AXLEARN: MODULAR, HARDWARE-AGNOSTIC LARGE MODEL TRAINING

Mark Lee[*1] **Chang Lan**[*1] **Tom Gunter**[†1] **John Peebles**[†1] **Hanzhi Zhou**[†1] **Kelvin Zou**[†1]
Sneha Bangalore[1] **Chung-Cheng Chiu**[1] **Nan Du**[1] **Xianzhi Du**[1] **Philipp Dufter**[1] **Liang He**[1] **Ruixuan Hou**[1]
Haoshuo Huang[1] **Dongseong Hwang**[1] **Xiang Kong**[1] **Jinhao Lei**[1] **Tao Lei**[1] **Meng Li**[1] **Li Li**[1] **Jiarui Lu**[1]
Zhiyun Lu[1] **Yiping Ma**[1] **David Qiu**[1] **Vivek Rathod**[1] **Senyu Tong**[1] **Zhucheng Tu**[1] **Chong Wang**[1]
Jianyu Wang[1] **Yongqiang Wang**[1] **Zirui Wang**[1] **Floris Weers**[1] **Sam Wiseman**[1] **Guoli Yin**[1] **Bowen Zhang**[1]
Xiyou Zhou[1] **Danyang Zhuo**[§2] **Cheng Leong**[1] **Ruoming Pang**[‡1]

ABSTRACT

AXLearn is a production system which facilitates scalable and high-performance training of large deep learning models. Compared to other state-of-the-art deep learning systems, AXLearn has a unique focus on modularity and support for hardware-agnostic training. AXLearn's internal interfaces between software components follow strict encapsulation, allowing different components to be assembled to facilitate rapid model development and experimentation on different hardware infrastructure. AXLearn maintains constant complexity as we scale the components in the system, compared to linear or quadratic complexity in state-of-the-art training systems. This allows integrating features such as Rotary Position Embeddings (RoPE) into AXLearn across hundreds of modules with just 10 lines of code, compared to hundreds, as required in other systems. At the same time, AXLearn maintains equivalent performance compared to state-of-the-art training systems. We also share our experience in the development and operation of AXLearn at Apple.

1 INTRODUCTION

Large-scale deep learning models are now integral to society—they power chatbots such as ChatGPT and Gemini (OpenAI, 2022; Google, 2025), enhance video conferencing (Zoom, 2025), and support modern coding tools (Cursor, 2025). Modern deep learning systems prioritize performance and scalability to accommodate large models. Numerous techniques have been explored to this end, including parallelization strategies (Narayanan et al., 2021; Zheng et al., 2022), memory optimizations (Rajbhandari et al., 2020; Zhao et al., 2023), and model-specific kernel optimizations (Dao et al., 2022; Dao, 2023).

At Apple, we have integrated many AI models into our products, catering to billions of users worldwide. We have two requirements for our deep learning systems besides training performance and scalability. First, we aim to empower our model engineers to experiment with diverse model architectures and training techniques. They should write only a minimal amount of code to configure complex model definitions and training methods. We call this *modularity* of the deep learning system. Second, as a large technology company, we cannot rely on a single hardware vendor—any hardware can run into supply issues and vary in pricing. Our design goal is to be *hardware-agnostic*, supporting GPU, TPU, and AWS Trainium. This allows us to train on major cloud providers (e.g., AWS, Google Cloud, Azure) as well as our on-premises servers.

To facilitate modularity, the core design decision of AXLearn is to enforce *strict encapsulation*. While encapsulation is a well known principle in object oriented programming, we find that it is often neglected in ML frameworks as almost all existing frameworks rely on subtyping. The prevalence of subtyping can be partially attributed to the lack of formal analysis on how to *quantify* the modularity and extensibility of a system, where conventional design principles or "rules of thumb" are non-exhaustive and hard to measure. To this end, we propose a framework for quantifying the complexity of a system by measuring *asymptotic* Lines-of-Code (LoC) changes incurred by the addition of a new feature. Such a framework is necessary as the traditional mechanistic way of counting LoC can only capture complexity under a frozen snapshot, when extensibility is really about how a system evolves over time. We also show that the framework is consistent with directly counting LoC via the case study of integrating Rotary Positional Embeddings (RoPE) (Su et al., 2024) and Mixture of Experts

[1]Apple [2]Duke University. [*] Mark Lee and Chang Lan are the first authors. [†] denotes the core authors. [§]Danyang Zhuo contributed to this work as a visiting scholar at Apple. [‡] Ruoming Pang is the corresponding author.

Proceedings of the 9^{th} MLSys Conference (Industry Track), Bellevue, WA, USA, 2026.

Table 1. Comparing AXLearn with state-of-the-art deep learning systems. We define "partial" as systems that decouple parallelism from model implementation via device abstractions (e.g., XLA), but in practice do not apply strict encapsulation.

System	Underlying Framework	Model Agnostic	3D Parallelism	Modular	GPU	TPU	Trainium
Megatron-LM	PyTorch		✓		✓		
DeepSpeed	PyTorch	✓	✓		✓		
PyTorch FSDP	PyTorch	✓			✓		
PyTorch XLA FSDP	PyTorch	✓			✓	✓	
TorchTitan	PyTorch		✓	partial	✓		
Haiku	JAX	✓	✓	partial	✓	✓	
Flax	JAX	✓	✓	partial	✓	✓	
Pax	JAX	✓	✓	partial	✓	✓	
MaxText	JAX		✓	partial	✓	✓	
AXLearn (Ours)	JAX	✓	✓	✓	✓	✓	✓

(MoE) (Shazeer et al., 2017) into AXLearn as compared to other state-of-the-art deep learning systems. To our knowledge, AXLearn is the only training framework that adheres strictly to encapsulation—any module is replaceable, including the input pipeline, checkpointer, trainer loop—allowing complex features to be implemented without increasing the complexity of the overall system.

To enable hardware-agnostic training, we build AXLearn on top of XLA (xla, 2025) and GSPMD (Xu et al., 2021). Our native integration with XLA allows parallelism strategies to be automatically generated, but still allows hand-crafting kernel code for specific accelerators. For instance, on each hardware backend, we replace the attention layer with a custom kernel like FlashAttention (Dao et al., 2022; Dao, 2023). We believe this design strikes an ideal balance: by leveraging the XLA ecosystem, we can seamlessly support multiple hardware accelerators without sacrificing high performance. Due to its modular design, AXLearn also enables succinct user configurations to customize the parallelization, rematerialization, and quantization strategies, further simplifying the scaling experience.

Table 1 provides a list of deep learning systems designed for large models. Besides modularity and hardware requirement, the table indicates whether each system is model-agnostic or supports broader parallelism strategies—such as 3D parallelism (Narayanan et al., 2021)—to enable efficient training across diverse model architectures. We summarize several important observations. Some systems only support specific architectures: for example, MaxText (MaxText team, 2025) provides LLM implementations but does not extend easily to custom architectures as its design encourages fork-and-modify rather than reusing building blocks. Similarly, TorchTitan (Liang et al., 2024) requires model-specific parallel plans that traverse the model architecture, limiting its applicability to architectures not explicitly supported. Other systems are designed for specific backends: for instance, Megatron-LM (Narayanan et al., 2021) has carefully designed optimizations for Transformers on GPU, but these optimizations do not directly apply to other hardware. Apple uses a diverse set of model architectures across a range of cloud backends, so we cannot directly use these systems.

Over the past few years, we have deployed AXLearn to train thousands of models involving hundreds of engineers. The rapid adoption of AXLearn largely owes to its modularity and unique ability to scale on various public clouds, including Google Cloud TPU, AWS GPU or Trainium2. We share how our experience evolved our design choices over several years. AXLearn is open-sourced under Apache 2.0 license at `https://github.com/apple/axlearn`.

2 Motivation

2.1 Modularity

In existing ML systems, neural network layers are implemented by subtyping: a layer inherits from some base layer, instantiates child layers as instance attributes, and implements a `forward` method that handles the layer's logic. Consider the changes required to replace a feed-forward network (FFN) of a Transformer architecture with an MoE layer using subtyping. Taking the example from DeepSpeed (DeepSpeed, 2025b), one applies such a change by replacing the instance attribute for the original FFN:

```
- self.fc3 = nn.Linear(84, 10)
+ self.fc3 = nn.Linear(84, 84)
+ self.fc3 = deepspeed.moe.layer.MoE(...)
+ self.fc4 = nn.Linear(84, 10)
```

At a glance, this seems like a simple 4 LoC change. However, in practice one would need to subtype self.fc3's parent layer to apply such a change. This effectively reduces the problem of replacing the FFN with MoE, to the problem of replacing the parent layer with the new subtyped layer. By induction, it is easy to see how such a change can compound to changes to multiple modules across the subtype hierarchy. Indeed, this happens in practice: between DeepSpeed's

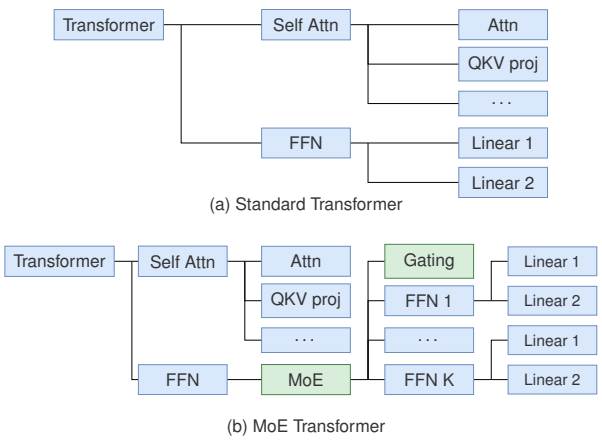


Figure 1. Specifying MoE Transformer in AXLearn. Blue components are reused from the specification of standard Transformer. In AXLearn, a user script that defines MoE only needs to specify the green parts of the neural network.

QwenV2 and QwenV2 MoE implementations (DeepSpeed, 2025a), over 200 LoC are required to apply the MoE layer, not accounting for the MoE layer itself—a far cry from the 4 LoC that we may have hoped.

Conceptually, MoE is a small change—ideally, one should be able to "swap" the FFN directly with an MoE equivalent, without incurring the side-effects of propagating changes up to ancestors like the Transformer layer. In AXLearn, this can be achieved by exploiting the compositional nature of neural networks: by implementing the FFN and MoE layers with compatible input/output interfaces, and *encapsulating* all MoE specific details within its layer implementation, we realize this ideal scenario of treating MoE as a "drop-in" replacement.

Note that by naively comparing the LoC between implementations, the benefits of composition cannot be fully observed. Specifically, the 200 LoC difference between the two implementations does not account for higher-order effects: if we consider that a production codebase may instead contain tens to hundreds of variants of the same model, the 200 LoC change quickly becomes thousands of LoC.

We instead propose to measure the extensibility of a system in terms of the *asymptotic* LoC changes required by some re-parameterization of its API. When we refer to the re-parameterization of the API, we essentially pose the question of how the LoC change when we reconfigure the system to support a different implementation signature—such as adding new functionality like the MoE layer. Note that while coding style affects absolute Lines of Code (LoC), it is less likely to change the LoC-complexity (the rate of *asymptotic* LoC growth relative to feature changes). This metric can be used to explain why composition should be favored over subtyping. The LoC-complexity(MoE) for DeepSpeed is lower-bounded by $O(N)$, where N denotes the number of modules in the system (§7): we require (1) at least one LoC change for each attention variant to subtype a `forward` implementation that replaces FFN with the MoE equivalent; (2) at least one LoC change in each ancestor module of an attention layer to incorporate the new subtyped layer. In contrast, the LoC-complexity(MoE) in AXLearn is $O(1)$: the code snippet in §4 can be used to integrate MoE without changing *any* model.

In practice, we can validate the LoC-complexity with concrete LoC counts. Indeed, in AXLearn we use the same 10-line snippet to configure MoE in over 1,000 different experiments[1]. On the other hand, if we were to adopt DeepSpeed's strategy of integrating MoE to our internal codebase, we would incur over 4,000 LoC just to modify different model variants to support their MoE counterparts.

2.2 Hardware-Agnostic Training

As one of the largest technology companies, Apple cannot feasibly rely on a single hardware platform for all of our machine learning workloads. Aside from supply constraints, our company is positioned to benefit from *not* committing to any single platform: Megatron-LM (Narayanan et al., 2021) has a vested interest in optimizing for Nvidia GPUs, while Haiku (Haiku, 2025), Flax (Heek et al., 2023), Pax (PAX team, 2022), and MaxText (MaxText team, 2025) are mostly optimized for Google TPUs.

Instead, we take the alternative approach of developing a hardware-agnostic system that can adapt seamlessly to multiple platforms, which allows our engineers to take advantage of the most cost-effective solution for their training workload. For example, while AWS Trainium2 did not exist when we first began development, AXLearn is one of the first deep learning systems that supports Trainium2 at scale.

While JAX/XLA is a key component to support hardware-agnostic training, it is not sufficient. XLA often provides reasonable out-of-the-box performance, but additional steps are needed to achieve the best performance—including providing "hints" to the XLA compiler at various points in the layer graph, using hand-tuned kernels for less mature compilation targets like GPU or Trainium2, or customizing the rematerialization strategy for each workload.

3 Overview

AXLearn allows model engineers to rapidly experiment with various model architecture and training methods across dif-

[1] An experiment refers to a training job configuration, such as the model architecture and hyper-parameters.

ferent hardware backends. AXLearn's developer interface is a hierarchical configuration of *modules*. A module can be viewed abstractly as a node in an object tree. In training, the root module is typically the trainer itself, with child modules such as the model, learner, and input, each of which may have its own children. A module's definition follows a consistent structure, including a config object that encapsulates all configurable parameters of the module.

Our view is that for ML researchers, writing configs and composing modules is much easier than implementing subtyped modules from existing ones, a view that is shared by TorchTitan (Liang et al., 2024). The pure composition approach allows replacing individual components and makes it easy to reuse components between teams or 3rd party libraries. Similar to TorchTitan, our model definition is independent of any specific parallelization strategy or trainer loop. However, from the developer experience point of view, AXLearn has several key differences in how modules are parameterized. First, AXLearn encourages hierarchical config composition over config flattening. In the example shown in Figure 1, a user script can build on top of a standard Transformer architecture by defining only the green components, and selectively replacing FFN modules with MoE equivalents without needing to inspect the details of the base Transformer architecture. In contrast, TorchTitan adopts a monolithic approach where config files "flatten" all possible configurations, including model, optimizer, checkpointer, and trainer configs. Flattening has significant ramifications in the extensibility of TorchTitan. (See §7.) Second, AXLearn uses an entirely Python-based interface, which has the benefit of allowing configs to be expressed with Python constructs like functions, loops, and recursion; as well as the advantage of being able to be directly unit-tested.

Figure 2 shows AXLearn's system diagram and workflow. AXLearn has two key components: (1) AXLearn composer and (2) AXLearn runtime. A user typically uses the layer library in AXLearn to define a training configuration. Given such a script, the AXLearn composer first materializes a full JAX program, which includes selecting the appropriate mesh shape for the desired accelerator instance, applying sharding annotations for certain layers, performing autotuning of XLA compilation options for the target hardware, selecting appropriate attention kernels for the backend, and applying appropriate rematerialization strategies based on tagged points in the module hierarchy. These annotations are crucial for training to run efficiently. The JAX program and compiler options are then passed to the XLA compiler to generate the accelerator program (including e.g. CUDA kernels), which is then orchestrated via the AXLearn runtime on distributed hardware (e.g., Kubernetes) using accelerator-specific runtimes (e.g., CUDA runtime). The AXLearn runtime monitors the execution of the accelerator program and provides additional capabilities like efficient

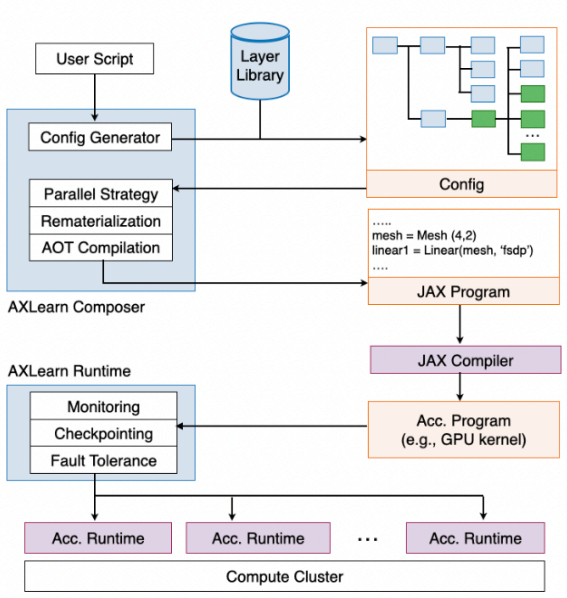


Figure 2. AXLearn's system diagram. The blue components belong to AXLearn.

checkpointing, monitoring, and fault tolerance.

4 AXLearn Composer

4.1 Modular Configuration

Hierarchical Configuration TorchTitan, DeepSpeed, Flax, and most other libraries utilize "flat" config layouts that provide a bird's-eye view of all configurations. However, a flat layout can become unwieldy as the number of layers grows, or when switching implementations of layers (in which case, one must flatten multiple combinations of configurations). In AXLearn, users instead specify training configurations via composition, which logically forms a tree. As an example, a Transformer layer's configuration consists of multiple child layer configurations:

```
class TransformerLayer(Module):
  class Config(Module.Config):
    self_attention: AttentionLayer.Config
    feed_forward: FeedForwardLayer.Config
    ...
```

Child configurations are encapsulated and can be specified independently of the parent: `TransformerLayer`'s config does not directly specify the hyperparameters of its child layers, allowing users to switch between different implementations of child layers. For instance, one can decide to replace `feed_forward` with MoE without changing the config of `TransformerLayer`. Further, since layers can be reused between models, the configs are often initially partially specified.

Note that we have not configured an input dimension on

feed_forward; instead, hidden_dim is configured to be a function of the (as of yet unspecified) input dimension. This allows the parent TransformerLayer to set input_dim when the layer is instantiated:

```
class TransformerLayer(Module):
  def __init__(self, cfg):
    # feed_forward uses the same input_dim.
    cfg.feed_forward.set(
      input_dim=cfg.input_dim)
    self._add_child("feed_forward",
      cfg.feed_forward)
```

By partially defining configs and propagating from parent to child, configurations often only need to be specified once, commonly at the layers near the root of the tree. So long as the parent and child agree on a configuration interface (often just input and output dimensions), arbitrarily complex configs can be constructed while keeping layers modular.

Config traversal Because configs are hierarchical and encapsulated, one can apply arbitrary modifications by traversing the config tree. We call such traversal a "config modifier". The illustrate, the following snippet recursively replaces any target config with new_cfg:

```
def replace_config(cfg, tgt, new_cfg):
  def enter_fn(child):
    for key, value in child.items():
      if isinstance(value, tgt.Config):
        new_cfg.set(**value.items())
        child.set(key, new_cfg)
  cfg.visit(enter_fn=enter_fn)
```

This can be used to apply MoE in the following manner:

```
# Replace any FFN with MoE.
replace_config(
  trainer_cfg,
  target=FeedForwardLayer,
  new_cfg=MoELayer.
    default_config().set(...),
)
```

Indeed, this roughly 10-line code snippet is used to apply MoE to over 1,000 experiment configs, without any additional changes to other modules.

4.2 Generating JAX Programs

Config-based parallelism AXLearn natively integrates parallelism support in every relevant layer for all common parallelism strategies, including fully-sharded data parallelism (FSDP) (Rajbhandari et al., 2020), pipeline parallelism (Huang et al., 2019), expert-parallelism (Du et al., 2024), sequence parallelism (Li et al., 2023), and tensor model parallelism (Narayanan et al., 2021). This means that users do not need to implement parallelism directly, but instead specify their desired parallelism strategy via configuration. This contrasts with Flax or PyTorch where sharding is not a native concept in the layer library, and thus code changes may be necessary depending on parallelism strategy. Users also have granular control over how specific parameters in specific layers are partitioned.

Memory optimizations AXLearn has several built-in mechanisms for optimizing high-bandwidth-memory (HBM) usage which are necessary for large-scale training. First, AXLearn layers natively support configuring rematerialization (a.k.a., activation checkpointing or remat). Depending on the hardware, the remat strategy must be tuned to selectively save or recompute certain activations in the backward pass. In AXLearn, common remat points (such as the attention QKV projections and output) are "tagged" with names, such that users can selectively target remat points to decide which activations to save in accelerator memory, offload to CPU memory (if supported by hardware), or recompute. Users can also employ programmatic remat strategies, such as only saving the output of linear layers. Second, AXLearn implements optimizer state offloading to CPU memory. This is essential for training models with more than hundreds of billions of parameters on certain platforms like TPU v5e where HBM is limited, and where sharding beyond a certain point can be inefficient.

Hardware-dependent optimizations In order to support efficient training, it is necessary to apply target-dependent parallelism strategies. AXLearn introduces the concept of "mesh rules" to address this problem. A mesh rule describes a mapping from accelerator types to config modifiers, which allows users to express complex per-target optimizations with succinct and self-contained configs. AXLearn uses these rules to automatically apply the appropriate modifiers based on the target platform. Appendix A shows a roughly 10 line code snippet that configures training to use FSDP within TPU v5e slices and data-parallel across slices, offloading activations from dot products to host memory and enabling INT8 training.

While the XLA compiler can fuse most operations to produce optimized target-specific code, in some cases we can achieve better performance with custom kernels. Again owing to the modular nature of AXLearn layers, enabling custom kernels only requires simple configuration changes. For instance, AXLearn provides a FlashAttention (Dao et al., 2022; Dao, 2023) layer, which can be used as a drop-in replacement for the default attention layer. Like above, this can be expressed as a config modifier in a mesh rule. Behind the scenes, the FlashAttention layer transparently dispatches kernels based on the backend: on GPU, cuDNN (Chetlur et al., 2014) is used when possible, falling back to a custom Pallas (pallas, 2025) kernel for cases like block-sparse at-

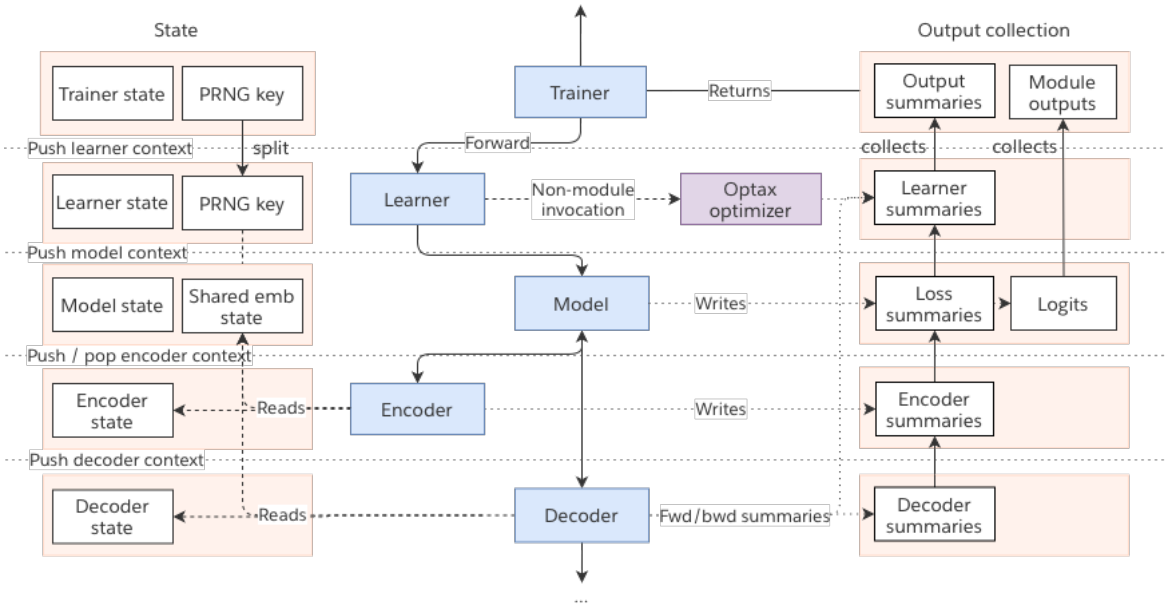


Figure 3. Invocation Context. Module invocations push contexts to the stack, which retrieve child states, split PRNG keys, and create child output collections. Upon returning, contexts are popped, collecting outputs into the parent collection. The context stack can be programmatically traversed to retrieve shared state, allowing features like tied weights to preserve encapsulation.

tention where cuDNN is not supported; on AWS Trainium, the Nki kernel from AWS Neuron Toolkit (AWS, 2025) is used; and on TPU, the SplashAttention Pallas kernel in JAX (Bradbury et al., 2018) is used.

We note that efficient hardware-agnostic training is largely possible because of several core design choices in AXLearn. First, AXLearn layers are already sharding and remat aware. This allows complex strategies to be expressed with configuration rather than code. Second, all components are implemented as strictly encapsulated modules. This allows expressing optimizations like quantization as a replacement of `DotGeneral` layers (XLA, 2025) with their quantization-aware equivalents. Third, configurations are entirely Python-based. This allows utilizing constructs like recursion to traverse and modify configs, which would otherwise not be possible with a DSL.

AOT compilation A core advantage of a compilation-based approach is that many of the errors that would otherwise be encountered in a full-scale distributed run can be checked entirely locally, from the convenience of a single host environment. AXLearn provides native support for JAX Ahead-of-Time (AOT) compilation, which allows users to analyze the memory and FLOPS utilization of a training program without executing a single line of the program, including catching errors like OOMs that would otherwise result in wasted resources. Because the same codepath is used for AOT and actual training, users can be confident that a program that AOT-compiles will run at a larger scale. This saves considerable time and resource costs when scaling development to large teams.

4.3 Maintaining States across Module Boundaries.

In order to be transformed by JAX primitives such as `jit` and `grad`, JAX programs must be purely functional, which means that they must be stateless. However, neural network training is inherently stateful: it is often necessary to store model parameters, maintain pseudo-random number generators (PRNGs), collect training summaries, and aggregate outputs. The canonical way to maintain state in a functional system is for the caller to pass in "side inputs" (such as layer parameters) along with the main inputs for the method. The outputs include not only the method results, but also "side outputs" such as summaries and state updates. Rather than requiring the user to maintain these side inputs and outputs, AXLearn introduces an abstraction called the `InvocationContext`, depicted in Figure 3. When a parent module invokes a child module, an `InvocationContext` for the child module and state is automatically pushed to the stack, which transparently "splits" the PRNG key and creates a new data store for any training summaries or outputs saved by the child module. When a child module returns, the context is popped off the stack, which transparently collects child summaries and outputs into the parent data store. Analogously to the traditional call stack, user code does not need to be aware of the `InvocationContext`. This allows users to implement modules in a familiar imperative fashion while retaining the

functional properties demanded by JAX.

A subtle design decision is that `InvocationContexts` contain references to modules, but not vice-versa. This allows contexts to be accessed outside of the module hierarchy, including from arbitrary function calls that may not have references to a module, allowing deep integration with 3rd party libraries that are not natively aware of the AXLearn state system (e.g., optax optimizers (Optax Authors, 2025)); as well as compatibility with codepaths with unique execution behavior, such as JAX's `custom_vjp` backward pass.

The design of `InvocationContext` is a key factor allowing complex architectures to be well-encapsulated. In other libraries, users may be required to perform nested instance attribute access in order to access shared state between layers. However, this requires that module implementations to be intricately aware of other modules being used and where they are in the hierarchy. In AXLearn, the system layer can instead transparently traverse the `InvocationContext` hierarchy to retrieve shared states, which allows module implementations to remain completely unaware of other modules in the system.

5 AXLearn Runtime

Monitoring and profiling. AXLearn's runtime provides observability at several layers. At the hardware layer, AXLearn natively integrates with JAX's profiler to provide insights into inefficiencies introduced by input pipelines, sharding strategies, compiler behavior, or other aspects of the program. For large scale distributed training, AXLearn supports a generic measurement interface that can be used to record arbitrary events such as the start of training or the start of a step. These events can be used to measure end-to-end inefficiencies such as those introduced by hardware provisioning or checkpointing recovery, which can be captured via metrics like overall job goodput.

Checkpointing. AXLearn's default checkpointing capabilities are similar to that of `orbax` (Orbax Authors, 2022), but supports multiple cloud storage backends, including AWS S3 and Google Cloud Storage (GCS), and provides additional memory optimizations for large-scale checkpointing. Checkpoints are saved asynchronously during training to avoid bottlenecks on slow networks, blocking only in rare cases where the checkpointer is waiting on a prior serialization to complete. A garbage collector runs in the background to remove old checkpoints according to a user-configurable policy to reduce storage costs.

Failure detection. Failures are inevitable for large-scale training spanning thousands of chips. A core problem is therefore identifying when and where a failure happens. To facilitate detection, AXLearn provides several components. First, the AXLearn runtime has a configurable "watchdog" that monitors the step time and hardware utilization of a host. Upon observing low hardware utilization or abnormal step times, the watchdog can be configured to force a restart of the host, alert an on-call for manual intervention, or dump stack traces for debugging.

Failure recovery. Efficient recovery is critical to ensure high hardware utilization and goodput. Large-scale training usually involves multiple data-parallel replicas with replicated weights. Upon a failure of a data-parallel replica, the checkpoint can be restored directly from a healthy replica and broadcasted through the fast interconnect to all other healthy or newly provisioned replicas. Applying a similar strategy, persistent compilation caches can be configured to eliminate any startup compilation time, as compilation artifacts can be entirely reused across restarts of the same model. Finally, AXLearn builds on native Kubernetes features to support slice-level hot-swap. In this case, the AXLearn scheduler over-provisions spare replicas within the same cluster, allowing failed nodes in an ongoing training job to be rapidly substituted with healthy nodes. Meanwhile, the over-provisioned hardware can still run low-priority jobs to reduce resource waste, or be sent for inspection and repair.

6 Unifying Training and Inference

One surprising discovery during the process of building AXLearn is that we can get an efficient inference engine by reusing a substantial subset of AXLearn components. Though inference performance is not our design goal, AXLearn achieves significantly higher performance than a state-of-the-art inference engine, vLLM (Kwon et al., 2023), on TPUs. (See §7 for details.)

AXLearn's design allows modules to easily be adapted to optimize for decoding. For example, because the attention layer's KV cache is an encapsulated component, an attention layer can incorporate inference-friendly cache layouts without changing the attention layer. This allows incorporating new inference techniques like continuous batching (Yu et al., 2022), disaggregated prefill and decode (Zhong et al., 2024), and paged KV caches (Kwon et al., 2023) without re-implementing models and layers. While we currently support TPU, we believe with additional effort we can support unified training and inference on other backends.

7 Evaluation

7.1 Modularity of Configurations

We analyze modularity using LoC-Complexity, motivated in §2. Specifically, we analyze the asymptotic LoC changes required to re-parameterize each system to support RoPE

Table 2. LoC analysis of deep learning systems. N denotes number of modules in a system, and M denotes number of variants of a feature. LoC Estimates represent changes necessary within each system's API interfaces to integrate a single variant of RoPE or MoE under a standard production setting. Note that in AXLearn, 0 LoC changes to existing interfaces are necessary.

System	LoC-Complexity(RoPE)	LoC-Complexity(MoE)	LoC Estimate (RoPE)	LoC Estimate (MoE)
Megatron-LM	$O(NM)$	$O(N)$	400	20
DeepSpeed	$O(NM)$	$O(NM)$	320	4000
TorchTitan	$O(NM)$	$O(NM)$	240	400
Flax	$O(NM)$	N/A	600	N/A
Praxis	$O(NM)$	$O(M)$	300	5
MaxText	$O(NM)$	$O(NM)$	200	300
AXLearn	$O(1)$	$O(1)$	0	0

Table 3. Comparing AXLearn with state-of-art systems on training performance. Systems are benchmarked on identical hardware within each model–hardware configuration. For example, the first four rows (PyTorch FSDP, Megatron, MaxText, and AXLearn) are evaluated on a 32 × H100-8 cluster.

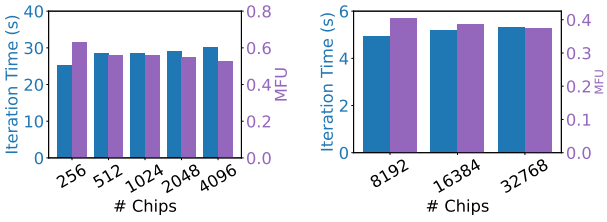
Model	Hardware	System	Iteration Time (s)	MFU	Throughput (tokens/s)
Llama2-7B	32 × H100-8	PyTorch FSDP	2.6	29.9%	1.6M
		Megatron-LM	1.7	44.9%	2.5M
		MaxText	1.4	54.7%	3.0M
		AXLearn	1.4	54.2%	3.0M
	tpu-v5p-512	PyTorch XLA FSDP	3.5	46.7%	1.2M
		MaxText	2.7	61.6%	1.6M
		AXLearn	2.5	66.2%	1.7M
	64 × Trainium2-16	**AXLearn**	1.2	24.2%	3.5M
Llama2-70B	64 × H100-8	PyTorch FSDP	10.6	34.7%	396K
		Megatron-LM	7.8	47.2%	538K
		MaxText	9.4	39.1%	446K
		AXLearn	9.2	40.0%	456K
	tpu-v5p-1024	PyTorch XLA FSDP	OOM		
		MaxText	12.3	64.4%	341K
		AXLearn	11.6	68.0%	360K
	64 × Trainium2-16	**AXLearn**	11.2	25.0%	374K
Qwen-3 30B-A3B	tpu-v5p-1024	MaxText	13.0	31.3%	1.3M
		AXLearn	12.9	31.6%	1.3M
	64 × B200-8	Megatron-LM	4.1	20.2%	4.1M
		AXLearn	4.3	19.2%	3.9M

and MoE. When measuring extensibility, we focus on LoC changes incurred in *existing* modules in the system as opposed to the new functionality itself. This allows for a comparable analysis, and in practice, the same feature across systems tends to incur similar LoC up to some constant.

Table 2 shows the summary of LoC-Complexities across systems. We omit PyTorch FSDP, which is not an LLM library; Haiku, which has no implementation of RoPE or MoE; and Pax, which is replaced with its layer library Praxis. For a concrete evaluation, we also provide LoC estimates based on the changes required in a specific production codebase. While many of the listed systems provide a few sample implementations, a production codebase can be orders of magnitude larger. We cannot disclose specific details of the scale of our internal model architectures and hyper-parameters. Therefore, we provide estimates under a realistic setting of a codebase with 20 model variants (e.g., variants of a GPT model) and 10 variants of attention (i.e., kernels, kv cache strategies, etc.). **Notation:** Let N be the number of modules in a system, and M be the number of variants of the feature to be added. For example, MoE has variants of routing or gating mechanisms.

AXLearn (ours) In AXLearn, RoPE and MoE are strictly encapsulated. §4 provides a 10-line code snippet to integrate both features into any experiment config. Internally we use such a code snippet to configure over 1,000 experiments to use RoPE, MoE, or both. As we scale the number of modules or RoPE or MoE variants, we require no changes to any existing interfaces, achieving constant LoC-Complexity.

Other Systems All other systems we have studied require $O(NM)$ to incorporate RoPE and MoE, with the following exceptions. Like AXLearn, Megatron-LM defines a MoE layer in place of the standard MLP. However, each MLP



(a) Model A (70B) (b) Model B (150B)

Figure 4. AXLearn's training throughput and MFU when scaling the number of TPUs.

introduces MoE-specific fields in its signature, so that any module composing a linear submodule incurs at least one LoC change for MoE-specific arguments. This scales with $O(N)$. Like AXLearn, Praxis uses a "template" composition approach, which allows LoC-Complexity(MoE) to be $O(M)$: each MoE variant incurs a small 5 LoC change. Flax has no public example of MoE so we exclude it from our analysis. See Appendix B for details on the LOC analysis.

7.2 Performance on Different Hardware

We compare AXLearn training performance with PyTorch FSDP, Megatron-LM, and MaxText, which achieve state-of-the-art performance on GPU and TPU. All comparisons are conducted on identical hardware to ensure fair evaluation—systems sharing a hardware group in Table 3 are benchmarked on the same cluster with the same number of accelerators. We use training iteration time, model FLOPS utilization (MFU), and throughput to represent a training system's performance. We evaluate both dense and MoE models: Llama2 7B and 70B (dense), and Qwen-3 30B-A3B (MoE), on four hardware backends: (1) 256/512 H100 GPUs (32/64 AWS P5d instances, each with 8 H100 GPUs); (2) 512 B200 GPUs (64 AWS P6 instances, each with 8 B200 GPUs); (3) TPU-v5p-512/1024 (64/128 GCP Cloud TPU hosts, each with 4 chips); and (4) 1024 Trainium2 (64 AWS trn2 instances, each with 16 Trainium2 chips). All runs use a global batch size of 1024. Note that since Pytorch FSDP does not run on TPU, we use Pytorch XLA FSDP as its closest replacement (PyTorch, 2025). Megatron-LM is a GPU-only system, so we only report results on GPU. None of the baselines support Trainium, so we only report results for AXLearn.

Table 3 summarizes the performance results. On TPUs, AXLearn achieves state-of-the-art performance, where MaxText lags slightly behind, likely due to its choices of rematerialization. On the other hand, PyTorch XLA FSDP fails to run entirely with out-of-memory errors. On H100 GPUs, AXLearn outperforms PyTorch FSDP due to fine-grained control over activation checkpointing and better support for memory bound operations such as RMSNorm and RoPE.

Table 4. Comparing AXLearn with vLLM on inference performance for Llama 2 models on Time-To-First-Token (TTFT), Time-Per-Output-Token (TPOT), and throughput on TPUs.

Model	System	TTFT	TPOT	Throughput
7B	vLLM	538.6ms	22.4ms	1117 tokens/s
	AXLearn	40.1ms	9.1ms	3125 tokens/s
70B	vLLM	80s	189.8ms	705 tokens/s
	AXLearn	150.5ms	28.1ms	1139 tokens/s

Megatron-LM has stronger performance on H100 and B200 GPUs over AXLearn because PyTorch currently have finer-grained scheduling capability over XLA. However, using XLA allows AXLearn to be hardware-agnostic. This is a trade-off we are willing to take.

To test AXLearn's scalability, we perform a weak-scaling study of two production models. Model A is a 70B parameter model with 4,096 context length, and Model B is a 150B parameter model with 8,192 context length. Fixing the per-device batch size, for Model A, scaling from 256 to 4,096 chips reduces MFU from 63.0% to 52.4%; and for Model B, scaling from 8,192 to 32,768 chips reduces MFU from 40.6% to 37.6%. These numbers demonstrate that AXLearn achieves very close to linear scaling, shown in Figure 4. We note that MFU for the 150B model is lower due to the need to limit the global batch size at 32,768 chip scale for good training convergence. All scaling experiments for the 150B model has 1/16 per chip sequence length compared to 70B model experiments.

A key benefit of AXLearn's modular design is that the same training codebase can serve inference workloads with minimal additional effort. We compare the inference performance in terms of Time-To-First-Token (TTFT) and Time-Per-Output-Token (TPOT) of Llama2 7B and 70B models between AXLearn and vLLM on TPUs. We use the ShareGPT dataset for the prompts. For the 7B model, the benchmark is performed on a Google Cloud TPU v5p-8 VM instance, with maximum input length of 1024 and maximum output length of 256. For the 70B model, the benchmark is on a Google Cloud TPU v6e-8 VM instance[2], with maximum input length of 1,800 and maximum output length of 256. As Table 4 shows, AXLearn achieves 500x and 6x speedup in TTFT and TPOT over vLLM, respectively. In terms of throughput, AXLearn is 2.8x faster for 7B model inference and 1.6x faster for 70B model inference. We note that TPU support for vLLM is still experimental, which likely contributes to the performance gap. Our goal is not to position AXLearn as a specialized inference engine, but to demonstrate that a modular training framework can achieve production-grade inference performance with minimal addi-

[2]vLLM does not support the 70B model on v5p-8 VM at the time the experiments were run.

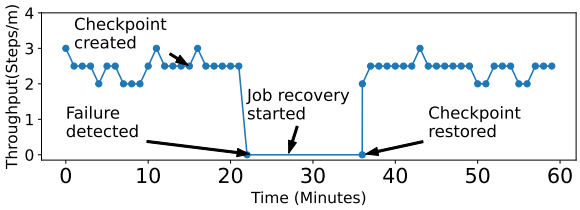


Figure 5. AXLearn's recovery latency under a hardware failure.

tional effort.

7.3 Failure Recovery Latency

We measure the speed of recovering from a hardware failure via analyzing a production training job. Figure 5 shows training throughput over time on 32,768 TPUs over a 1-hour window that contains a complete event which the job recovers from a hardware failure. Since checkpointing is performed asynchronously, no training throughput reduction is observed during checkpoint creation. When hardware fails, AXLearn detects the training job has failed immediately and initiates slice-level hot-swap, completing the hot-swap within 4 minutes. Checkpoint restoration finishes in 9 minutes after the hot-swap completes. In total, the failure results in 21 minutes of lost training time, which includes both the downtime during hot-swap and checkpoint restoration, as well as the lost progress from training step completed after the most recent checkpoint before the failure occurred.

7.4 Experience of using AXLearn at Apple

AXLearn's development began in late 2021 using PyTorch with a small team of engineers. At the time, signs of the Transformer architecture becoming a de facto choice were evident (e.g., BERT (Devlin et al., 2019), T5 (Raffel et al., 2020), GPT3 (Brown et al., 2020)). Architectural convergence meant that most layers in many new models can be reused if implemented modularly. However, due to limited support for automatic parallelization, layer implementations ultimately could not be fully modularized—in many cases, different parallelization strategies meant that logic had to be rewritten, with inevitable interactions across modules.

GSPMD (Xu et al., 2021) (published late 2021) meant that communication can instead be injected by a compiler. We chose to deeply integrate with XLA as a bet on the compiler-first approach. First, layers can be modularized as sharding propagation across the graph can be automatically and transparently handled. Second, we believed eventual adoption would push hardware providers to optimize compilers on their platforms, which means performance improvements can be achieved without a single LoC. As it turns out, while JAX/XLA did not initially support GPUs, today it achieves competitive performance across AWS and GCP; and is now natively supported by platforms like Trainium2. Where compilers fall short, we rely on custom kernels to close the performance gap. While AXLearn's modular design makes integrating custom kernels straightforward—as drop-in replacements via config modifiers—achieving peak hardware utilization on each backend requires backend-specific kernel implementations, which demands specialized expertise and ongoing maintenance as hardware evolves.

Adopting JAX/XLA meant several paradigm shifts. First, it shifts from imperative to functional programming, which posed a challenge as familiar patterns like mutable state were prohibited. Instead, developers manually specified parameters as inputs to each module, and bubbled up outputs through the call stack. This hurt modularity as it required implementations to be intricately aware of state structure, and hurt usability as users lost the comfort of the PyTorch imperative style. This motivated the design of `InvocationContext`, which allowed state to be decoupled, outputs to be collected across the module hierarchy, and layer implementations to be implemented in familiar PyTorch fashion. Second, it shifts focus to the cloud. JAX/XLA initially only supported Google Cloud TPU, which meant that a number of internal components could not be directly used. As GCP was in a nascent state with limited TPU capacity, we knew we could not rely on it entirely, and hoped to leverage our internal compute infrastructure. To that end, we designed the system to allow all components of training to be configurable. For example, we initially leveraged Flax's GCS-compatible checkpointer to bootstrap the layer library, which we later replaced to support internal storage backends. This migration was completely seamless with no changes to user code, which was only possible due to AXLearn's modular design.

As we scaled development, we faced additional barriers. First, avoiding resource contention was crucial due to limited TPU capacity within GCP. Aside from leveraging other backends, we realized that a significant portion of resources were consumed by jobs with low resource utilization or no progress due to preventable errors. However, XLA compilation inherently operates on device abstractions—many errors (e.g., OOMs due to suboptimal sharding) can be theoretically caught from a local machine. We therefore capitalized on JAX's compiler-first approach by deeply integrating AOT-compilation, which allowed users to debug training entirely on CPU. This has allowed development to scale even with limited capacity. Second, we learned the hard way that ML testing practices are often insufficient. While it is common to unit test layers, ML experiments heavily depend on correctness of configs. With a large codebase, changes to one experiment can inadvertently affect others, which are difficult to catch with unit or integration testing (e.g., subtle changes in training dynamics). To address this

issue, AXLearn introduces the notion of "golden configuration" tests: key training configs are serialized into human readable format and committed along with code changes. This allows changes to produce consistent and reviewable diffs, to trigger relevant code-owner reviews, avoid code-contribution-level conflicts, and provide a traceable commit history of experiments. Lastly, public cloud infrastructure can fail in opaque ways. In contrast to an internally managed cluster, we learned that we must account for opaque failures often out of our control, including hardware failures, ICI failures, silent data corruption, kernel panics, file system throttling, and more. While we have built many layers of resiliency, debugging and mitigating many of these failures required close collaboration with Google, Amazon, and Nvidia.

Today, AXLearn has grown from a handful of developers training models at million parameter scales, to hundreds of developers training models at billion-to-trillion parameter scales. It actively supports over 10,000 experiments under development at a given time, running across tens of different hardware clusters. Some of the models trained with AXLearn now power features used by over a billion users, including intelligent assistants, multimodal understanding and generation, and code intelligence.

8 Discussion

Tradeoffs between development agility and modularity. Training systems inevitably need to balance development agility and modularity. Directly modifying Python execution code offers the highest degree of agility, as it allows arbitrary changes to be introduced quickly. In AXLearn, we instead encourage users to configure parameterized modules, which can reduce agility in the short term. Our perspective is that, across many model architectures, a large number of components are structurally similar (e.g., RoPE, MoE). Implementations of these building blocks benefit from being shared across architectures and reused across experiments, including those that change hyperparameters such as model dimension or number of experts. When existing modules are insufficient, developers can still implement new modules in AXLearn and integrate them seamlessly. While no framework is universally optimal, our model engineers have conducted extensive experiments exploring a wide range of model architectures and training methods. To date, we have not encountered cases where AXLearn's modular design constrained the specification of model architectures or training approaches.

Is AXLearn a complete answer for hardware-agnostic training? AXLearn enables training to run across different hardware platforms. However, similar to other training systems, achieving peak performance still depends on the availability of custom kernels. This reflects current limitations in tensor compilers, which often cannot automatically derive highly optimized kernels (e.g., FlashAttention (Dao et al., 2022; Dao, 2023)) from high-level specifications alone. In AXLearn, such custom kernels are treated as black-box nodes within the computation graph, ensuring they remain encapsulated and do not interfere with higher-level configuration or composability. A key strength of AXLearn is that it allows a wide range of machine learning experiments, such as exploring new model architectures or training methods, to leverage these kernels with minimal code changes across different experiments. This helps reduce friction in experimentation while maintaining strong performance.

Is AXLearn's performance on GPUs and Trainium limited by XLA? AXLearn is built on top of XLA, and AXLearn therefore inherits its backend performance characteristics. In addition to XLA compiler, XLA provides custom calls as an escape hatch to allow experts to write hand-tuned kernels that outperform compilers. On GPUs, we demonstrate that integrating custom kernels (e.g., FlashAttention) enables AXLearn to achieve performance comparable to other state-of-the-art training systems. For emerging hardware platforms such as Trainium, the broader ecosystem support across training frameworks is still evolving, making comprehensive comparisons an ongoing area of investigation. By relying on XLA, we prioritize a hardware-agnostic approach, which we view as a pragmatic and forward-looking design choice.

9 Related Work

Training systems for large models. High-performance large model training is an active research area. FSDP (Zhao et al., 2023) and DeepSpeed (Rajbhandari et al., 2020) use sharding to reduce GPU memory usage. Megatron-LM (Narayanan et al., 2021) combines data, tensor-model, and pipeline parallelism to train LLMs efficiently on GPUs. MegaScale (Jiang et al., 2024) further scales LLM training to more than 10,000 GPUs and deals with straggler and failures in GPU clusters. AXLearn's focus is complementary to these systems, with the goal of enabling engineers to use these techniques with minimal effort.

Software modularity. Maintaining a modular structure is a standard software engineering practice in any complex computer system, facilitating maintenance and integration of new features (Bershad et al., 1995; Kohler et al., 2000). Modularity is especially important for a fast-moving machine learning system: at Apple, we iterate on new model architectures, hyper-parameters, and training methods every day. To facilitate this development, configurations must be separated from system details like parallelism or rematerialization in the simplest way possible, so that researchers can

focus on model iteration.

Configuration systems for distributed workloads. One of our contributions is the ability to configure distributed deep learning execution in a modular and automated manner. Configuration for distributed systems is a topic beyond deep learning. Meta's configuration system (Tang et al., 2015) can enable easy A/B testing, similar to how we conduct A/B testing on model accuracy and training performance by comparing AXLearn's configurations. However, our main design goal for the configuration system is to enable modules to be re-parametrized and composed to easily integrate with new features (e.g., RoPE, MoE).

10 Conclusion

We have developed AXLearn, a modular, hardware-agnostic large model training system. AXLearn maintains constant complexity as the number of modules scales. Further, our evaluations show that AXLearn has achieved comparable performance compared to other state-of-the-art training systems. Finally, we shared our development and operation experiences with AXLearn.

Acknowledgments

We thank the anonymous reviewers and our shepherd for their helpful feedback on the paper. We thank Yi Wang, Ke Ye, Dong Yin, Muyang Yu, Yi Zhang, and other key members of the Apple Foundation Model team, for their deep collaboration and valuable feedback in the development of AXLearn. We also thank Benoit Dupin and Daphne Luong for their leadership support.

References

Xla: Optimizing compiler for machine learning., 2025. URL https://openxla.org/xla/tf2xla.

AWS. Neuron documentation, 2025. URL https://awsdocs-neuron.readthedocs-hosted.com/en/v2.21.0.beta/general/nki/api/generated/nki.kernels.flash_fwd.html.

Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C., and Eggers, S. Extensibility safety and performance in the spin operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pp. 267–283, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897917154. doi: 10.1145/224056.224077. URL https://doi.org/10.1145/224056.224077.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/jax-ml/jax.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners, 2020. URL https://arxiv.org/abs/2005.14165.

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. cudnn: Efficient primitives for deep learning, 2014. URL https://arxiv.org/abs/1410.0759.

Cursor. The ai code editor, 2025. URL https://www.cursor.com/.

Dao, T. Flashattention-2: Faster attention with better parallelism and work partitioning, 2023. URL https://arxiv.org/abs/2307.08691.

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022. URL https://arxiv.org/abs/2205.14135.

DeepSpeed. Model implementations, 2025a. URL https://github.com/deepspeedai/DeepSpeed/tree/a21e5b9db68adf25e9fc797d0e67fdb5879f6069/deepspeed/inference/v2/model_implementations.

DeepSpeed. Mixture of experts - deepspeed, 2025b. URL https://www.deepspeed.ai/tutorials/mixture-of-experts/#moe-layer-api.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. In *North American Chapter of the Association for Computational Linguistics*, 2019. URL https://api.semanticscholar.org/CorpusID:52967399.

Du, X., Gunter, T., Kong, X., Lee, M., Wang, Z., Zhang, A., Du, N., and Pang, R. Revisiting moe and dense speed-accuracy comparisons for llm training, 2024. URL https://arxiv.org/abs/2405.15052.

Google. Gemini: Our most intelligent ai models, built for the agentic era, 2025. URL https://deepmind.google/technologies/gemini/.

Haiku. Haiku: Sonnet for jax, 2025. URL https://github.com/google-deepmind/dm-haiku.

Heek, J., Levskaya, A., Oliver, A., Ritter, M., Rondepierre, B., Steiner, A., and van Zee, M. Flax: A neural network library and ecosystem for JAX, 2023. URL http://github.com/google/flax.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, M. X., Chen, D., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., and Chen, Z. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

Jiang, Z., Lin, H., Zhong, Y., Huang, Q., Chen, Y., Zhang, Z., Peng, Y., Li, X., Xie, C., Nong, S., Jia, Y., He, S., Chen, H., Bai, Z., Hou, Q., Yan, S., Zhou, D., Sheng, Y., Jiang, Z., Xu, H., Wei, H., Zhang, Z., Nie, P., Zou, L., Zhao, S., Xiang, L., Liu, Z., Li, Z., Jia, X., Ye, J., Jin, X., and Liu, X. MegaScale: Scaling large language model training to more than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pp. 745–760, Santa Clara, CA, April 2024. USENIX Association. ISBN 978-1-939133-39-7. URL https://www.usenix.org/conference/nsdi24/presentation/jiang-ziheng.

Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, August 2000. ISSN 0734-2071. doi: 10.1145/354871.354874. URL https://doi.org/10.1145/354871.354874.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pp. 611–626, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Li, S., Xue, F., Baranwal, C., Li, Y., and You, Y. Sequence parallelism: Long sequence training from system perspective. In *ACL*, pp. 2391–2404. Association for Computational Linguistics, 2023. ISBN 978-1-959429-72-2.

Liang, W., Liu, T., Wright, L., Constable, W., Gu, A., Huang, C.-C., Zhang, I., Feng, W., Huang, H., Wang, J., Purandare, S., Nadathur, G., and Idreos, S. Torchtitan: One-stop pytorch native solution for production ready llm pre-training, 2024. URL https://arxiv.org/abs/2410.06511.

MaxText team. Maxtext., 2025. URL https://github.com/AI-Hypercomputer/maxtext.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., Phanishayee, A., and Zaharia, M. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384421. doi: 10.1145/3458817.3476209. URL https://doi.org/10.1145/3458817.3476209.

OpenAI. Introducing chatgpt, 2022. URL https://openai.com/index/chatgpt/.

Optax Authors, 2025. URL https://github.com/google-deepmind/optax.

Orbax Authors, 2022. URL https://github.com/google/orbax.

pallas. Pallas: a jax kernel language., 2025. URL https://docs.jax.dev/en/latest/pallas/index.html.

PAX team. Framework to configure and run machine learning experiments on top of jax., 2022. URL http://github.com/google/paxml.

PyTorch. Fully sharded data parallel in pytorch xla, 2025. URL https://pytorch.org/xla/master/perf/fsdp.html.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21(1), January 2020. ISSN 1532-4435.

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. Zero: memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020. ISBN 9781728199986.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q. V., Hinton, G. E., and Dean, J. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *ICLR (Poster)*, 2017. URL http://dblp.uni-trier.de/db/conf/iclr/iclr2017.html#ShazeerMMDLHD17.

Su, J., Ahmed, M., Lu, Y., Pan, S., Bo, W., and Liu, Y. Roformer: Enhanced transformer with rotary position embedding. *Neurocomput.*, 568(C), February 2024. ISSN 0925-2312. doi: 10.1016/j.neucom.2023.127063. URL https://doi.org/10.1016/j.neucom.2023.127063.

Tang, C., Kooburat, T., Venkatachalam, P., Chander, A., Wen, Z., Narayanan, A., Dowell, P., and Karl, R. Holistic configuration management at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pp. 328–343, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338349. doi: 10.1145/2815400.2815401. URL https://doi.org/10.1145/2815400.2815401.

XLA. Operation semantics - openxla project, 2025. URL https://openxla.org/xla/operation_semantics.

Xu, Y., Lee, H., Chen, D., Hechtman, B., Huang, Y., Joshi, R., Krikun, M., Lepikhin, D., Ly, A., Maggioni, M., Pang, R., Shazeer, N., Wang, S., Wang, T., Wu, Y., and Chen, Z. Gspmd: General and scalable parallelization for ml computation graphs, 2021. URL https://arxiv.org/abs/2105.04663.

Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., and Chun, B.-G. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/yu.

Zhao, Y., Gu, A., Varma, R., Luo, L., Huang, C.-C., Xu, M., Wright, L., Shojanazeri, H., Ott, M., Shleifer, S., Desmaison, A., Balioglu, C., Damania, P., Nguyen, B., Chauhan, G., Hao, Y., Mathews, A., and Li, S. Pytorch fsdp: Experiences on scaling fully sharded data parallel. *Proc. VLDB Endow.*, 16(12):3848–3860, August 2023. ISSN 2150-8097. doi: 10.14778/3611540.3611569. URL https://doi.org/10.14778/3611540.3611569.

Zheng, L., Li, Z., Zhang, H., Zhuang, Y., Chen, Z., Huang, Y., Wang, Y., Xu, Y., Zhuo, D., Xing, E. P., Gonzalez, J. E., and Stoica, I. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 559–578, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1.

Zhong, Y., Liu, S., Chen, J., Hu, J., Zhu, Y., Liu, X., Jin, X., and Zhang, H. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 193–210, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin.

Zoom. Work happy with zoom ai companion, 2025. URL https://www.zoom.com/en/products/ai-assistant/.

A Mesh Rules

In AXLearn, users can specify per-target platform configuration changes using "mesh rules". These rules are mappings from instance type regular expressions to config modifiers discussed in §4. In the example below, when launching training on TPU v5e, we configure training to use FSDP within-slices and data-parallel across slices, offloading activations from dot products to host memory and enabling INT8 training. On the other hand, when launching training on H100s, we instead switch to 8-way tensor parallelism within node and FSDP across nodes, saving query, key, value, and output (QKVO) projections to HBM, as well as enabling FP8 training with delayed scaling.

These configs are all that are necessary to apply per-target optimizations, allowing users to scale training on different platforms with ease.

```
[("tpu-v5e-256-*",
  [MeshShapeModifier.default_config().set(
      mesh_shape=mesh(data=-1, fsdp=256)),
    RematSpecModifier.default_config().set(
      remat_policies={
        "model.decoder.transformer.layer":
          RematSpec(policy=offload_dots)}),
    INT8ConfigModifier.default_config()]),
("gpu-H100-*",
  [MeshShapeModifier.default_config().set(
      mesh_shape=mesh(fsdp=-1, model=8)),
    RematSpecModifier.default_config().set(
      remat_policies={
        "model.decoder.transformer.layer":
          RematSpec(policy=save_qkvoflash)}),
    FP8ConfigModifier.default_config().set(
      fp8_amax_history_length=128)])]
```

B LoC Analysis

We provide additional details and rationale of how we derived the LoC estimates in Table 2.

Megatron-LM Megatron-LM's Transformer implementation composes `TransformerBlockSubmodules`. However, the RoPE-specific parameters are flattened in the `init` signature of each model implementation, e.g. `rotary_percent`, `rotary_base`, `rotary_scaling`, and `position_embedding_type` in `GPTModel`. Additionally, these parameters are propagated to submodules like `TransformerBlock`, `Transformer Layer`, or `Attention`.

This means to integrate a RoPE variant, one potentially incurs LoC changes to each module in each model imple-

mentation, as at minimum the RoPE parameters must be propagated down an arbitrary number of modules to the `Attention` layer.

Additionally, each model's `init` implementation includes branching logic to instantiate the RoPE embedding layer variant depending on the desired `position_embedding_type`. For example, `RotaryEmbedding` should be instantiated if the embedding type is "rope", while `MultimodalRotaryEmbedding` should be instantiated when embedding type is "mrope". Therefore, if we additionally consider RoPE variants, the LoC-complexity scales quadratically to $O(NM)$, as in the worst case each module must account for each variant in its `init` signature if it receives an embedding type.

To integrate MoE, Megatron-LM is able to leverage composition via `TransformerBlockSubmodules` to specify a MoE layer in place of the standard `MLP`. However, once again the encapsulation is not applied strictly. Each MLP layer implementation introduces an is_expert field in its init signature, which is propagated to the linear submodules. Any module that uses a linear submodule therefore needs to incur at least one LoC change, which scales with $O(N)$. Indeed, Megatron-LM itself has a number of modules that are impacted, including `ColumnParallelLinear`, `RowParallelLinear`, `Attention`, `CrossAttention`, and more.

If we assume the production setting of 20 model variants, with each conservatively incurring at least 20 LoC changes to integrate RoPE[3], we incur at least 400 LoC. For MoE, if we assume 10 MLP variants[4], each incurring at least 1 LoC change to integrate support for `is_expert`, we incur at minimum 10 LoC change. In addition, if we assume at least 10 modules in the system using a linear submodule (corresponding to each linear variant), each requiring 1 LoC change to `build_module`, we incur additional 10 LoC.

DeepSpeed DeepSpeed applies the "config flattening" methodology discussed in §4. RoPE configs like `rotary_dim` and `rope_theta` are grouped under a monolithic config class, like `DeepSpeedInferenceConfig`. Every model implementation reads the config and overrides the method `positional_embedding_type` to indicate whether RoPE should be enabled for the model by returning a specific value of the embedding type. With this design, we can already observe that LoC-Complexity(RoPE) must be *at least* $O(N)$, as each model implementation incurs LoC changes to override the necessary methods to enable RoPE[5].

In addition, the base model implementation propagates this embedding type (and additional RoPE configs) to child layers like the self attention layer. As a consequence, every attention layer implementation, such as `DSDenseBlockedAttention`, must first update its `init` signature to handle the input embedding type. It must also update its forward implementation to apply RoPE based on the embedding type.

In a similar way as Megatron-LM, LoC-complexity(RoPE) scales quadratically with $O(NM)$, because we must propagate embedding type and RoPE parameters down an arbitrary number of modules to the attention layer, and because each attention module in the worst case receives and must be prepared to handle all possible values of embedding type.

If we assume conservatively 20 models with at least 6 LoC per model[6], we incur 120 LoC; in addition, with 10 attention variants each requiring approximately 20 LoC[7] we incur another 200 LoC.

If the MoE case, DeepSpeed requires subclassing each model from a custom subclass `DSMoETransformerModelBase`, which requires in some cases a re-implementation of most methods. For 20 model variants, each can incur 100s of LoC—in the case of DeepSpeed's QwenV2MoE, more than 200 LoC—which conservatively incurs 4,000 LoC changes.

TorchTitan TorchTitan adopts a similar design as DeepSpeed by using a monolithic `BaseModelArgs` subclass that flattens all configs for each model. In the case of RoPE, each such config subclass introduces RoPE specific configurations like `rope_theta` and `rope_scaling`. Similar to prior analysis, such a design already incurs at least $O(N)$ LoC-Complexity as each model adopting RoPE must accordingly modify its config class signature.

In addition, each model has its own `Attention` implementation. For example, the `deepseek_v3 Attention` implementation conditions on the value of RoPE configs (e.g. `rope_scaling`) to decide which child RoPE layer to instantiate. In the worst case, each RoPE variant may incur the same conditional logic for each model implementation that intends to support it. This causes the LoC-Complexity to degrade to $O(NM)$ if we consider RoPE variants.

With 20 model variants, each incurs 2 LoC on average to update its corresponding `ModelArgs` class to incorporate RoPE configs, as well as 10 LoC for each attention implementation. In total, this incurs 240 LoC, although

[3] Based on the changes required in `GPTModel`.

[4] Based on Megatron-LM's own Linear variants.

[5] E.g., `positional_embedding_type` and `positional_embedding_config`

[6] To override two properties.

[7] Based on changes in `DSDenseBlockedAttention`.

we note that this varies greatly between TorchTitan model implementations[8].

In the MoE case, TorchTitan conditionally instantiates either a MoE child layer or a standard `FeedForward` layer, based on the config field `moe_enabled`. Because TorchTitan implements customized layers for each model (e.g., `TransformerBlock` for Llama, `DecoderLayer` for DeepSeekV3), this results in $O(NM)$ complexity as we must modify the corresponding Transformer layer for each model for each MoE variant.

Like with RoPE, we incur at least 10 LoC for each `ModelArgs` (due to more complex configuration of MoE), as well as 10 LoC for each attention layer implementation[9]. Over 20 model variants, this results in 400 LoC changes.

Flax Flax implements the Gemma model via mostly self-contained Transformer modules which are not shared with other implementations. To integrate RoPE, the Gemma `TransformerConfig` is modified to add the RoPE parameters; the Gemma `Transformer` module is modified to propagate the RoPE parameters from the config to its `Block` submodules; each `Block` submodule is modified so that its `init` signature can accommodate additional RoPE parameters; and each `Attention` module must take these RoPE parameters to finally implement the RoPE logic. It is easy to see that the LoC-Complexity(RoPE) in this case is $O(NM)$, as for each RoPE variant, one would need to correspondingly update RoPE parameters in the top-level `TransformerConfig`, and then propagate those configs down an arbitrary number of modules to the `Attention` layer implementation. For a single variant, the Gemma model incurs at least 30 LoC across `TransformerConfig`, `Transformer`, `Block`, and `Attention` parameterization modifications, not including the actual RoPE implementations. For 20 model variants, this results in at least 600 LoC changes.

Praxis Praxis is the layer library of Pax. It internally uses fiddle, a config system similar to the one in AXLearn, which allows it to express certain re-parameterizations using composition. For example, Praxis uses a "template" approach to configure the MoE layer in each Transformer layer stack. This "template" is configured along with several MoE configs (like `num_experts`) in the `StackedTransformerLayer` definition, which in theory allows the integration of MoE to scale with $O(M)$, or the number of MoE variants. Because some MoE configs are still flattened, although not to the extent of causing quadratic interactions, each MoE variant incurs 5 LoC change[10].

However, using a composable config system does not guarantee strict encapsulation. In particular, Praxis flattens a number of RoPE-specific configs (e.g., `use_rotary_position_emb`) into each attention layer implementation, which means that it incurs at least $O(N)$ LoC-Complexity for a single RoPE variant. As it turns out, Praxis attention layers compose the actual RoPE implementation itself via defining a RoPE layer "template" `rotary_position_emb_tpl`. However, because each RoPE variant may have different configuration interfaces, the flattening ultimately means that LoC-Complexity under RoPE variants is $O(NM)$, as each RoPE variant may nevertheless require modifications to each attention layer.

For 10 attention variants[11], each incurring approximately 30 LoC[12], this incurs at least 300 LoC.

MaxText MaxText builds on top of Flax modules, and follows a similar layer design and analysis. MaxText's `Attention` conditions on configs (like `attention_type` and `rope_type`) to choose the RoPE module. This has undesirable interactions between the attention and RoPE variants; in the worst case, `Attention` must account for the cross-product of attention and RoPE variants. We can already observe this with MaxText's MLA implementation; while MLA implemented as a subtype, its RoPE logic must also be handled in its parent Attention. For each RoPE variant and model, we can expect at least 10 LoC[13], which results in 200 LoC across 20 variants.

In a similar way, MaxText's MoE implementation details are flattened into each model's decoder. Each model implements its own `DecoderLayer`, while a monolithic `Decoder` selects a decoder layer implementation based on the name of the model in the config. With 10 LoC per decoder[14], this results in 200 LoC for 20 model variants. MaxText's trainer also includes MoE-specific logic: each loss function uses MoE configs to apply auxiliary losses. With 5 LoC for each loss function[15], across 20 model variants this results in additional 100 LoC.

[8] Based on DeepSeekV3 and Llama4 implementations.

[9] Based on `TransformerBlock` and `DecoderLayer`.

[10] Based on the number of flattened configs.

[11] Based on number of attention layers in Praxis

[12] Based on `DotProductAttention` and `MultiQueryDotProductAttention`

[13] Based on Attention implementation for "llama3.1", "yarn", etc.

[14] Based on changes in Decoder.

[15] Based on `loss_fn` and `dpo_loss_fn`.