’s paging mechanism, we observe that the idea of chunking strikes good balance between the utilization of device memory and I/O bandwidth, and outperforms token-level or context-level management.

Chunking fully leverages the unique characteristics of KV cache for optimizing context switching. (1) KV cache is easy to be chunked. As shown in §3.1, its layout grows at the token dimension that can be divided into chunks with flexible granularity. (2) KV cache is unevenly tolerant to compression. A portion of KV cache can be compressed more aggressively. Compressing in chunk-level allows for maximizing the compression potential of KV cache. (3) KV cache can be recomputed. As an intermediate activation of LLM, KV cache can be recomputed from the prompt text to recover it into memory. By managing KV cache in chunks, during

context switching, chunks that need to be loaded from disk can be concurrently recovered into memory through both recompute and I/O, thereby fully utilizing hardware.

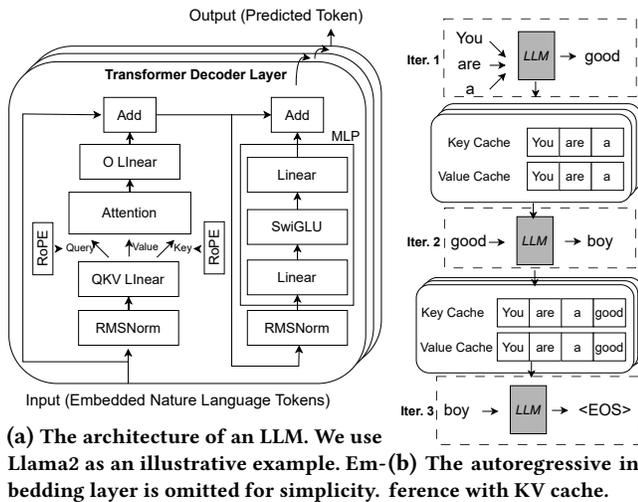
To fully explore the design space of chunk-level memory management, LLMS incorporates three novel techniques.

(1) Tolerance-Aware Compression (§3.2). LLMS uses the information density of a chunk as a metric to quantify its tolerance to compression. It calculates the information density based on attention scores, which indicate the level of attention that the tokens pay to each other. The rationale is that a token that attracts more “attention” from other tokens is more likely to be informative. LLMS then judiciously determines the compression rate for each chunk to maximize the overall information intensity of a context, while meeting a global average compression ratio configured by the OS.

(2) Swapping-Recompute Pipeline (§3.3). LLMS recomputes chunks from their original text and overlaps the computation time with the I/O time of other chunks through pipeline. However, chunks can be swapped independently, while the LLM’s continuous position encoding and causal mask cannot handle recompute of interleaved chunks. Thereby, LLMS devises the encoding/mask to fit the interleaved chunks on the fly.

(3) Chunk Lifecycle Management (§3.4). LLMS designs the lifecycle of KV cache chunks to be more friendly to context switching. Regarding which chunk to swap out, it employs an LCTRU Least Compression-Tolerable and Recently Used queue to determine the eviction priority. Its rationale is that swapping heavy and least recent used chunks out to disk can better leverage chunks’ time locality and LLMS’s swapping-recompute pipeline. Regarding when to swap out, it adopts an ahead-of-time swapping-out approach to hide the time for reclaiming memory during context switching.

**Results** We have fully implemented LLMS on Commercial Off-The-Shelf (COTS) devices including Jetson Orin NX [7], Jetson TX2 [8], and MI14 smartphone [15] with Llama2-7B [62] and OPT-7B [83]. We then evaluate its performance with a 72-hours-long context switching traces synthesized from 6 representative datasets. The results show that LLMS significantly outperforms its baselines. It reduces switching latency by up to 2 orders of magnitude compared to managing contexts via the conventional app-level memory manager low-memory killer [2], or directly managing contexts via disk swapping. Compared to the state-of-the-art chunk-based context managing system vLLM [45] with statically quantizing all chunks to 8-bits [70], LLMS achieves up to 20× and on average 9.7× switching latency reduction. LLMS achieves the aforementioned latency reduction without any noticeable accuracy loss on 6 datasets. For the first time, LLMS addresses the issue of LLM context switching, enabling LLMAaaS to provide low switching-latency and stateful services for mobile apps.



**Figure 1: Illustrations for LLM’s representative architecture and inference procedure.**

**Contributions** This work makes following contributions.

- We paint a picture of LLM as a system service (LLMaaS) to fully leash the power of LLM on devices, presenting its strong motivations as well as the key challenge of LLM context management in memory.
- We flesh out LLMaaS with LLMS, a concrete LLM Service design based on fine-grained, chunk-wise KV cache compression and swapping. LLMS aims to maximize the context switching speed through a set of novel techniques, including tolerance-aware compression, swapping-recompute pipeline, and strategic chunk lifecycle management.
- We prototype LLMS and comprehensively evaluate its performance on COTS mobile devices and typical LLMs. The results demonstrate the efficacy of LLMS compared to competitive baselines.

## 2 LLM-AS-A-SERVICE: MOTIVATIONS AND CHALLENGES

### 2.1 On-Device Large Language Model

**Large language models.** Large Language Models (LLMs), such as GPT4 [23], Llama2 [62], Gemini [60], etc., are transforming mobile AI. Many cutting-edge applications are empowered by LLM, encompassing agent-based UI automation [65, 67], app built-in chatbot [13], smart voice assistant [11], automated email writer [5], etc.

We briefly introduce the model architecture and inference procedure of LLMs. The main body of LLMs is a series of stacked transformer [63] decoder layers shown in Figure 1a, where input tokens (natural language word pieces) are processed and new output tokens are predicted. Its key component is the attention mechanism, where tokens are mapped to Query, Key and Value tensors to compute the cross-token

connections. LLMs perform inference in an autoregressive manner: in each iteration, it predicts a new token by history tokens (i.e., prompted tokens and generated tokens). Specifically, LLM caches the Key and Value tensor of previous tokens, known as KV Cache [56], to avoid repeated computations. We show such an inference procedure in Figure 1b. In iteration 1, the LLM is prompted by “You are a”. The model predicts a new token “good”, and saves the KV cache “You are a”. In iteration 2, a new token “boy” is jointly predicted by input token ‘good’ and KV cache “You are a”. Meanwhile, the KV cache “good” is also saved. In iteration 3, an “<EOS>” token is predicted by input token ‘boy’ and KV cache “You are a good”, and the LLM inference ends.

**On-device LLM.** A growing demand is to deploy LLMs on devices to support privacy-preserving mobile AI. Taking agent-based UI automation as an example, it takes screen UI as input [66], which is extremely privacy-sensitive as it might include chat history, photos and account information rendered during UI operation. On-device LLM alleviates such privacy concerns as no data leaves devices. Moreover, on-device LLM improves service availability and cost efficiency without relying on network and expensive cloud GPUs.

Recently, remarkable progress has been made towards the fast and energy-efficient on-device inference of LLMs, encompassing a full-stack optimization from algorithm to hardware [10, 20, 59, 73, 75]. For example, mobile SoC vendors have already provided off-the-shelf hardware support for LLMs. For instance, Qualcomm reports Snapdragon 8gen3’s NPU (an ASIC specialized for DNN inference) to be “meticulously designed with generative AI in mind”, capable of executing LLM at 20 tokens/second [20].

### 2.2 On-Device LLM as a Mobile OS Service

We envision a major paradigm shift of mobile AI when on-device LLM matures: *LLM as a system service on mobile devices (LLMaaS)*. It indicates that, the mobile OS exposes an LLM (as well as the inference infrastructure) as a system feature to apps for use, just like location and notification services, instead of each app owning an LLM individually. This vision is supported by a recent survey on mobile agent where many industry experts explicitly call for OS-level LLM [47]. Meanwhile, Google has recently released a preview of such LLMaaS design, named AICore, as an Android system service [1]. The service is already in use by several Google products, such as voice recorder and smart reply. The interface between apps and LLM Service is text in nature language: app sends prompts to LLM Service as LLM inputs; LLM Service sends back tokens generated during LLM autoregressive inference. §3.1 will show a concrete API design of LLM Service.

Such a vision is backed up by following observations.

- **LLMaaS is feasible.** LLM has world knowledge and can support generic ML tasks [23, 29, 40, 62]; in-context learning [33, 52] and PEFT [43, 50] are capable of further enhancing LLM’s capability on downstream tasks. A recent study [81] shows that a 7B-sized LLM can achieve better accuracy on most mobile AI tasks compared to the non-LLM models.
- **LLMaaS is desirable.** Sharing one LLM across apps guarantees affordable, mostly static memory footprint; otherwise, the LLMs owned by each app easily blow up the device memory. The apps might load the LLMs on demand to save memory, yet the weights I/O time easily overwhelms the token generation process: loading LLaMA-7B (4-bit) takes 4.76 seconds, which equals the time to generating 95 tokens on MI 14 smartphone.
- **LLMaaS facilitates hardware and OS design.** On one hand, LLMaaS facilitates the accelerator design on mobile SoCs, since the LLM architecture becomes static and determinable by device vendors. A recent paper [81] shows that only the most common 13% of operators can be fully executed on mobile NPUs. On the other hand, LLMaaS grants the full visibility of LLM execution to OS, who thus can schedule, batch, and cache-reuse the inference requests from different apps for better energy efficiency or throughput [36, 80].

### 2.3 LLMaaS Context Managing

A crucial system challenge this work identifies and tackles is *how to efficiently manage the LLM contexts*. When an app uses LLM Service, it maintains a stateful LLM context, including mainly the KV cache as aforementioned. Notably, long model contexts are important to LLM-powered mobile apps to provide customized and stateful functionality. With a long context, the mobile app can define more complex tasks at a time and augment each task with more information so that it can generate more accurate and personalized output.

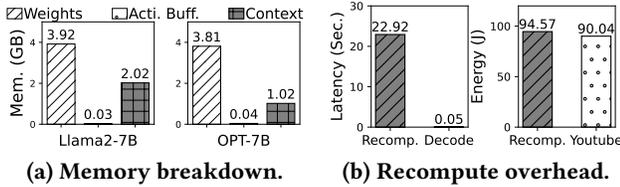
**Observation#1: LLM contexts are memory-intensive.** Since memory is a scarce resource in executing LLMs [24, 59, 79], we study the memory footprint of typical device-affordable LLMs and break down it into three categories: LLM weights, activation buffers, and contexts. As shown in Figure 2a, the model contexts contribute significantly to the memory footprint. For instance, a single context with the maximal context window (4k tokens) of Llama2-7b consumes over 2GB memory, which is over 50% of model weights. Note that the LLM weights are shared across all LLM invocations (thereby static), yet the memory usage of LLM contexts proportionally scales with the number of active contexts and longer context window (e.g., up to 128k for recent LLMs [6]). Consequently, the LLM context is more likely to be the bottleneck in LLM Service and needs to be carefully managed.

**Observation#2: LLM contexts often need to be persistent.** In this work, we use the term “persistence” to indicate that an LLM context needs to be memorized across multiple invocations that could span hours or even days, regardless of whether apps being switched to background or even killed. For example, a multi-round conversation agent needs to remember the historical input/output (though with a maximal context window size) whenever it is used, akin to ChatGPT or any other web-based chatting bots. Besides, an UI automation agent needs to memorize its history operations to serve users’ further interactions, such as “order a pizza in the same restaurant as yesterday”. Furthermore, the smart reply of Google Gboard calls LLM Service to realize “generating reply suggestions based on the full context of a conversation, not just a single message.” [3]. Such persistence feature fundamentally differentiates LLM from prior deep learning models like CNN whose each invocation is independent.

Without system-level persistence support in LLM Service, the apps could instead remember the LLM input/output and feeding them to LLM to “replay” (or recompute) the whole LLM context at each invocation. This approach, however, not only incurs extra programming efforts to developers, but are also highly inefficient on mobile devices. We conducted preliminary experiments to showcase the resource cost of such context recomputing. As shown in Figure 2b, recomputing a Llama2-7B’s context takes 22.92 seconds on MI14, while decoding a token for users only takes 0.05 seconds. The energy consumption of one recompute is roughly equivalent to watching one minute of YouTube video.

**Observation#3: the conventional app-level memory management is not satisfactory.** One intuitive and plausible design is to account the LLM context memory as part of the whole app’s memory who actually uses the LLMaaS, and rely on the app memory manager to manage them as a whole. For example, mobile devices typically employ low-memory killer (LMK) [2] that kills background apps and frees their memory when system goes out of physical memory. In this approach, the app memory and LLMaaS context memory are managed together without differentiating them, e.g., both get released if either of them is too large when out of memory.

However, we find this approach not efficient, since the context memory and app memory are inherently different in following aspects. (1) LLM contexts are more expensive in terms of time/energy expenditure to obtain. Constructing a context memory takes much time and energy, as this procedure is via LLM inference. As shown in Figure 2b, killing an app and recomputing its KV cache incur huge time and energy overhead (e.g., 22.92 seconds and 94.57 J). (2) LLM contexts are relatively cold, e.g., used in low frequency, even when LLM becomes an indispensable feature on mobile devices. For instance, Gboard only invokes LLM Service for



**Figure 2: Experiments results on MI14. Experiments in (a) is performed on Llama.cpp framework [10]. “re-comp.” in (b) means recompute; “Decode” means generating a token; “Youtube” means watching videos on YouTube for 1 minute.**

Interface	Descriptions
Class LLMService	A system service class that is similar to the Android’s android.app.Service class.
Class LLMCtx	A class that defines LLM context. LLMService interacts with apps via LLMCtx.
Method newLLMCtx(Optional systemPrompt) ->LLMCtxStub	A method that returns a stub of a new LLM context. On initializing, optional system prompts can be assigned to LLMCtx.
Method bindLLMService(app) ->LLMService	A method that binds the LLMService to an app.
Method callLLM(LLMCtxStub, newPrompt) ->outputs	A method that calls LLMService via an LLMCtx. It takes an LLMCtxStub and a new prompt as input, and returns the updated LLMCtxStub and decoding results.
Method deLLMCtx(LLMCtxStub)	A method that deletes an LLMCtxStub.

**Table 1: LLMS interfaces and descriptions.**

smart reply functionality during chatting on telegram. Consequently, a lightweight app could easily get killed by OS’s LMK if it possess an active LLM context, which though will not be used in a near future. (3) LLM contexts are naturally compressible, as will be discussed in §3.1, while app memory are mostly not. Treating the memory of LLM contexts and app as a whole misses such optimization opportunity. Instead, we propose decoupling the management of app and model context memory for better LLM efficiency.

### 3 LLMS DESIGN

#### 3.1 LLMS Overview.

This work advances the vision of LLaaS with LLMS, a first-of-its-kind LLM Service design on devices with a dedicated memory management mechanism of LLM contexts.

**LLMS APIs.** As shown in Table 1, we design LLMS’s interfaces to be compatible with Android Services. Apps interact with

LLMS via LLMCtx. Specifically, we show a chatbot app in Figure 3. Each round, an app appends new prompts to LLMCtx and invokes LLMS. When token generation completes, LLMCtx is updated. Such a chatbot can hold multiple contexts, which are persistent until the app explicitly deletes them through deLLMCtx(). LLMS globally configures the maximal context numbers per app and the maximal context length. LLMS allows each app to hold up to K (a configurable system parameter) active LLM contexts. Note that each active LLM context has a maximal memory usage with a maximal context window size supported by the LLM.

**LLMS’s design goal.** As a system service, LLMS optimizes for the Quality-of-Service (QoS) it exposes to apps. Specifically, LLMS’s design centers around the memory inefficiency challenge as presented in §2.3, aiming to minimize the *context switching latency*, akin to how the mobile memory managers minimize the app cold start latency [26, 55, 77]. For example, in Figure 3, LLMS ensures fast context preparation (i.e., make it in memory for LLM inference) when a context is invoked. This goal has not been explored in prior literature, and is fundamentally different with (as well as orthogonal to) LLM inference speed optimizations during continuous token generation [59, 73].

**LLMS’s context memory model.** LLMS exploits a unique design space: unlike the conventional app memory, LLM context memory is essentially *data chunks that can be approximated*. We show the main body of context memory, KV cache’s memory details in Figure 4(a/b). Its layout is a growing token sequence that can be divided into chunks with flexible granularity; each token is a tensor whose numbers can be approximated (e.g. through quantization [70] or sparsification [85]). Identifying the above characteristics, LLMS further introduces *data swapping* to extend the limited device memory, and builds its memory model with swappable and approximatable chunks. As shown in Figure 4(c), a context is divided to swappable fragment (KV cache) and memory-resident fragment (prompt/output texts) by LLMS. *The swappable fragment is managed in chunks*. Each chunk has the same in-chunk compression ratio and same numbers of tokens. In doing so, LLMS makes full use of the limited device RAM and exposes ample virtual memory to contexts. By carefully selecting the chunk size, LLMS can utilize memory with minimal fragmentation compared to managing the context as a whole; compared to token-level managing, a chunk can make better use of IO bandwidth. Such a chunk size is set empirically to a default number 16 tokens. We demonstrate its effectiveness and discuss its selection rationale with experiments in §5.4.

**Overview of LLMS’s context memory management.** Figure 5 shows an overview of LLMS. The core of LLMS manages chunks of context memory during their lifecycle. It performs the following primitives: ①Claim, which directly allocates

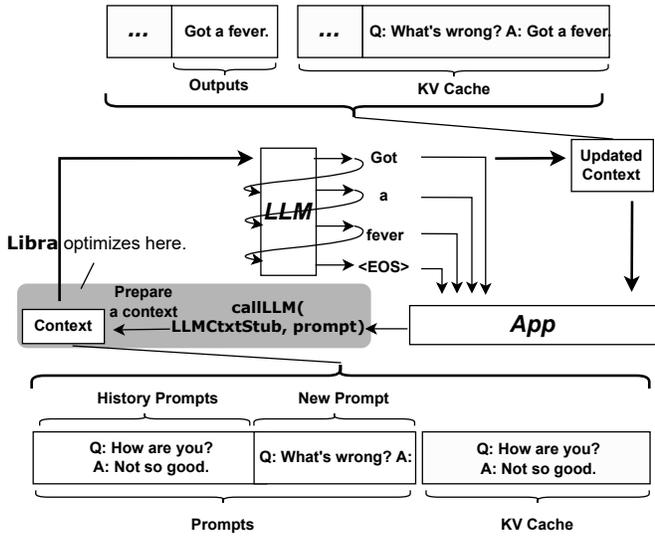


Figure 3: Left: a workflow of a chatbot app calling LLM Service via LLMS. Right: pseudo codes in Java.

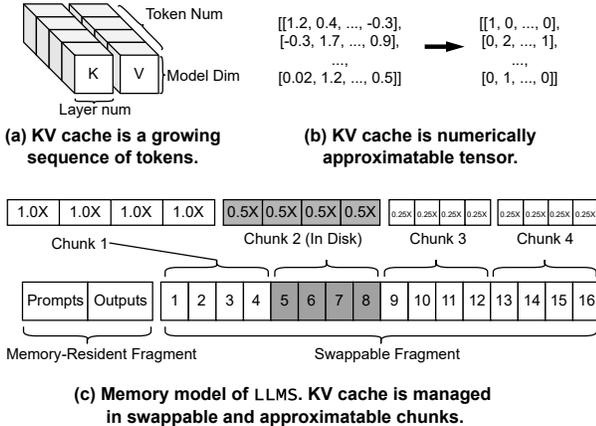


Figure 4: LLMS's chunk-wise memory model. Chunks are memory blocks that contain same number of compressed tokens and can be swapped to disk.

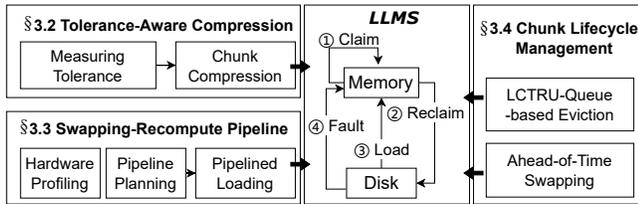


Figure 5: LLMS context memory managing overview.

free memory to a chunk; ②Reclaim, which swaps a chunk out to disk and reallocates its memory to a new chunk when memory is under pressure; ③Load, which moves a missing chunk from disk to memory before LLM inference; ④Fault, which moves a missing chunk from disk to memory at each

LLM inference iteration. In the rest of the section, we introduce the three key techniques of LLMS to meet its design goal: (1) Tolerance-Aware Compression (§5.2) adopts chunk-wise compression to minimize I/O overhead without noticeable accuracy loss; (2) Swapping-Recompute Pipeline (§3.3) further utilizes the idle computing to accelerate context switching; (3) Chunk Lifecycle Management (§3.4) judiciously decides which chunk and when to swap-out to enhance LLMS's QoS.

### 3.2 Tolerance-Aware Compression

**Chunks exhibit different accuracy tolerance to compression.** KV cache compression methods such as quantization or sparsity have been studied by many recent literature [70, 85]. However, such methods treat the context as a whole and do not make full use of the idea of chunking. LLMS's key idea is that, different chunks do not contribute coequally to LLM inference. For instance, a context chunk with tokens like "context management system" should contain more information than an "and so on" chunk, and the latter should show stronger tolerance to compression (i.e., can be further compressed). Recognizing that, LLMS's chunk compression is tolerance-aware: it first applies a conservative KV cache compression to all chunks first, and then iteratively compresses those with higher accuracy-loss tolerance aggressively.

**Measuring the tolerance.** Compression tolerance is measured by the information density. Information density  $D_i$  of the  $i$ th chunk is calculated by the attention scores:

$$D_i = \frac{1}{q-p} \sum_{col=p}^q \frac{1}{L} \sum_{l=0}^L \frac{1}{H} \sum_{h=0}^H \left( \frac{1}{R-row} \sum_{row=0}^R A_{row,col}^{l,h} \right), \quad (1)$$

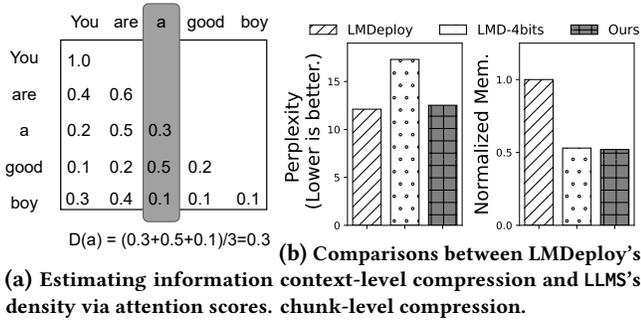


Figure 6: LLMS's tolerance-aware compression.

where  $A_{row,col}^{l,h}$  is the attention score of the  $l_{th}$  layer and  $h_{th}$  head, and the  $i_{th}$  chunk contains tokens from the  $p_{th}$  to  $q_{th}$ . Specifically, as shown in Figure 6a, the attention score is a  $R$  rows and  $C$  columns lower triangular matrix calculated by  $A = softmax(mask(\frac{Q \cdot K^T}{\sqrt{d_k}}))$  [63]. Each row represents the "attention score" that a token pays to others. The scores are softmaxed and sums to 1.0 at each row. For instance, the number "0.5" at "a" row and "are" col means that the token "a" pays 50% attention to "are". If a token is always paid more attention by other tokens, it should be more informative. Thereby, LLMS estimates a token's information density by averaging its column in attention score matrix. In Figure 6a, the information density of token "a" is  $(0.3+0.5+0.1)/3 = 0.3$ . As shown in Equation 1, such token-level information density is further accumulated by heads, layers and tokens to achieve chunk-level density, i.e., compression tolerance.

**Compressing chunks.** LLMS provides multiple levels of compression ratio for chunks, denoted as  $\{ratio_w\}$ . Before compression, LLMS computes each chunk's  $D_i$  and determines its ranking  $Rank_i(\%)$  among all other chunks in the context. Then a series of thresholds  $\{\sigma_{ratio}\}$  are formed. Chunk  $i$  is compressed to  $ratio_w$  based on  $Rank_i, s.t.,$

$$\sigma_{ratio_{w+1}} < Rank_i \leq \sigma_{ratio_w}. \quad (2)$$

Specifically, the thresholds are formed by maximizing the overall information intensity of a context under a given global average compression ratio  $ratio_{global}$ , which is configurable by the OS. LLMS maximizes

$$ctxInfo = \sum_w \frac{1}{ratio_w} \sum_{\sigma_{ratio_{w+1}} < Rank_i \leq \sigma_{ratio_w}} D_i, \quad (3)$$

$$s.t., \sum_w ratio_w \cdot (\sigma_{ratio_w} - \sigma_{ratio_{w+1}}) = ratio_{global}.$$

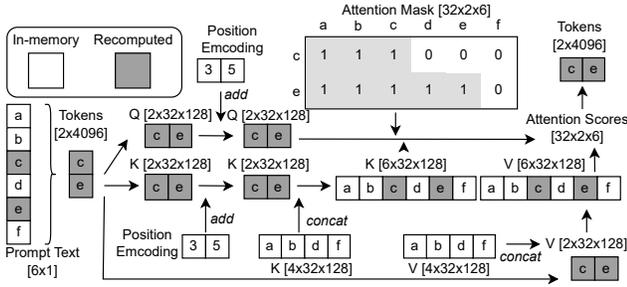
In practice, LLMS adopts quantization for compression. During LLM inference, the generated KV cache is quantized by state-of-the-art context-level quantization methods [14, 70] that have already been built in LLM. LLMS further performs channel-wise linear [39] quantization with lower

bitwidth. For instance, atop an 8-bits quantization method, LLMS can further provide 4-bits and 2-bits quantization for some chunks. Formula 3 can be optimized through a simple differentiation in this case, as there is only one variable. Note that LLMS's tolerance-aware compression is general. When the LLM's default KV quantization algorithm is 4-bits, it can still work by providing 2-bits and 1-bit further quantization. **Micro experiments** are conducted to show the effectiveness of LLMS's tolerance-aware compression. We adopt the KV cache quantization method of a state-of-the-art LLM inference framework LMDeploy [14], whose default quantization bitwidth is 8-bit (INT8). We set LLMS compression ratios to  $\{ratio_w\} = \{8/8, 4/8, 2/8\}$ . With the global compression ratio  $ratio_{global}$  set to 50%, we run language modeling with Llama2-7B model on WikiText-2 dataset [51]. As shown in Figure 6b, we compare our method to LMDeploy's 8-bits (LMDeploy) and 4-bits (LMD-4bits) quantization. Our method achieves comparable accuracy (perplexity) to 8-bits quantization and comparable memory consumption compared to 4-bits quantization. We show detailed discussion of our method's efficacy and rationale of ratio selection in §5.2.

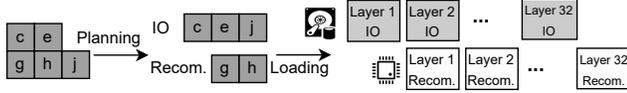
### 3.3 Swapping-Recompute Pipeline

The Load primitive in LLMS loads missing chunks from disk to memory upon calling `callLLM()`. LLMS introduces recompute to swapping-in to further accelerate it, as processor is idle during disk I/O. Recompute here means that use the original prompt text peices to calculate a portion of KV cache, as KV cache is essentially LLM activations. By recomputing some chunks in a pipelined maner instead of loading them through disk I/O, we can fully utilize the hardware.

**Making interleaved chunks recomputable.** Recall that in LLMS's context memory model, each chunk can be swapped out to disk. Thereby, LLMS faces a challenge: employing recompute to recover interleaved non-contiguous chunks. To do so, LLMS refines the LLM recompute procedure. Given a chunk-wise KV cache  $C = \{chunk_i\}$  and corresponding prompt text  $\mathcal{T} = \{text_i\}$ , LLMS recovers the missing chunks  $\{chunk_o\}$  through  $\{text_o\}$  and  $\{chunk_i\} - \{chunk_o\}$ . Specifically, Figure 7 gives an example of LLMS's chunk-recomputing procedure. In Figure 7, the KV cache of a text sequence "a b c d e f" is partly swapped out of memory ("c" and "e"). To recover, LLMS embeds "c" and "e" to tokens, and recomputes them to get Q, K and V tensors. Here, LLMS applies a global position encoding to Q and K, i.e., encoding "c" with position 3 and "e" with position 5. Then, the recomputed K/V of "c" and "e" is inserted to the K/V of "a", "b", "d" and "f"; the entire K/V is recovered in current layer. After that, Q is calculated with the recovered K to get the attention scores. The attention mask is a causal mask [63] that retains all tokens before the current token. In Figure 7, the attention mask masks out



**Figure 7: An example of LLMs’s chunk-recomputing procedure. We use a Llama2-7B layer here for representation. It has 32 heads and 4096 hidden size. LLMS recomputes the missing “c” and “e” in KV cache based on their prompt text and other tokens’ KV cache.**



**Figure 8: Swapping-recompute pipeline.**

“d e f” for “c” and “f” for “e”. Finally, the tokens that input to the next layer are calculated by the attention scores and the recovered V. In doing so, LLMS realizes the exact recompute of any chunks.

**Swapping-recompute pipeline.** LLMS concurrently recomputes a portion of chunks and swaps other chunks into memory from disk. As shown in Figure 8, I/O and recompute are overlapped in a pipeline, where the next layer’s I/O is performed during the current layer’s recompute.

Such a pipeline is elastic: LLMS can adjust the loading configuration (i.e., which chunks are recomputed and which chunks are swapped) to maximize the efficiency. LLMS plans the configuration of each loading based on offline profiled hardware information.

*i. Profiling.* Recompute delay  $T_{re}(x, f, e)$  is a function w.r.t. the number of chunks  $x$ , the hardware frequency  $f$  and the energy mode  $e$ . IO delay  $T_{IO}(m)$  is a function w.r.t. the size  $m$  of on-loading chunks. In practice, we approximate  $T_{re}$  and  $T_{IO}$  with linear functions. LLMS performs a cost-effective one-shot measurement with discrete test points at installation time. The above functions fit these test points.

*ii. Planning.* LLMS plans its elastic pipeline by solving the following optimization. Given the on-loading memory size  $m$  and the number of chunks  $\{x_{ratio_w}\}$  with different compression ratios, LLMS minimizes the following equation by determining a proper number of recomputed chunks  $\{x_{ratio_w}^{re}\}$

with different compression ratios.

$$\begin{aligned}
 pipelineDelay = \max [ & T_{re}(\sum_w x_{ratio_w}^{re}), \\
 & T_{IO}(m - \sum_w ratio_w \cdot x_{ratio_w}^{re})], \quad (4) \\
 s.t., \forall w, & x_{ratio_w}^{re} < x_{ratio_w}.
 \end{aligned}$$

Such a problem is solved by a linear programming, which incurs negligible overhead on devices.

### 3.4 Chunk Lifecycle Management

LLMS manages the lifecycle of KV cache chunks to be more friendly to context switching. It mainly decides which chunk and when to swap out. Regarding which chunk to swap out, it employs an LCTRU queue to determine the eviction priority. Regarding when to swap out, it adopts an ahead-of-time swapping-out approach to hide the time for reclaiming memory during context switching.

*i. AoT Swapping.* Compared to complex memory modification timing of apps [38, 46, 89], LLM Service’s memory modification is much easier to monitor: chunks are sequentially modified during LLM inference. Thus, swapping-out can be performed ahead of reclaim. LLMS swaps out all the modified chunks at the returning stage of callLLM() (even when not under memory pressure). Its delay is imperceptible to the LLM Service caller. In doing so, the reclaim primitive at context preparation stage is overhead-free.

*ii. LCTRU-Queue-based Eviction.* A good eviction policy can reduce the overhead of swapping. App memory managing methods, such as LMK, evict memory based on the app types (ranked by oom\_adj\_score). As a system service, LLMS does not differentiate the type of context owner. Its eviction policy is based on two principles: *i)* leveraging contexts’ time locality; *ii)* heavy chunks should be evicted first. Principle *ii)* is derived by the swapping-recompute pipeline in §3.3. Given memory size  $m$ , the total number of chunks is inversely proportional to the number of less-compressed chunks. According to Equation 4, under the same memory size, a smaller number of chunks will result in a lower pipeline delay, as the recomputing delay  $T_{re}(x, f, e)$  is irrelevant to memory size.

Based on the above principles, LLMS’s employs an LCTRU (Least Compression-Tolerable and Recently-Used) queue. The LCTRU queue consists of multiple concatenated sub-queues with different compression ratios, i.e.,  $Q_{LCTRU} = \{Q_{ratio_w}\}$ . Each sub-queue is ordered by the recently accessed time of its in-memory chunks.  $Q_{LCTRU}$  is updated upon each invocation of callLLM(). When memory reclaiming occurs, LLMS pops out the corresponding number of elements from  $Q_{LCTRU}$  based on the required memory size.

Device Name	RAM	Disk	Hardware Accelerator
Jetson Orin NX	8 GB	NVMe SSD	1024-core Ampere™ GPU
Jetson TX2	8 GB	SATA HDD	256-core Pascal™ GPU
MI14 Smartphone	8 GB	UFS 4.0	Hexagon™ 8Gen3 NPU

**Table 2: Details of mobile/edge devices we use.**

Besides, LLMS manages a context’s chunks as a memory-resident working set: during `callLLM()`, LLMS *locks* the context memory, forbidding reclaiming their own chunks. In doing so, LLMS avoids system thrashing and realizes being transparent to LLM inference. Notably, although it will not be triggered by LLMS, the Fault primitive is still retained for robustly handling exceptions such as system crush.

## 4 IMPLEMENTATION AND METHODOLOGY

**LLMS implementation.** We have fully implemented a LLMS prototype with 3.5k LoC in Python/C++. We implement an LLM Service on three representative COTS mobile/edge devices shown in Table 2. Jetson Orin NX/TX2 [7, 8] are high-end edge boards for autonomous robotics or cars. MI14 [15] is a smartphone equipped with UFS4.0 storage and Hexagon 8Gen3 NPU. We build LLM Service atop Huggingface Transformers [68] and mllm [53]. The former is the most popular off-the-shelf LLM deployment framework on devices with Pytorch support [19]. On smartphones, we choose the latter because it is lightweight and resource-efficient. The LLM Service runs as an independent process. It receives inference requests from client processes through socket IPC. Except for the client processes and the LLM Service process, we do not run any other non-essential application processes on device. We select two LLMs: Llama2-7B [62] and OPT-6.7B [83], with a maximal context length as 4K and 2K, respectively. We apply a sliding window [71] to the context to enable LLM inference in a streaming manner. The weights are downloaded from the official repositories on huggingface website. We quantize LLM weights to 4-bit integers with GPTQ [34]. The KV cache is stored in 8-bits integer by default [70]; LLMS further applies chunk-wise compression to it with its tolerance-aware compression technique (§5.2) with a global compression ratio  $ratio_{global}$  as 50%. We select three levels of chunk-wise compression ratio, i.e.,  $\{ratio_w\} = \{8/8, 4/8, 2/8\}$ .

The context memory management module of LLMS is embedded within LLM Service. We use pickle [17] and pickle-in-cpp [18] to implement memory-disk swapping. The chunk size is set to 16 tokens. Since the sub-byte data format is not natively supported by the LLM inference framework, LLMS utilizes parallel bit-shift operations to pack the compressed data into a supported INT8 format. The swapping-recompute

Task	Dataset	Delta length
News Classification	AGnews [84]	0.2k–0.5k
Document Summary	Xsum [54]	1k–2k
Chat History Summary	Samsun [37]	0.1k–0.3k
Text Comprehension	cnn dailymail [41]	0.5–1k
Translation	WMT17-de-en [28]	0.1k–0.5k
Sentiment Classification	SST-2 [28]	0.01k–0.1k

**Table 3: Datasets we use for trace synthesis. An entry in a dataset is regarded as one LLM calling. "Delta length" refers to the length of context growth after each `callLLM()`.**

pipeline is implemented by multithread. We use an independent I/O thread to load chunks from disk to memory. The computation thread proceeds to the next layer only after the I/O thread for the current layer (reading the next layer’s KV cache) has completed.

**Context switching trace.** To the best of our knowledge, there is no publicly available trace of LLMS context switching on devices. In order to comprehensively and accurately evaluate LLMS, we synthesized a trace that formulated by

$$Trace = \{(Time_i, CtxtID_i, Prompt_i, groundTruth_i)\}, \quad (5)$$

Where  $Time_i$  is the  $i_{th}$  calling time of `callLLM()` with  $CtxtID_i$ , and  $Prompt_i$  and  $groundTruth_i$  are the input and ideal output text. The prompts and groundTruths for a context are derived from a dataset in Table 3, while a dataset can derive multiple contexts. We generate calling time  $Time_i$  using Poisson distribution with different calling rates. Akin to apps’ switching [26, 55, 77], context switching pattern could be complex: it can be either irregular or with preference (e.g., influenced by invoke history or workloads). Recognizing that, we construct the following various patterns of context switching to simulate real-world scenarios.

- **Random.** Contexts are switched with the same probability.
- **Markov.** Context switching is determined by a first-order Markov process, which assigns higher priority to recently used contexts.
- **Gaussian.** Context switching follows a Gaussian distribution w.r.t delta length. In this pattern, a context with moderate workload is more likely to be requested by the apps.

Note that LLMS will not try to predict the context switching pattern, as such a pattern cannot be known a priori.

We synthesize 72-hours-long traces with different settings to evaluate LLMS. We will make the traces used in our experiments publicly available for reproducibility and further research on on-device LLMS.

**Baselines.** We compare LLMS to the following alternatives by reporting the performance on the same trace:

- **LMK.** Contexts are killed by a low-memory killer [2] when memory is under pressure. The killed contexts need to be recomputed when called again.

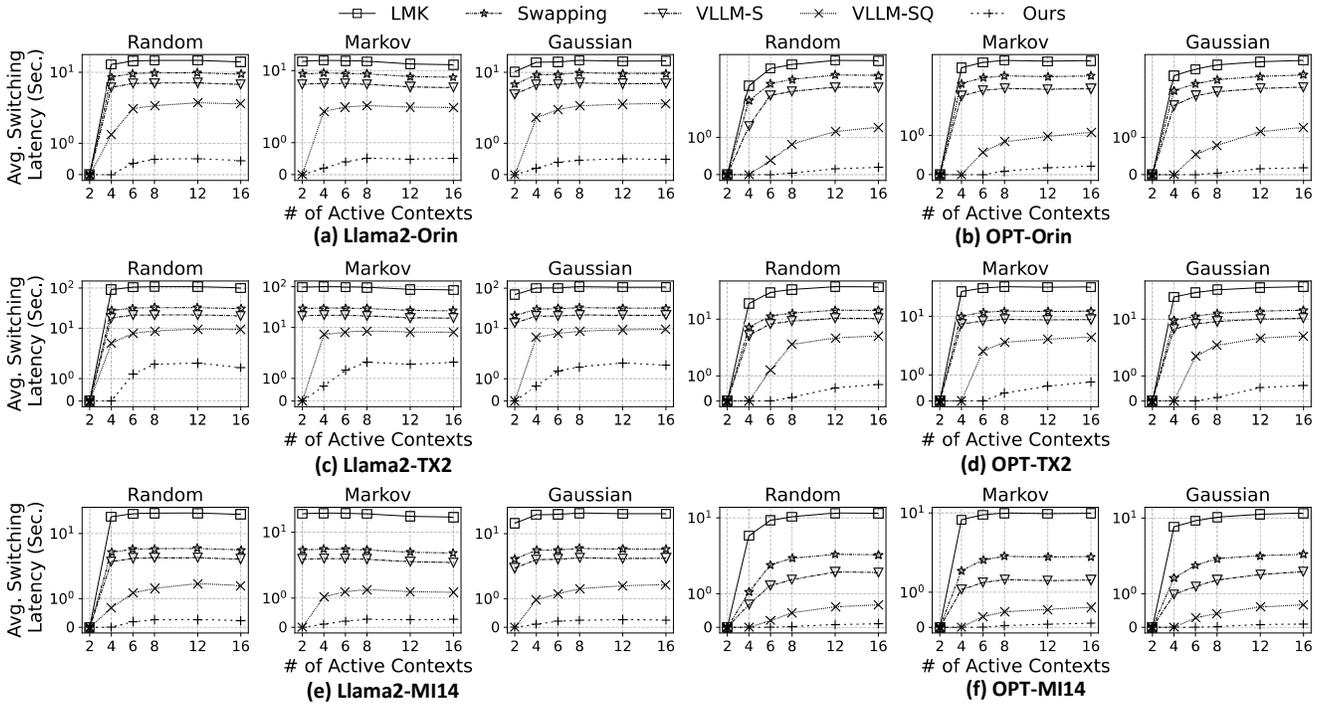


Figure 9: On-average context switching latency on a 72-hours-long trace.

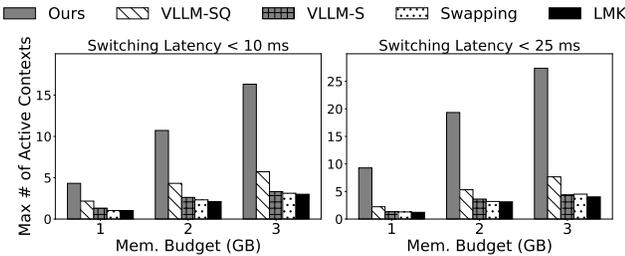


Figure 10: Performance under various memory budgets. Model: Llama2; device: Orin.

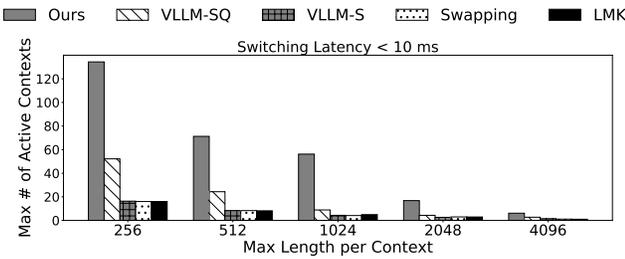


Figure 11: Performance under various maximal context lengths. Model: Llama2; device: Orin.

- Swapping. Contexts are swapped out to disk as a whole when memory is under pressure. The swapped-out contexts need to be swapped-in when called again.

- VLLM-S. VLLM [45] is a state-of-the-art KV cache managing system without compression. We reproduce its chunk-wise KV cache managing on devices. Chunks are swapped under memory pressure.

- VLLM-SQ. KV cache chunks in VLLM-S are further compressed by a state-of-the-art activation quantization algorithm SmoothQuant [70]. Chunks are equally quantized to the same level (INT8).

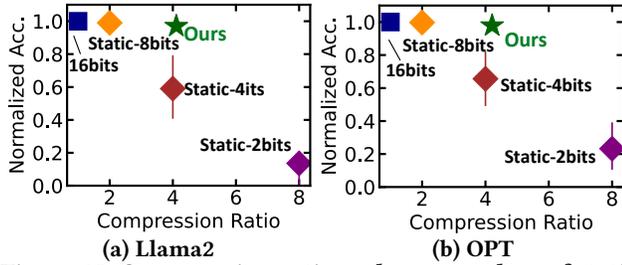
**Metrics.** We mainly report two metrics of LLM Service’s context switching QoS: *Context Switching Latency on Average* and *Maximal Number of Active Contexts*. The former is under a given number of active contexts; the latter is under a given switching latency constraint. “Active context” refers to contexts that are persistent (have not been deleted by `delLLMctx()`) and could be invoked.

## 5 EVALUATION

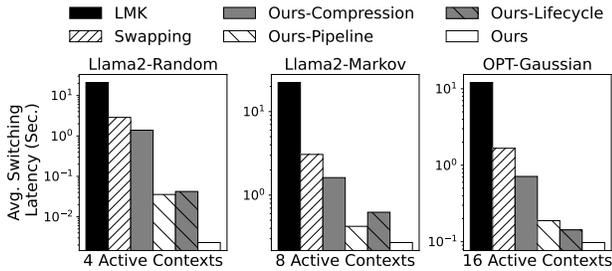
### 5.1 End-to-End Context Switching Performance

We first evaluate to what extent LLMs enhances the context switching QoS of LLM Service.

**End-to-end switching latency.** We report the on-average switching latency of 2/4/6/8/12/16 active contexts by running the synthesized trace in §4. The maximal retained context length is set to the LLM’s default context window (4k for Llama2 and 2k for OPT). The calling rate is one request in five



**Figure 12: Compression ratio and accuracy loss of static quantization v.s. our quantization. The compression ratio is taken reciprocally here, meaning the higher, the better.**



**Figure 13: Ablation study.**

minutes on average. All remaining memory after running the LLM Service on the device is dedicated to contexts. The experimental results are shown in Figure 9. The y-axis has been taken with symmetric logarithmic scale. We have the following observations.

In general, compared to the de facto app memory managing method LMK, LLMS achieves a significant reduction in switching latency by up to 2 orders of magnitude; compared to the vanilla Swapping baseline, LLMS reduces switching latency by 1–2 orders of magnitude; compared to other strong baselines that applies chunking+swapping (VLLM-S) and chunking+swapping+compression (VLLM-SQ) to KV cache managing, LLMS still achieves up to 20 $\times$  and on average 9.7 $\times$  reduction.

LLMS brings substantial context switching speed improvements across various context switching patterns, devices, and LLMs. We conducted experiments with identical settings for the three patterns (Random, Markov and Gaussian). The results indicate that LLMS can consistently handle different context access patterns. On three different devices (Orin/TX2/MI14), LLMS consistently demonstrates significant performance improvements. Notably, on the TX2, the overall switching latency is longer compared to the other two devices. This is due to its lower disk bandwidth (SATA HDD) and less-powerful hardware accelerators, which restrict LLMS’s swapping and swapping-recompute pipeline. Nevertheless, LLMS still outperforms baselines significantly, owing to its context compression and chunk lifecycle management. Additionally, LLMS achieves substantial performance

improvements across different LLMs. On the OPT model, the switching latency is lower, attributed to its smaller context window (traded for shorter memory and lower in-context learning ability).

**Various memory budgets.** We report the maximal number of active context under different switching latency constraints. As shown in Figure 10, under a 10 ms latency constraint, LLMS supports the switching between 4.32/10.72/16.32 contexts with 1GB/2GB/3GB memory budget, 1.99 $\times$ /2.48 $\times$ /2.85 $\times$  higher than baselines; under a 25 ms latency constraint, LLMS supports the switching between 9.28/19.34/27.38 contexts with 1GB/2GB/3GB memory budget, 4.16 $\times$ /3.62 $\times$ /3.57 $\times$  higher than baselines.

**Various maximal context lengths.** We evaluate maximal number of active context under various maximal context lengths. As shown in Figure 11, under a 10 ms latency constraint, LLMS supports 2.57 $\times$ /2.95 $\times$ /3.31 $\times$ /3.73 $\times$ /2.24 $\times$  more active contexts with 256–4096 maximal context lengths.

## 5.2 Compression Efficacy

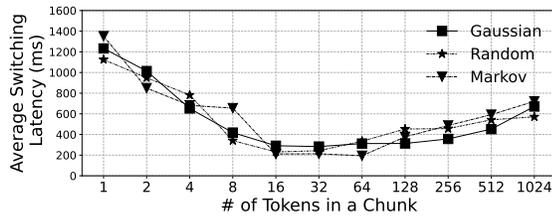
Recall that LLMS employs various levels of compression for different chunks. We evaluate its overall efficacy and discuss the rationale of parameter selection specified in §4. In Figure 12, we compare statically quantizing all chunks to the same bitwidth to our quantization. We choose [70] for static quantization. The accuracy and compression ratio are averaged across datasets mentioned in Table 3. The  $ratio_{global}$  and  $\{ratio_w\}$  are same to §4.

**Overall efficacy.** We observe that our approach outperforms static methods. Due to plenty of extreme outliers [70], aggressively compressing the entire KV cache into 4-bits/2-bits incurs significant accuracy loss (up to 59%/99%). In comparison, our method achieves a compression ratio 2 $\times$  higher than static methods with negligible loss in accuracy.

**Rationale of 4-bits/2-bits compression.** Generally, as shown in Figure 12, a three levels compression have provided substantial space for almost lossless approximation. From the perspective of implementation, 2-bits and 4-bits compression are more hardware-friendly on mobile SoCs.

## 5.3 Ablation Study

We further conduct a breakdown analysis of the benefit brought by LLMS’s each technique. The experiments are performed on Jetson Orin NX. The results are illustrated in Figure 13. We observe that all techniques have non-trivial contribution to the improvement. For instance, when using Llama2 model to serve 8 active contexts that called in a Markov pattern, LLMS takes 0.27 seconds to switch to a new context on average. Without our chunk lifecycle management, this number becomes 0.62 seconds; without our tolerance-aware



**Figure 14: Influence of chunk size. Device: Orin; model: Llama2; active contexts: 8.**

compression or swapping-recompute pipeline, this number becomes 0.42 seconds and 1.62 seconds, respectively.

## 5.4 Chunk Size Selection

LLMS manages KV cache in chunks. In our all experiments, the chunk size is set to 16 tokens. Here we evaluate the influence of chunk size on context switching to validate this setting. In Figure 14, we report the context switching latency under various token numbers in a chunk. We observe that a too large or too small chunk size incurs undesirable switching latency. The reasons are two fold: small chunks cannot fully utilize disk bandwidth; large chunks result in redundant swapping. Therefore, LLMS selects the optimal trade-off point to fully utilize the idea of chunking.

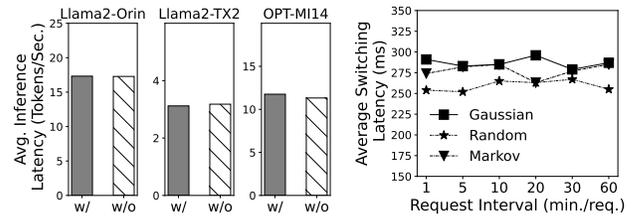
## 5.5 Service Stability Analysis

We analyze LLMS’s influence on the stability of LLM Service. **Influence on LLM inference.** By design, LLMS aims to minimize the context switching latency. In Figure 15a, we compare the performance of LLM inference between scenarios with and without the use of LLMS. As observed, there is no significant performance difference (within 5%).

**Sensitivity to service calling frequency.** Recall that we generate the trace’s LLMAaaS calling time with a Poisson distribution, which is influenced by the calling rate, i.e., service calling frequency. In Figure 15b, we report the switching latency under various request interval with 16 active contexts on Jetson Orin NX. LLMS demonstrates stable context switching latency at both high and low calling frequencies.

## 6 RELATED WORK

**Foundation models on devices.** AI has become a prevalent workload on mobile devices. Empowered by the recent progress, especially LLMs in ML community, some work proposes replacing fragmented task-specific models with a one-size-fits-all foundation model [48, 60, 62, 69, 75, 78, 81]. Such models have larger parameter sizes, stronger general-purpose task capabilities, and support multiple modalities. For instance, M4 [81] achieves comparable accuracy to specialized models on five modalities (image/text/audio/IMU/mix) with a multi-path executed architecture. With the support of corresponding software [10, 53, 59, 61, 73] and hardware [20],



**(a) Influence on LLM inference (b) Sensitivity to service calling performance.**

**Figure 15: LLMS’s influence on LLMAaaS stability. “w/” and “w/o” in (a) means managing context memory with and without LLMS, respectively.**

a plethora of revolutionary mobile applications [4, 12, 16, 30, 66, 72, 74] are built based on foundation models. One killer app among them is the LLM agent-based UI automation [47, 66]. For instance, Autodriod [66] employs Vicuna [31] to complete an arbitrary task by interacting with the smartphone GUI. LLMS sheds light on deploying the aforementioned foundation models on devices. At the OS level, LLMS provides opportunities to leverage NPU or batching; at the application level, by treating context as LLMAaaS interface, LLMS can provide personalized and stateful services for apps. **Swapping-based mobile OS memory management.** A considerable amount of work [38, 44, 46, 88, 89] employs swapping to mitigate the cold-start latency caused by low-memory killer. For instance, MARS [38] optimizes Linux swapping to enhance performance on flash storage devices. By deactivating garbage collection, it reclaims memory from background apps. Exploiting the opportunity of KV cache, LLMS’s swapping differs from these approaches. For instance, its swapping operates at the granularity of token chunks, rather than pages or objects. Also, in contrast to lossless app memory compression (e.g., zram [21]), LLMS introduces approximations for chunks.

**KV cache approximation.** KV cache facilitates LLM in memorizing historical knowledge. Some recent work [22, 70, 82, 85, 87] focuses on optimizing KV cache by sparsification or quantization. Dynamic sparsification methods, such as Big Bird [82], only mitigates the compute overhead; static sparsification methods, such as  $H_2O$  [85], reduces memory footprint by permanently removing tokens from subsequent LLM decoding. Regarding quantization, a considerable amount of work [14, 39, 70] can losslessly quantize KV cache to 8-bit integers. Some work [86] even propose quantizing it to 4 bits, partly sacrificing generation quality for lower memory consumption. LLMS’s compression is orthogonal to these techniques. Atop them, it further performs more aggressive quantization on less informative chunks to achieve a better accuracy-memory trade-off.

## 7 CONCLUSION

This work advocates LLM as a service on mobile devices (LLMaaS), a new paradigm to fully unleash the power of on-device LLM. We then present an end-to-end LLMaaS design named LLMS with an efficient memory management system of LLM contexts. LLMS enables low-overhead LLM context switching under tight memory constraint through fine-grained, chunk-wise, globally-optimized KV cache compression and swapping. Extensive experiments demonstrate the efficacy of LLMS.

## REFERENCES

- [1] 2024. AICore. <https://developer.android.com/ml/aicore>.
- [2] 2024. Android low-memory killer. [https://developer.android.com/topic/performance/memory-management#low-memory\\_killer](https://developer.android.com/topic/performance/memory-management#low-memory_killer).
- [3] 2024. Gboard Smart Reply. <https://developers.google.com/ml-kit/language/smart-reply>.
- [4] 2024. Glarity. <https://glarity.app/>.
- [5] 2024. GPT-based email writer. <https://hix.ai/ai-email-writer-email-generator>.
- [6] 2024. GPT4-Turbo. <https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo>.
- [7] 2024. Jetson Orin NX. <https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/>.
- [8] 2024. Jetson TX2. <https://developer.nvidia.com/embedded/jetson-tx2>.
- [9] 2024. Large Language Models On-Device with MediaPipe and TensorFlow Lite. <https://developers.googleblog.com/2024/03/running-large-language-models-on-device-with-mediapipe-andtensorflow-lite.html>.
- [10] 2024. Llama.cpp. <https://github.com/gganganov/llama.cpp>.
- [11] 2024. LLM-based AI-Assistant. <https://github.com/avsirma/LLM-based-AI-Assistant>.
- [12] 2024. LLM customer service and support. <https://www.databricks.com/solutions/accelerators/llms-customer-service-and-support>.
- [13] 2024. LLM telegram chatbot. <https://github.com/Fatal3xcept10n/LLM-Telegram-Chatbot>.
- [14] 2024. LM Deploy. <https://github.com/InternLM/lmdeploy/tree/main>.
- [15] 2024. MI14 smartphone. [https://en.wikipedia.org/wiki/Xiaomi\\_14](https://en.wikipedia.org/wiki/Xiaomi_14).
- [16] 2024. News Summarization with LLM. <https://github.com/KillerStrike17/News-Summarization-with-LLM>.
- [17] 2024. Pickle. <https://docs.python.org/3/library/pickle.html>.
- [18] 2024. Pickle-in-Cpp. <https://github.com/Usama-Azad/Pickle-in-Cpp>.
- [19] 2024. Pytorch. <https://pytorch.org/>.
- [20] 2024. Snapdragon 8 gen 3 mobile platform product brief. [https://docs.qualcomm.com/bundle/publicresource/87-71408-1\\_REV\\_C\\_Snapdragon\\_8\\_gen\\_3\\_Mobile\\_Platform\\_Product\\_Brief.pdf](https://docs.qualcomm.com/bundle/publicresource/87-71408-1_REV_C_Snapdragon_8_gen_3_Mobile_Platform_Product_Brief.pdf).
- [21] 2024. zRAM. <https://en.wikipedia.org/wiki/Zram>.
- [22] Reyna Abhyankar, Zijian He, Vikranth Srivatsa, Hao Zhang, and Yiying Zhang. 2024. APIServe: Efficient API Support for Large-Language Model Inference. arXiv:2402.01869 [cs.LG]
- [23] OpenAI: Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, et al. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]
- [24] Keivan Alizadeh, Iman Mirzadeh, Dmitry Belenko, Karen Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. 2024. LLM in a flash: Efficient Large Language Model Inference with Limited Memory. arXiv:2312.11514 [cs.CL]
- [25] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, M erouane Debbah,  tienne Goffinet, Daniel Hesslow, Julien Launay, Quentin Malartic, Daniele Mazzotta, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. 2023. The Falcon Series of Open Language Models. arXiv:2311.16867 [cs.CL]
- [26] Ricardo Baeza-Yates, Di Jiang, Fabrizio Silvestri, and Beverly Harrison. 2015. Predicting The Next App That You Are Going To Use. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining (Shanghai, China) (WSDM '15)*. Association for Computing Machinery, New York, NY, USA, 285–294. <https://doi.org/10.1145/2684822.2685302>
- [27] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2016. Neural Machine Translation by Jointly Learning to Align and Translate. arXiv:1409.0473 [cs.CL]
- [28] Ondrej Bojar, Rajen Chatterjee, Christian Federmann, Yvette Graham, Barry Haddow, Shujian Huang, Matthias Huck, Philipp Koehn, Qun Liu, Varvara Logacheva, Christof Monz, Matteo Negri, Matt Post, Raphael Rubino, Lucia Specia, and Marco Turchi. 2017. Findings of the 2017 Conference on Machine Translation (WMT17). In *Proceedings of the Second Conference on Machine Translation, Volume 2: Shared Task Papers*. Association for Computational Linguistics, Copenhagen, Denmark, 169–214. <http://www.aclweb.org/anthology/W17-4717>
- [29] Tom B. Brown, Benjamin Mann, Nick Ryder, et al. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL]
- [30] Qiwei Chen, Huan Zhao, Wei Li, Pipei Huang, and Wenwu Ou. 2019. Behavior Sequence Transformer for E-commerce Recommendation in Alibaba. arXiv:1905.06874 [cs.IR]
- [31] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. 2023. Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%\* ChatGPT Quality. <https://lmsys.org/blog/2023-03-30-vicuna/>
- [32] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL]
- [33] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, Lei Li, and Zhifang Sui. 2023. A Survey on In-context Learning. arXiv:2301.00234 [cs.CL]
- [34] Elias Frantar, Saleh Ashkboos, Torsten Hoefer, and Dan Alistarh. 2023. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. arXiv:2210.17323 [cs.LG]
- [35] Thomas Mesnard Gemma Team, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Laurent Sifre, Morgane Riviere, Mihir Sanjay Kale, Juliette Love, Pouya Tafti, L eonard Hussenot, and et al. 2024. Gemma. (2024). <https://doi.org/10.34740/KAGGLE/M/3301>
- [36] In Gim, Guojun Chen, Seung seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2023. Prompt Cache: Modular Attention Reuse for Low-Latency Inference. arXiv:2311.04934 [cs.CL]
- [37] Bogdan Gliwa, Iwona Mochol, Maciej Biesek, and Aleksander Wawer. 2019. SAMSum Corpus: A Human-annotated Dialogue Dataset for Abstractive Summarization. In *Proceedings of the 2nd Workshop on New Frontiers in Summarization*. Association for Computational Linguistics, Hong Kong, China, 70–79. <https://doi.org/10.18653/v1/D19-5409>
- [38] Weichao Guo, Kang Chen, Huan Feng, Yongwei Wu, Rui Zhang, and Weimin Zheng. 2016. MARS : Mobile Application Relaunching Speed-Up through Flash-Aware Page Swapping. *IEEE Trans. Comput.* 65, 3 (2016), 916–928. <https://doi.org/10.1109/TC.2015.2428692>
- [39] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. arXiv:1510.00149 [cs.CV]
- [40] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring Massive Multitask Language Understanding. arXiv:2009.03300 [cs.CY]

- [41] Karl Moritz Hermann, Tomas Kocisky, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. 2015. Teaching Machines to Read and Comprehend. In *NIPS*. 1693–1701. <http://papers.nips.cc/paper/5945-teaching-machines-to-read-and-comprehend>
- [42] Wenyi Hong, Weihang Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxuan Zhang, Juanzi Li, Bin Xu, Yuxiao Dong, Ming Ding, and Jie Tang. 2023. CogAgent: A Visual Language Model for GUI Agents. [arXiv:2312.08914](https://arxiv.org/abs/2312.08914) [cs.CV]
- [43] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. LoRA: Low-Rank Adaptation of Large Language Models. [arXiv:2106.09685](https://arxiv.org/abs/2106.09685) [cs.CL]
- [44] Sang-Hoon Kim, Jinkyu Jeong, and Jin-Soo Kim. 2017. Application-Aware Swapping for Mobile Systems. *ACM Trans. Embed. Comput. Syst.* 16, 5s, Article 182 (sep 2017), 19 pages. <https://doi.org/10.1145/3126509>
- [45] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.
- [46] Niel Lebeck, Arvind Krishnamurthy, Henry M. Levy, and Irene Zhang. 2020. End the Senseless Killing: Improving Memory Management for Mobile Operating Systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 873–887. <https://www.usenix.org/conference/atc20/presentation/lebeck>
- [47] 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, Guanqing Xiong, Fan Zhang, Xiang Li, Mengwei Xu, Zhijun Li, Peng Li, Yang Liu, Ya-Qin Zhang, and Yunxin Liu. 2024. Personal LLM Agents: Insights and Survey about the Capability, Efficiency and Security. [arXiv preprint arXiv:2401.05459](https://arxiv.org/abs/2401.05459) (2024).
- [48] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. 2023. Visual Instruction Tuning. [arXiv:2304.08485](https://arxiv.org/abs/2304.08485) [cs.CV]
- [49] Shuming Ma, Hongyu Wang, Lingxiao Ma, Lei Wang, Wenhui Wang, Shaohan Huang, Li Dong, Ruiping Wang, Jilong Xue, and Furu Wei. 2024. The Era of 1-bit LLMs: All Large Language Models are in 1.58 Bits. [arXiv:2402.17764](https://arxiv.org/abs/2402.17764) [cs.CL]
- [50] Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. 2022. PEFT: State-of-the-art Parameter-Efficient Fine-Tuning methods. <https://github.com/huggingface/peft>.
- [51] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer Sentinel Mixture Models. [arXiv:1609.07843](https://arxiv.org/abs/1609.07843) [cs.CL]
- [52] Sewon Min, Xinxin Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. 2022. Rethinking the Role of Demonstrations: What Makes In-Context Learning Work? [arXiv:2202.12837](https://arxiv.org/abs/2202.12837) [cs.CL]
- [53] mllm team. 2023. *mllm*. <https://github.com/UbiquitousLearning/mllm>
- [54] Shashi Narayan, Shay B. Cohen, and Mirella Lapata. 2018. Don't Give Me the Details, Just the Summary! Topic-Aware Convolutional Neural Networks for Extreme Summarization. [ArXiv abs/1808.08745](https://arxiv.org/abs/1808.08745) (2018).
- [55] Abhinav Parate, Matthias Böhmer, David Chu, Deepak Ganesan, and Benjamin M. Marlin. 2013. Practical prediction and prefetch for faster access to applications on mobile phones. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing (Zurich, Switzerland) (UbiComp '13)*. Association for Computing Machinery, New York, NY, USA, 275–284. <https://doi.org/10.1145/2493432.2493490>
- [56] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2022. Efficiently Scaling Transformer Inference. [arXiv:2211.05102](https://arxiv.org/abs/2211.05102) [cs.LG]
- [57] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100,000+ Questions for Machine Comprehension of Text. [arXiv:1606.05250](https://arxiv.org/abs/1606.05250) [cs.CL]
- [58] Christopher Rawles, Alice Li, Daniel Rodriguez, Oriana Riva, and Timothy Lillicrap. 2023. Android in the Wild: A Large-Scale Dataset for Android Device Control. [arXiv:2307.10088](https://arxiv.org/abs/2307.10088) [cs.LG]
- [59] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. 2023. PowerInfer: Fast Large Language Model Serving with a Consumer-grade GPU. [arXiv:2312.12456](https://arxiv.org/abs/2312.12456) [cs.LG]
- [60] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, et al. 2023. Gemini: A Family of Highly Capable Multimodal Models. [arXiv:2312.11805](https://arxiv.org/abs/2312.11805) [cs.CL]
- [61] MLC team. 2023. *MLC-LLM*. <https://github.com/mlc-ai/mlc-llm>
- [62] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. [arXiv:2307.09288](https://arxiv.org/abs/2307.09288) [cs.CL]
- [63] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention Is All You Need. [arXiv:1706.03762](https://arxiv.org/abs/1706.03762) [cs.CL]
- [64] Bryan Wang, Gang Li, and Yang Li. 2023. Enabling Conversational Interaction with Mobile UI using Large Language Models. [arXiv:2209.08655](https://arxiv.org/abs/2209.08655) [cs.HC]
- [65] Bryan Wang, Gang Li, and Yang Li. 2023. Enabling Conversational Interaction with Mobile UI using Large Language Models (CHI '23). Association for Computing Machinery, New York, NY, USA. <https://doi.org/10.1145/3544548.3580895>
- [66] Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. 2023. Empowering llm to use smartphone for intelligent task automation. [arXiv preprint arXiv:2308.15272](https://arxiv.org/abs/2308.15272) (2023).
- [67] Hao Wen, Hongming Wang, Jiaxuan Liu, and Yuanchun Li. 2024. DroidBot-GPT: GPT-powered UI Automation for Android. [arXiv:2304.07061](https://arxiv.org/abs/2304.07061) [cs.SE]
- [68] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi 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. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45. <https://www.aclweb.org/anthology/2020.emnlp-demos.6>
- [69] Shengqiong Wu, Hao Fei, Leigang Qu, Wei Ji, and Tat-Seng Chua. 2023. NExT-GPT: Any-to-Any Multimodal LLM. [arXiv:2309.05519](https://arxiv.org/abs/2309.05519) [cs.AI]
- [70] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models. [arXiv:2211.10438](https://arxiv.org/abs/2211.10438) [cs.CL]
- [71] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2023. Efficient Streaming Language Models with Attention Sinks. [arXiv \(2023\)](https://arxiv.org/abs/2303.08052).
- [72] Le Xiao and Xiaolin Chen. 2023. Enhancing LLM with Evolutionary Fine Tuning for News Summary Generation. [arXiv:2307.02839](https://arxiv.org/abs/2307.02839) [cs.CL]
- [73] Daliang Xu, Wangsong Yin, Xin Jin, Ying Zhang, Shiyun Wei, Mengwei Xu, and Xuanzhe Liu. 2023. LLMcad: Fast and Scalable On-device Large Language Model Inference. [arXiv:2309.04255](https://arxiv.org/abs/2309.04255) [cs.NI]
- [74] Huatao Xu, Liying Han, Qirui Yang, Mo Li, and Mani Srivastava. 2024. Penetrative AI: Making LLMs Comprehend the Physical World. [arXiv:2310.09605](https://arxiv.org/abs/2310.09605) [cs.AI]
- [75] Mengwei Xu, Wangsong Yin, Dongqi Cai, Rongjie Yi, et al. 2024. A Survey of Resource-efficient LLM and Multimodal Foundation Models.

- arXiv:2401.08092 [cs.LG]
- [76] An Yan, Zhengyuan Yang, Wanrong Zhu, Kevin Lin, Linjie Li, Jianfeng Wang, Jianwei Yang, Yiwu Zhong, Julian McAuley, Jianfeng Gao, Zicheng Liu, and Lijuan Wang. 2023. GPT-4V in Wonderland: Large Multimodal Models for Zero-Shot Smartphone GUI Navigation. arXiv:2311.07562 [cs.CV]
- [77] Tingxin Yan, David Chu, Deepak Ganesan, Aman Kansal, and Jie Liu. 2012. Fast app launching for mobile devices using predictive user context. In Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (Low Wood Bay, Lake District, UK) (MobiSys '12). Association for Computing Machinery, New York, NY, USA, 113–126. <https://doi.org/10.1145/2307636.2307648>
- [78] Bufang Yang, Lixing He, Neiwen Ling, Zhenyu Yan, Guoliang Xing, Xian Shuai, Xiaozhe Ren, and Xin Jiang. 2023. EdgeFM: Leveraging Foundation Model for Open-set Learning on the Edge. arXiv:2311.10986 [cs.LG]
- [79] Rongjie Yi, Liwei Guo, Shiyun Wei, Ao Zhou, Shangguang Wang, and Mengwei Xu. 2023. EdgeMoE: Fast On-Device Inference of MoE-based Large Language Models. arXiv:2308.14352 [cs.LG]
- [80] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). USENIX Association, Carlsbad, CA, 521–538. <https://www.usenix.org/conference/osdi22/presentation/yu>
- [81] Jinliang Yuan, Chen Yang, Dongqi Cai, Shihe Wang, Xin Yuan, Zeling Zhang, Xiang Li, Dingge Zhang, Hanzi Mei, Xianqing Jia, Shangguang Wang, and Mengwei Xu. 2023. Rethinking Mobile AI Ecosystem in the LLM Era. arXiv:2308.14363 [cs.AI]
- [82] Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. 2021. Big Bird: Transformers for Longer Sequences. arXiv:2007.14062 [cs.LG]
- [83] 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. 2022. OPT: Open Pre-trained Transformer Language Models. arXiv:2205.01068 [cs.CL]
- [84] Xiang Zhang, Junbo Jake Zhao, and Yann LeCun. 2015. Character-level Convolutional Networks for Text Classification. In NIPS.
- [85] 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. 2023. H<sub>2</sub>O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models. arXiv:2306.14048 [cs.LG]
- [86] Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, Size Zheng, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, and Baris Kasikci. 2023. Atom: Low-bit Quantization for Efficient and Accurate LLM Serving. arXiv:2310.19102 [cs.LG]
- [87] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2023. Efficiently Programming Large Language Models using SGLang. arXiv:2312.07104 [cs.AI]
- [88] Kan Zhong, Xiao Zhu, Tianzheng Wang, Dan Zhang, Xianlu Luo, Duo Liu, Weichen Liu, and Edwin H.-M. Sha. 2014. DR-Swap: Energy-efficient paging for smartphones. In 2014 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED). 81–86. <https://doi.org/10.1145/2627369.2627647>
- [89] Xiao Zhu, Duo Liu, Kan Zhong, Jinting Ren, and Tao Li. 2017. SmartSwap: High-performance and user experience friendly swapping in mobile systems. In 2017 54th ACM/EDAC/IEEE Design Automation