# The JAX Ecosystem: A Modular, Scalable, and High-Performance ML Ecosystem

## A Modular, Compiler-First Architecture for Modern AI

The JAX AI stack extends the JAX numerical core with a collection of Google-backed composable libraries, evolving it into a robust, end-to-end, open-source platform for Machine Learning at extreme scales. As such, the JAX AI stack consists of a comprehensive and robust ecosystem that addresses the entire ML lifecycle:

- **Industrial-Scale Foundation:** The stack is architected for massive scale, leveraging ML Pathways for orchestrating training across tens of thousands of chips and Orbax for resilient, high-throughput asynchronous checkpointing, enabling production-grade training of state-of-the-art models.
- **Complete, Production-Ready Toolkit:** It provides a comprehensive set of libraries for the entire development process: Flax for flexible model authoring and "surgery," Optax for composable optimization strategies, and Grain for the deterministic data pipelines essential for reproducible large-scale runs.
- **Peak, Specialized Performance:** To achieve maximum hardware utilization, the stack offers specialized libraries including Tokamax for state-of-the-art custom kernels, Qwix for non-intrusive quantization that boosts training and inference speed, and XProf for deep, hardware-integrated performance profiling.
- **Full Path to Production:** The stack provides a seamless transition from research to deployment. This includes MaxText as a scalable reference for foundation model training, Tunix for state-of-the-art reinforcement learning (RL) and alignment, and a unified inference solution via vLLM-TPU integration and the native JAX Serving runtime.

The JAX ecosystem philosophy is one of loosely coupled components, each of which does one thing well. Rather than being a monolithic ML framework, JAX itself is narrowly-scoped and focuses on efficient array operations and program transformations. The ecosystem is built upon this core framework to provide a wide array of functionalities, related to both the training of ML models and other types of workloads such as scientific computing.

This system of loosely coupled components hands freedom of choice back to users, enabling them to select and combine libraries in the best way to suit their requirements. From a software engineering perspective, this architecture also allows parts that would traditionally be considered core framework components (for example data pipelines, checkpointing, etc.) to be iterated upon rapidly without the risk of destabilizing the core framework or being caught up in release cycles. Given that most functionality is brought in using libraries rather than via changes to a monolithic framework, it makes the core numerics library more durable and adaptable to future shifts in the technology landscape.

The following sections provide a technical overview of the JAX ecosystem, its key features, the design decisions behind them, and how they combine to build a durable platform for modern ML workloads.

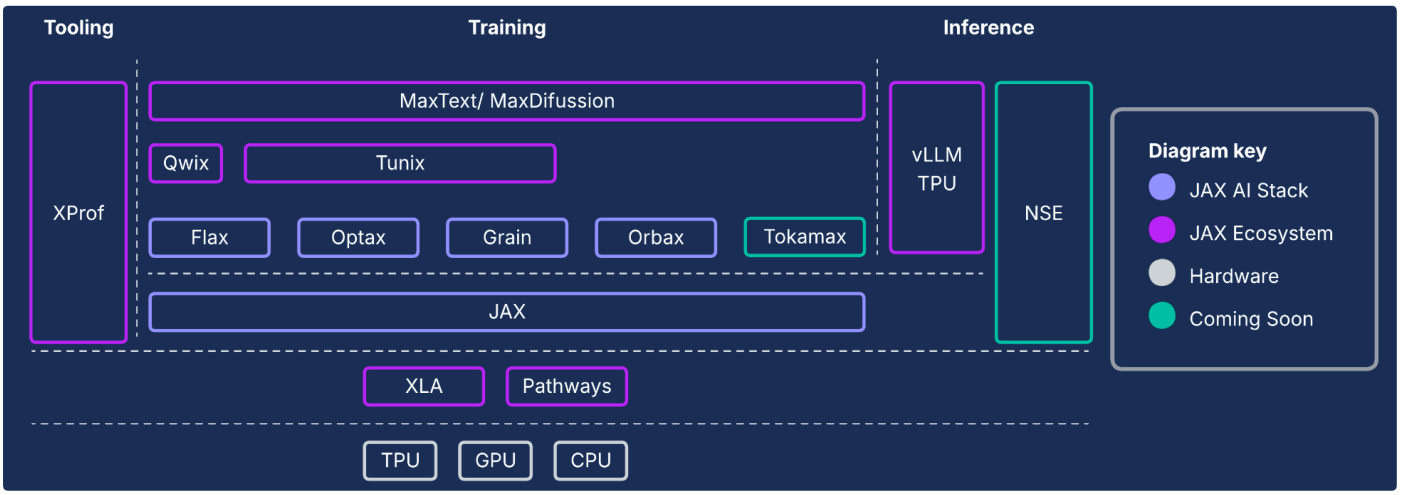
**Table 1: The JAX AI Stack and other Ecosystem Components**

| Component | Function / Description |
|---|---|
| **JAX AI stack core and components**[1] | |
| JAX | Accelerator-oriented array computation and program transformation (JIT, grad, vmap, pmap). |
| Flax | Flexible neural network authoring library for intuitive model creation and "surgery." |
| Optax | A library of composable gradient processing and optimization transformations. |
| Orbax | "Any-scale" distributed checkpointing library for hero-scale training resilience. |
| Grain | A scalable, deterministic, and checkpointable input data pipeline library. |
| **JAX Ecosystem - Infrastructure** | |
| XLA | Distributed runtime for orchestrating computation across tens of thousands of chips. |
| Pathways | A JAX extension for writing low-level, high-performance custom kernels in Python. |
| **JAX Ecosystem - Adv. Development** | |
| Pallas | A JAX extension for writing low-level, high-performance custom kernels in Python. |
| Tokamax | A curated library of state-of-the-art, |

[1] Included in the jax-ai-stack Python package

| | |
|---|---|
| | high-performance custom kernels (e.g., Attention). |
| Qwix | A comprehensive, non-intrusive library for quantization (PTQ, QAT, QLoRA). |
| **JAX Ecosystem - Application** | |
| MaxText / MaxDiffusion | Flagship, scalable reference frameworks for training foundation models (LLM, Diffusion). |
| Tunix | A framework for state-of-the-art post-training and alignment (RLHF, DPO). |
| vLLM | A high-performance LLM inference solution via native integration of the vLLM framework. |
| **Neptune Serving Engine** (coming soon) | JAX Serving Runtime: a high-performance, JAX-native C++ server for non-LLM models. |
| XProf | A deep, hardware-integrated profiler for system-wide performance analysis. |

**Figure 1: The JAX AI Stack and Ecosystem Components**




## The Architectural Imperative: Performance Beyond Frameworks

As model architectures converge—for example, on multimodal Mixture-of-Experts (MoE) Transformers—the pursuit of peak performance is leading to the emergence of "Megakernels." A Megakernel is effectively the entire forward pass (or a large portion) of one specific model, hand-coded using a lower-level API like the CUDA SDK on NVIDIA GPUs. This approach achieves maximum hardware utilization by aggressively overlapping compute, memory, and communication. Recent work from the research community has demonstrated that this approach can yield significant throughput gains, over 22% in some cases, for inference on GPUs. This trend is not limited to inference; evidence suggests that some large-scale training efforts have involved low-level hardware control to achieve substantial efficiency gains.

If this trend accelerates, all high-level frameworks as they exist today risk becoming less relevant, as low-level access to the hardware is what ultimately matters for performance on mature, stable architectures. This presents a challenge for all modern ML stacks: how to provide expert-level hardware control without sacrificing the productivity and flexibility of a high-level framework.

For TPUs to provide a clear path to this level of performance, the ecosystem must expose an API layer that is closer to the hardware, enabling the development of these highly specialized kernels. As this report will detail, the JAX stack is designed to solve this by offering a continuum of abstraction (See Figure 2), from the automated, high-level optimizations of the XLA compiler to the fine-grained, manual control of the Pallas kernel-authoring library.

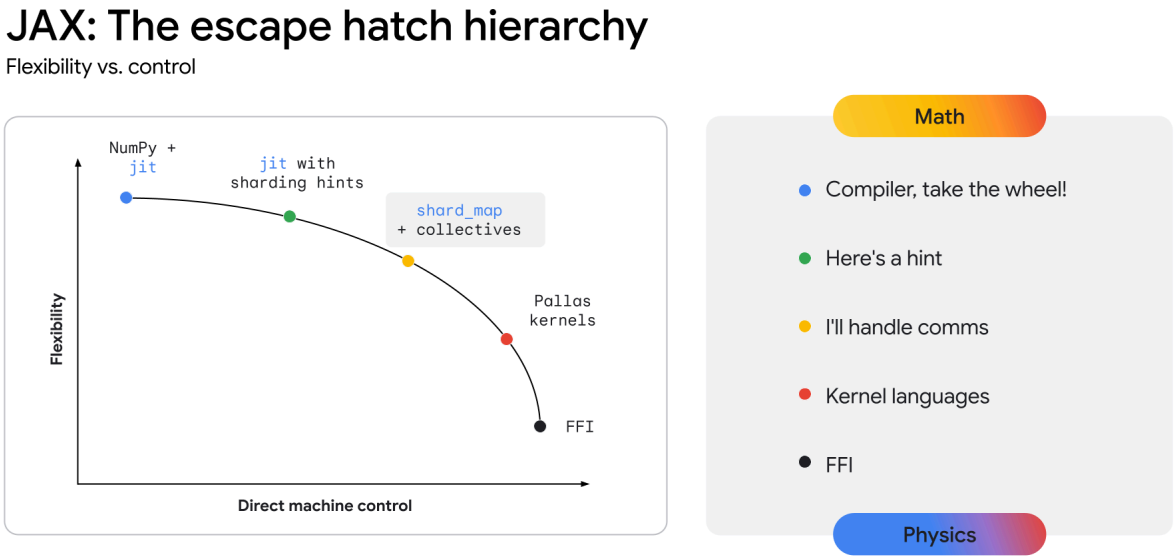



**Figure 2: The JAX continuum of abstraction**

## The Core JAX AI Stack

The core JAX AI Stack consists of five key libraries that provide the foundation for model development: JAX, Flax, Optax, Orbax and Grain.

## JAX: A Foundation for Composable, High-Performance Program Transformation

JAX is a Python library for accelerator-oriented array computation and program transformation, designed for high-performance numerical computing and large-scale Machine Learning. With its functional programming model and friendly, NumPy-like API, JAX provides a solid foundation for higher-level libraries.

With its compiler-first design, JAX inherently promotes scalability by leveraging XLA (see the XLA Section) for aggressive, whole-program analysis, optimization, and hardware targeting. The JAX emphasis on functional programming (i.e., pure functions) makes its core program transformations more tractable and, crucially, composable

These core transformations can be mixed and matched to achieve high performance and scaling of workloads across model size, cluster size, and hardware types:

- **jit**: Just-in-time compilation of Python functions into optimized, fused XLA executables.
- **grad**: Automatic differentiation, supporting forward- and reverse-mode, as well as higher-order derivatives.
- **vmap**: Automatic vectorization, enabling seamless batching and data parallelism without modifying function logic.
- **pmap / shard_map**: Automatic parallelization across multiple devices (e.g., TPU cores), forming the basis for distributed training.

The seamless integration with XLA's GSPMD (General-purpose SPMD) model allows JAX to automatically parallelize computations across large TPU pods with minimal code changes. In most cases, scaling simply requires high-level sharding annotations, a stark contrast to frameworks where scaling may require more manual management of device placement and communication collectives

## Flax: Flexible Neural Network Authoring and "Model Surgery"

Flax simplifies the creation, debugging, and analysis of neural networks in JAX by providing an intuitive, object-oriented approach to model building. While JAX's functional API is powerful, Flax offers a more familiar layer-based abstraction for developers accustomed to frameworks like PyTorch, without any performance penalty.

This design simplifies modern ML practices like "model surgery"—the process of modifying or combining trained model components. Techniques such as LoRA and quantization require easily manipulable model definitions, which Flax's NNX API provides through a simple, Pythonic interface. NNX encapsulates model state, reducing user cognitive load, and allows for programmatic traversal and modification of the model hierarchy.

Key Strengths:

- Intuitive Object-Oriented API: Simplifies model construction and enables advanced use cases like submodule replacement and partial initialization.
- Consistent with Core JAX: Flax provides lifted transformations that are fully compatible with JAX's functional paradigm, offering the full performance of JAX with enhanced developer friendliness.

## Optax: Composable Gradient Processing and Optimization Strategies

Optax is a gradient processing and optimization library for JAX. It is designed to empower model builders by providing building blocks that can be recombined in custom ways in order to train deep learning models amongst other applications. It builds on the capabilities of the core JAX library to provide a well tested high performance library of losses and optimizer functions and associated techniques that can be used to train ML models.

## Motivation

The calculation and minimization of losses is at the core of what enables the training of ML models. With its support for automatic differentiation the core JAX library provides the numeric capabilities to train models, but it does not provide standard implementations of popular optimizers (ex. `RMSProp`, `Adam`) or losses (`CrossEntropy`, `MSE` etc). While it is true that a user could implement these functions by themselves (and some advanced users will choose to do so), a bug in an optimizer implementation would introduce hard to diagnose model quality issues. Rather than having the user implement such critical pieces, Optax provides implementations of these algorithms that are tested for correctness and performance.

The field of optimization theory lies squarely in the realm of research, however its central role in training also makes it an indispensable part of training production ML models. A library that serves this role needs to be both flexible enough to accommodate rapid research iterations and also robust and performant enough to be dependable for production model training. It should also provide well tested implementations of state of the art algorithms which match the standard equations. The Optax library, through its modular composable architecture and emphasis on correct readable code is designed to achieve this.

## Design

Optax is designed to both enhance research velocity and the transition from research to production by providing readable, well-tested, and efficient implementations of core algorithms. Optax has uses beyond the context of deep learning, however in this context it can be viewed as a collection of well known loss functions, optimization algorithms and gradient transformations implemented in a pure functional fashion in line with the JAX philosophy. The collection of well known losses and optimizers enable users to get started with ease and confidence.

The modular approach taken by Optax easily allows users to chain multiple optimizers together followed by other common transformations like gradient clipping for example and wrap it using common techniques like MultiStep or Lookahead to achieve powerful optimization strategies all within a few lines of code. The flexible interface allows for easy research into new optimization algorithms and also enables powerful second order optimization techniques like shampoo or muon.

```python
# Optax implementation of a RMSProp optimizer with a custom learning rate schedule, gradient clipping and gradient accumulation.
optimizer = optax.chain(
  optax.clip_by_global_norm(GRADIENT_CLIP_VALUE),

optax.rmsprop(learning_rate=optax.cosine_decay_schedule(init_value=lr,decay_steps=decay)),
  optax.apply_every(k=ACCUMULATION_STEPS)
)

# The same thing, in PyTorch
optimizer = optim.RMSprop(model_params, lr=LEARNING_RATE)
scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=TOTAL_STEPS)
for i, (inputs, targets) in enumerate(data_loader):
    # ... Training loop body ...
    if (i + 1) % ACCUMULATION_STEPS == 0:
        torch.nn.utils.clip_grad_norm_(model.parameters(), GRADIENT_CLIP_VALUE)
        optimizer.step()
        scheduler.step()
        optimizer.zero_grad()
```

As it can be seen in the example above, setting up an optimizer with a custom learning rate, gradient clipping and gradient accumulation is a simple drop in replacement block of code, compared to PyTorch which forces the user to modify their training loop to directly manage the learning rate scheduler, gradient clipping and gradient accumulation.

Key Strengths

- **Robust Library:** Provides a comprehensive library of losses, optimizers, and algorithms with a focus on correctness and readability.
- **Modular Chainable Transformations:** As shown above, this flexible API allows users to craft powerful, complex optimization strategies declaratively, without modifying the training loop.
- **Functional and Scalable:** The pure functional implementations integrate seamlessly with JAX's parallelization mechanisms (e.g., pmap), enabling the same code to scale from a single host to large clusters.

## Orbax / TensorStore - Large scale distributed checkpointing

Orbax is a checkpointing library for JAX designed for any scale, from single-device to large-scale distributed training. It aims to unify fragmented checkpointing implementations and deliver critical performance features, such as asynchronous and multi-tier checkpointing, to a

wider audience. Orbax enables the resilience required for massive training jobs and provides a flexible format for publishing checkpoints.

Unlike generalized checkpoint/restore systems that snapshot the entire system state, ML checkpointing with Orbax selectively persists only the information essential for resuming training—model weights, optimizer state, and data loader state. This targeted approach minimizes accelerator downtime. Orbax achieves this by overlapping I/O operations with computation, a critical feature for large workloads. The time accelerators are halted is thus reduced to the duration of the device-to-host data transfer, which can be further overlapped with the next training step, making checkpointing nearly free from a performance perspective. At its core, Orbax uses TensorStore for efficient, parallel reading and writing of array data. The Orbax API abstracts this complexity, offering a user-friendly interface for handling PyTrees, which are the standard representation of models in JAX.

Key Strengths:

- Widespread Adoption: With millions of monthly downloads, Orbax serves as a common medium for sharing ML artifacts.
- Easy to Use: Orbax abstracts away the complexities of distributed checkpointing, including asynchronous saving, atomicity, and filesystem details.
- Flexible: While offering simple APIs for common use cases, Orbax allows for customization to handle specialized requirements.
- Performant and Scalable: Features like asynchronous checkpointing, an efficient storage format (OCDBT), and intelligent data loading strategies ensure that Orbax scales to training runs involving tens of thousands of nodes.

## Grain: Deterministic and Scalable Input Data Pipelines

Grain is a Python library for reading and processing data for training and evaluating JAX models. It is flexible, fast and deterministic and supports advanced features like checkpointing which are essential to successfully training large workloads. It supports popular data formats and storage backends and also provides a flexible API to extend support to user specific formats and backends that are not natively supported. While Grain is primarily designed to work with JAX, it is framework independent, does not require JAX to run and can be used with other frameworks as well.

### Motivation

Data pipelines form a critical part of the training infrastructure - they need to be flexible so that common transformations can be expressed efficiently, and performant enough that they are able to keep the accelerators busy at all times. They also need to be able to accommodate multiple storage formats and backends. Due to their higher step times, training large models

at scale pose unique additional requirements on the data pipeline beyond those that are required by regular training workloads, primarily focused around determinism and reproducibility[2]. The Grain library is designed with a flexible enough architecture to address all these needs.

### Design

At the highest level, there are two ways to structure an input pipeline, as a separate cluster of data workers or by co-locating the data workers on the hosts that drive the accelerators. Grain chooses the latter for a variety of reasons.

Accelerators are combined with powerful hosts that typically sit idle during training steps, which makes it a natural choice to run the input data pipeline. There are however additional advantages to doing so - it simplifies the user's view of data sharding by providing a consistent view of sharding across input and compute. It could be argued that putting the data worker on the accelerator host risks saturating the host CPU, however this does not preclude offloading compute intensive transformations to another cluster via RPCs[3].

On the API front, with a pure python implementation that supports multiple processes and a flexible API, Grain enables users to implement arbitrarily complex data transformations by composing together pipeline stages based on well understood transformation paradigms.

Out of the box, Grain supports efficient random access data formats like `ArrayRecord` and `Bagz` alongside other popular data formats such as Parquet and `TFDS`. Grain includes support for reading from local file systems as well as reading from GCS by default. Along with supporting popular storage formats and backends, a clean abstraction to the storage layer allows users to easily add support for or wrap their existing data sources to be compatible with the Grain library.

### Key Strengths

- **Deterministic data feeding:** Colocating the data worker with the accelerator and coupling it with a stable global shuffle and checkpointable iterators allows the model state and data pipeline state to be checkpointed together in a consistent snapshot using Orbax, enhancing the determinism of the training process.
- **Flexible APIs to enable powerful data transformations:** A flexible pure Python transformations API allows for extensive data transformations within the input processing pipeline.

---

[2] In the Section 5.1 of the Palm paper, the authors note that they observed very large loss spikes despite having gradient clipping enabled and the solution was to remove the offending data batches and restart training from a checkpoint before the loss spike. This is only possible with a fully deterministic and reproducible training setup.

[3] This is indeed how multimodal data pipelines would need to operate - image and audio tokenizers for example are models themselves which run in their own clusters on their own accelerators and the input pipelines would make RPCs out to convert data examples into streams of tokens.

- **Extensible support for multiple formats and backends:** An extensible data sources API supports popular storage formats and backends and allows users to easily add support for new formats and backends.
- **Powerful debugging interface:** Data pipeline visualization tools and a debug mode allow users to introspect, debug and optimize the performance of their data pipelines.

## The Extended JAX Ecosystem

Beyond the core stack, a rich ecosystem of specialized libraries provides the infrastructure, advanced tools, and application-layer solutions needed for end-to-end ML development.

## Foundational Infrastructure: Compilers and Runtimes

### XLA: The Hardware-Agnostic, Compiler-Centric Engine

#### Motivation

XLA or Accelerated Linear Algebra is our domain specific compiler, which is well integrated into JAX and supports TPU, CPU and GPU hardware devices.  From inception, XLA has been designed to be  a hardware agnostic code generator targeting TPUs, GPUs, and CPUs.

Our compiler-first design is a fundamental architectural choice that creates a durable advantage in a rapidly evolving research landscape. In contrast, the prevailing kernel-centric approach in other ecosystems relies on hand-optimized libraries for performance. While this is highly effective for stable, well-established model architectures, it creates a bottleneck for innovation. When new research introduces novel architectures, the ecosystem must wait for new kernels to be written and optimized. Our compiler-centric design, however, can often generalize to new patterns, providing a high-performance path for cutting-edge research from day one.

#### Design

XLA works by Just-In-Time (JIT) compiling the computation graphs that JAX generates during its tracing process (e.g., when a function is decorated with `@jax.jit`).

This compilation follows a multi-stage pipeline:

> JAX Computation Graph → High-Level Optimizer (HLO) → Low-Level Optimizer (LLO) → Hardware Code

- **From JAX Graph to HLO**: The captured JAX computation graph is converted into XLA's HLO representation. At this high level, powerful, hardware-agnostic

optimizations like operator fusion and efficient memory management are applied. The **StableHLO** dialect serves as a durable, versioned interface for this stage.
- **From HLO to LLO:** After high-level optimizations, hardware-specific backends take over, lowering the HLO representation into a machine-oriented LLO.
- **From LLO to Hardware Code:** The LLO is finally compiled into highly-efficient machine code. For TPUs, this code is bundled as **Very Long Instruction Word (VLIW)** packets that are sent directly to the hardware.

For scaling, XLA's design is built around parallelism. It employs algorithms to maximally utilize the matrix multiplication units (MXUs) on a chip. Between chips, XLA uses **SPMD (Single Program Multiple Data)**, a compiler-based parallelization that uses a single program across all devices. This powerful model is exposed through JAX APIs, allowing users to manage data, model, or pipeline parallelism with high-level sharding annotations.

For more complex parallelism patterns, **Multiple Program Multiple Data (MPMD)** is also possible, and libraries like `PartIR:MPMD` allow JAX users to provide MPMD annotations as well.

#### Key strengths

- **Compilation**: just in time compilation of the computation graph allows for optimizations to memory layout, buffer allocation, and memory management. Alternatives such as kernel based methodologies put that burden on the user. In most cases, XLA can achieve excellent performance without compromising developer velocity.
- **Parallelism:** XLA implements several forms of parallelism with SPMD, and this is exposed at the JAX level. This allows for users to express sharding strategies easily, allowing experimentation and scalability of models across thousands of chips.

### Pathways: A Unified Runtime for Massive-Scale Distributed Computation

Pathways offers abstractions for distributed training and inference with built in fault tolerance and recovery, allowing ML researchers to code as if they are using a single, powerful machine.

#### Motivation

To be able to train and deploy large models, hundreds to thousands of chips are necessary. These chips are spread across numerous racks and host machines. A training job is a large-scale synchronous program that requires all of these chips, and their respective hosts to be working in tandem on XLA computations that have been parallelized (sharded). In the case of large language models, which may need more than tens of thousands of chips, this service must be capable of spanning multiple pods across a data center fabric in addition to using ICI and OCI fabrics within a pod.

#### Design

ML Pathways is the system we use for coordinating distributed computations across hosts and TPU chips. It is designed for scalability and efficiency across hundreds of thousands of accelerators. For large-scale training, it provides a single Python client for multi-slice/multi-pod jobs, Megascale XLA integration, Compilation Service, and Remote Python. It also supports cross-slice parallelism and preemption tolerance, enabling automatic recovery from resource preemptions.

Pathways incorporates optimized cross host collectives which enable XLA computation graphs to further extend beyond a single TPU pod.  It expands XLA's support for data, model, and pipeline parallelism to work across TPU slice boundaries using DCN by means of integrating a distributed runtime that manages DCN communication with XLA communication primitives.

#### Key strengths

- The single-controller architecture, integrated with JAX, is a key abstraction. It allows researchers to explore various sharding and parallelism strategies for training and deployment while scaling to tens of thousands of chips with ease.
- Scaling to tens of thousands of chips with ease, allowing exploration of various sharding and parallelism strategies during model research, training and deployment.

## Advanced Development: Performance, Data, and Efficiency

### Pallas: Writing High-Performance Custom Kernels in JAX

While JAX is compiler first, there are situations where the user would like to exercise fine grained control over the hardware to achieve maximum performance. Pallas is an extension to JAX that enables writing custom kernels for GPU and TPU. It aims to provide precise control over the generated code, combined with the high-level ergonomics of JAX tracing and the `jax.numpy API`.

Pallas exposes a grid-based parallelism model where a user-defined kernel function is launched across a multi-dimensional grid of parallel work-groups. It enables explicit management of the memory hierarchy by allowing the user to define how tensors are tiled and transferred between slower, larger memory (e.g., HBM) and faster, smaller on-chip memory (e.g., VMEM on TPU, Shared Memory on GPU), using index maps to associate grid locations with specific data blocks. Pallas can lower the same kernel definition to execute efficiently on both Google's TPUs and various GPUs by compiling kernels into an intermediate representation suitable for the target architecture – Mosaic for TPUs, or utilizing technologies like Triton for the GPU path. With Pallas, users can write high performance kernels that specialize blocks like attention to achieve the best model performance on the target hardware without needing to rely on vendor specific toolkits.

## Tokamax: A Curated Library of State-of-the-Art Kernels

If Pallas is the *tool* for authoring kernels, Tokamax is a *library* of state-of-the-art custom accelerator kernels supporting both TPUs and GPUs, built on top of JAX and Pallas enabling users to push their hardware to the maximum. It also provides tooling for users to build and autotune their own custom kernels.

#### Motivation

JAX, with its roots in XLA, is a compiler-first framework, however a narrow set of cases exists where the user needs to take direct control of the hardware to achieve maximum performance [4]. Custom kernels are critical to squeezing out every last ounce of performance from expensive ML accelerator resources such as TPUs and GPUs. While they are widely employed to enable performant execution of key operators such as Attention, implementing them requires a deep understanding of both the model and the target hardware (micro)architecture. Tokamax provides one authoritative source of curated, well-tested, high-performance kernels, in conjunction with robust shared infrastructure for their development, maintenance, and lifecycle management. Such a library can also act as a reference implementation for users to build on and customize as necessary. This allows users to focus on their modeling efforts without needing to worry about infrastructure.

#### Design

For any given kernel, Tokamax provides a common API that may be backed by multiple implementations. For example, TPU kernels may be implemented either by standard XLA lowering, or explicitly via Pallas/Mosaic-TPU. GPU kernels may be implemented by standard XLA lowering, via Mosaic-GPU, or Triton. By default, it picks the best-known implementation for a given configuration, determined by cached results from periodic autotuning and benchmarking runs, though users may choose specific implementations if desired. New implementations may be added over time to better exploit specific features in new hardware generations for even better performance.

A key component of the library, beyond the kernels themselves, is the supporting infrastructure that will help power users choosing to write their own custom kernels. For example, the autotuning infrastructure lets the user define a set of configurable parameters (e.g., tile sizes) that Tokamax can perform an exhaustive sweep on, to determine and cache the best possible tuned settings. Nightly regressions protect users from unexpected performance and numerics issues caused by changes to underlying compiler infrastructure or other dependencies.

[4] This is a well established paradigm and has precedent in the CPU world, where compiled code forms the bulk of the program with developers dropping down to intrinsics or inline assembly to optimize performance critical sections.

#### Key Strengths

- **Seamless developer experience:** A unified, curated, library will provide known-good high-performance implementations of key kernels, with clear expressions of supported hardware generations and expected performance, both programmatically and in documentation. This minimizes fragmentation and churn.
- **Flexibility and lifecycle management:** Users may choose different implementations as desired, even changing them over time if appropriate. For example, if the XLA compiler enhances support for certain operations obviating the need for custom kernels, there is a simple path to deprecation and migration.
- **Extensibility:** Users may implement their own kernels, while leveraging well-supported shared infrastructure, allowing them to focus on their value added capabilities and optimizations. Clearly authored standard implementations serve as a starting point for users to learn from and extend.

### Qwix: Non-Intrusive, Comprehensive Quantization

Qwix is a comprehensive quantization library for the JAX ecosystem, supporting both LLMs and other model types across all stages, including training (QAT, QT, QLoRA) and inference (PTQ), targeting both XLA and on-device runtimes.

#### Motivation

Existing quantization libraries, particularly in the PyTorch ecosystem, often serve limited purposes (e.g., only PTQ or only QLoRA). This fragmented landscape forces users to switch tools, impeding consistent code usage and precise numerical matching between training and inference. Furthermore, many solutions require substantial model modifications, tightly coupling the model logic to the quantization logic.

#### Design

Qwix's design philosophy emphasizes a comprehensive solution and, critically, **non-intrusive model integration**. It is architected with a hierarchical, extensible design built on reusable functional APIs.

This non-intrusive integration is achieved through a meticulously designed **interception mechanism** that redirects JAX functions to their quantized counterparts. This allows users to integrate their models without any modifications, completely decoupling quantization code from model definitions.

The following example demonstrates applying `w4a4` (4-bit weight, 4-bit activation) quantization to an LLM's MLP layers and `w8` (8-bit weight) quantization to the embedder. To change the quantization recipe, only the rules list needs to be updated.

```python
fp_model = ModelWithoutQuantization(...)
rules = [
    qwix.QuantizationRule(
        module_path=r'embedder',
        weight_qtype='int8',
    ),
    qwix.QuantizationRule(
        module_path=r'layers_\d+/mlp',
        weight_qtype='int4',
        act_qtype='int4',
        tile_size=128,
        weight_calibration_method='rms,7',
    ),
]
quantized_model = qwix.quantize_model(fp_model, qwix.PtqProvider(rules))
```

Key Strengths

- **Comprehensive Solution:** Qwix is broadly applicable across numerous quantization scenarios, ensuring consistent code usage between training and inference.
- **Non-Intrusive Model Integration:** As the example shows, users can integrate models with a single line of code, without modification. This allows developers to easily sweep hyperparameters over many quantization schemes to find the best quality/performance tradeoff.
- **Federated with Other Libraries:** Qwix seamlessly integrates with the JAX AI stack. For example, Tokamax automatically adapts to use quantized versions of kernels, without additional user code, when the model is quantized with Qwix.
- **Research Friendly:** Qwix's foundational APIs and extensible architecture empower researchers to explore new algorithms and facilitate straightforward comparisons with integrated benchmark and evaluation tools.

## The Application Layer: Training and Alignment

### Foundation Model Training: MaxText and MaxDiffusion

MaxText and MaxDiffusion are Google's flagship LLM and Diffusion model training frameworks, respectively. With a large selection of highly optimized implementations of popular open-weights models, these repositories serve a dual purpose: they function as both a ready-to-go model training codebase and as a reference that foundation model builders

can use to build upon.

#### Motivation

There is rapid growth of interest across the industry in training GenAI models. The popularity of open models has accelerated this trend, providing users with proven architectures. To train and adapt these models, users require high performance, efficiency, scalability to extreme numbers of chips, and clear, understandable code. They need a framework that can adapt to new techniques and target both TPUs and GPUs. MaxText and MaxDiffusion are comprehensive solutions designed to fulfill these needs.

#### Design

MaxText and MaxDiffusion are foundation model codebases designed with readability and performance in mind. They are structured with well-tested, reusable components: model definitions that leverage custom kernels (like Tokamax) for maximum performance, a training harness for orchestration and monitoring, and a powerful config system that allows users to control details like sharding and quantization (via Qwix) through an intuitive interface. Advanced reliability features like multi-tier checkpointing are incorporated to ensure sustained goodput.

They leverage the best-in-class JAX libraries—Qwix, Tunix, Orbax, and Optax—to deliver core capabilities. This allows them to provide robust, scalable infrastructure, reducing development overhead and allowing users to focus on the modeling task. For inference, the model code is shared to enable efficient and scalable serving.

#### Key Strengths

- **Performant by Design:** With training infrastructure set up for high "goodput" (useful throughput) and model implementations optimized for high MFU (Model Flops Utilization), MaxText and MaxDiffusion deliver high performance at scale out of the box
- **Built for Scale:** Leveraging the power of the JAX AI stack (especially Pathways), these frameworks allow users to scale seamlessly from tens of chips to tens of thousands of chips
- **Solid Base for Foundation Model Builders:** The high-quality, readable implementations serve as a solid starting point for builders to either use as an end-to-end solution or as a reference implementation for their own customizations

### Post-Training and Alignment: The Tunix Framework

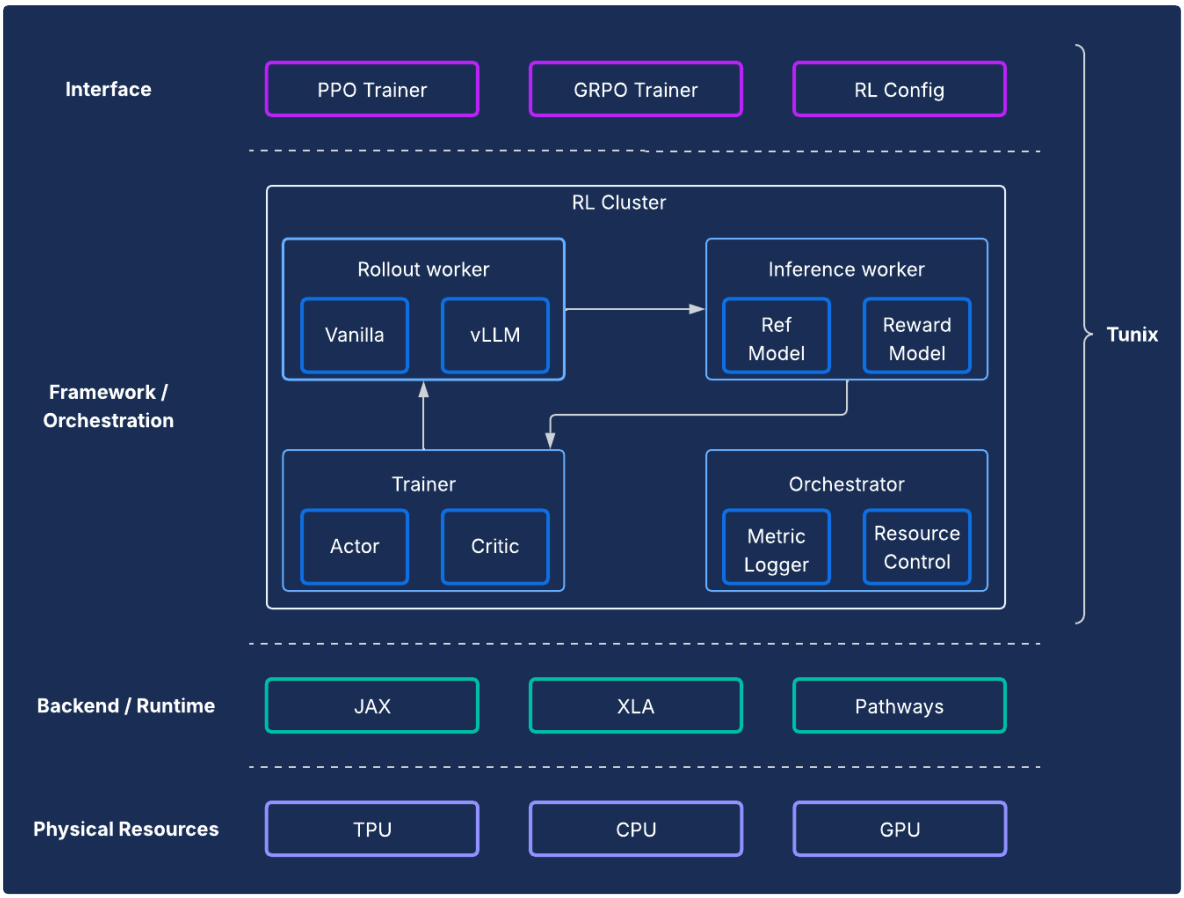
Tunix offers state-of-the-art open-source reinforcement learning (RL) algorithms, along with a robust framework and infrastructure, providing a streamlined path for users to experiment with LLM post-training techniques (including Supervised Fine-Tuning (SFT) and alignment)

using JAX and TPUs.

**Motivation**

Post-training is the critical step in unlocking the true power of LLMs. The Reinforcement Learning stage is particularly crucial for developing alignment and reasoning capabilities. While fast-moving open-source development in this area has been prolific, it has been almost exclusively based on PyTorch and GPUs, leaving a fundamental gap for JAX and TPU solutions. Tunix (Tune-in-JAX) is a high-performance, JAX-native library designed to fill this gap.

**Design**




From a framework perspective, Tunix enables a state-of-the-art setup that **clearly separates RL algorithms from the infrastructure**. It offers a lightweight, client-like API that hides the

complexity of the RL infrastructure, allowing users to develop new algorithms easily. Tunix provides out-of-the-box solutions for popular algorithms, including PPO, DPO, and others.

On the infrastructure side, Tunix has native integration with Pathways, enabling a single-controller architecture that makes multi-node RL training easily accessible. On the trainer side, Tunix natively supports parameter-efficient training (e.g., LoRA) and leverages JAX sharding and XLA (GSPMD) to generate a performant compute graph. It supports popular open-source models like Gemma and Llama out of the box.

#### Key Strengths

- **Simplicity:** It provides a high-level, client-like API that abstracts away the complexities of the underlying distributed infrastructure.
- **Developer Efficiency:** Tunix accelerates the R&D lifecycle with out-of-the-box algorithms and pre-built "recipes," enabling users to get a working model and iterate quickly.
- **Performance and Scalability:** Tunix enables a highly efficient and horizontally scalable training infrastructure by leveraging Pathways as a single controller on the backend.

## The Application Layer: Production and Inference

A historical challenge for JAX adoption has been the path from research to production. The JAX AI stack now provides a mature, two-pronged production story that offers both ecosystem compatibility and native JAX performance.

### High-Performance LLM Inference: The vLLM Solution

vLLM-TPU is Google's high-performance inference stack designed to run PyTorch and JAX native Large Language Models (LLMs) efficiently on Cloud TPUs. It achieves this by natively integrating the popular open-source vLLM framework with Google's JAX and TPU ecosystem.

#### Motivation

The industry is rapidly evolving, with growing demand for seamless, high-performing, and easy-to-use inference solutions. Users often face significant challenges from complex and inconsistent tooling, subpar performance, and limited model compatibility. The vLLM stack addresses these issues by providing a unified, performant, and intuitive platform.

#### Design

This solution pragmatically extends the vLLM framework, rather than reinventing it. vLLM-TPU is a highly optimized open-source LLM serving engine known for its high throughput, achieved via key features like `PagedAttention` (which manages KV caches like virtual memory to minimize fragmentation) and `Continuous Batching` (which dynamically adds requests to the batch to improve utilization).

vLLM-TPU builds on this foundation and develops core components for request handling, scheduling, and memory management. It introduces a **JAX-based backend** that acts as a bridge, translating vLLM's computational graph and memory operations into TPU-executable code. This backend handles device interactions, JAX model execution, and the specifics of managing the KV cache on TPU hardware. It incorporates TPU-specific optimizations, such as efficient attention mechanisms (e.g., leveraging JAX Pallas kernels for Ragged Paged Attention) and quantization, all tailored for the TPU architecture.

#### Key Strengths

- **Zero Onboarding/Offboarding Cost for Users:** Users can adopt this solution without significant friction. From a user-experience perspective, processing inference requests is identical to on GPUs. The CLI to start the server, accept prompts, and return outputs are all shared.
- **Fully Embrace the Ecosystem:** This approach utilizes and contributes to the vLLM interface and user experience, ensuring compatibility and ease of use.
- **Fungibility between TPUs and GPUs:** The solution works efficiently on both TPUs and GPUs, allowing users flexibility.
- **Cost Efficient (Best Perf/$):** Optimizes performance to provide the best

performance-to-cost ratio for popular models.

## JAX-Native Serving: Orbax Serialization and Neptune Serving Engine

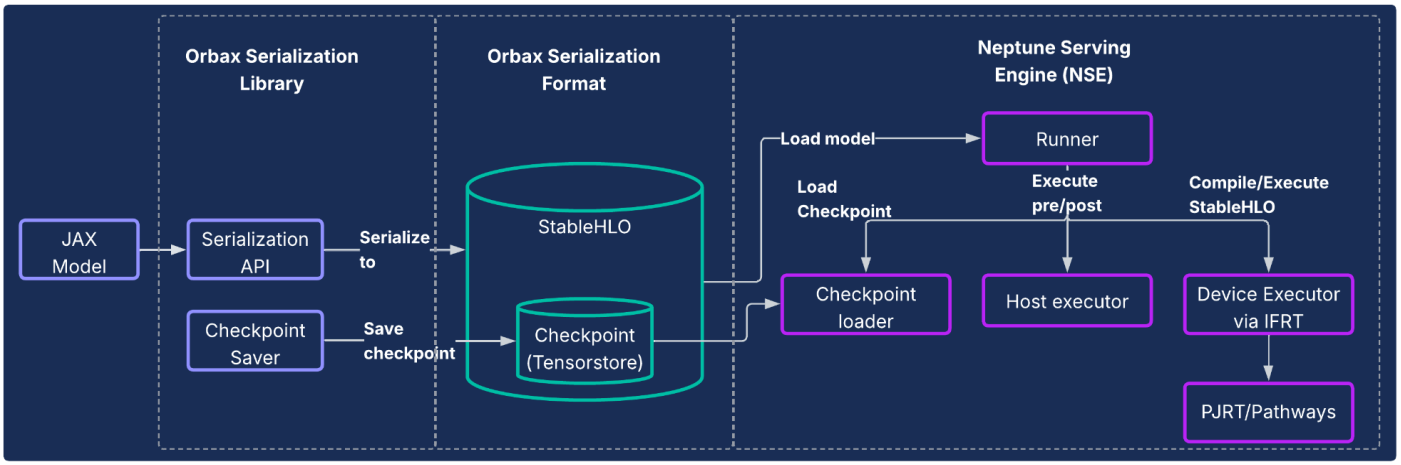
For models other than LLMs, or for users desiring a fully JAX-native pipeline, the Orbax serialization library and Neptune serving engine (NSE) system provide an end-to-end, high-performance serving solution.

#### Motivation

Historically, JAX models often relied on a circuitous path to production, such as being wrapped in TensorFlow graphs and deployed using TensorFlow serving. This approach introduced significant limitations and inefficiencies, forcing developers to engage with a separate ecosystem and slowing down iteration. A dedicated JAX-native serving system is crucial for sustainability, reduced complexity, and optimized performance.

#### Design

This solution consists of two core components, as illustrated in the diagram below.




1. **Orbax Serialization Library:** Provides user-friendly APIs for serializing JAX models into a new, robust Orbax serialization format. This format is optimized for production deployment. Its core includes: (a) directly representing JAX model computations using **StableHLO**, allowing the computation graph to be represented natively, and (b) leveraging **TensorStore** for storing weights, enabling fast checkpoint loading for serving.
2. **Neptune Serving Engine (NSE):** This is the accompanying high-performance, flexible serving engine (typically deployed as a C++ binary) designed to natively run JAX models in the Orbax format. NSE offers production-essential capabilities, such as fast model loading, high-throughput concurrent serving with built-in batching, support for multiple model versions, and both single- and multi-host serving (leveraging PJRT and

Pathways). Use the Neptune Serving Engine for:

- Non-LLM models: It is a general-purpose solution ideal for workloads like recommender systems, diffusion models, and other traditional AI models.
- Small LLMs and "one-shot" serving: It is designed for non-autoregressive models or smaller models that are served in a "unary" fashion, where the entire output is generated in a single pass without the need for complex state management like a KV cache.

In short, Neptune Serving Engine fills the gap for serving the wide variety of models that are not large, autoregressive language models, providing a high-performance TPU-native solution for the broader ML ecosystem.

#### Key Strengths

- **JAX Native Serving:** The solution is built natively for JAX, eliminating inter-framework overhead in model serialization and serving. This ensures lightning-fast model loading and optimized execution across CPUs, GPUs, and TPUs.
- **Effortless Production Deployment:** Serialized models provide a **hermetic deployment path** that is unaffected by drift in Python dependencies and enables runtime model integrity checks. This provides a seamless, intuitive path for JAX model productionization.
- **Enhanced Developer Experience:** By eliminating the need for cumbersome framework wrapping, this solution significantly reduces dependencies and system complexity, speeding up iteration for JAX developers.

## System-Wide Analysis and Profiling

### XProf: Deep, Hardware-Integrated Performance Profiling

XProf is a profiling and performance analysis tool that provides in-depth visibility into various aspects of ML workload execution, enabling users to debug and optimize performance. It is deeply integrated into both the JAX and TPU ecosystems.

#### Motivation

On one hand, ML workloads are growing ever more complicated. On the other, there is an explosion of specialized hardware capabilities targeting these workloads. Matching the two effectively to ensure peak performance and efficiency is critical, given the enormous costs of ML infrastructure. This requires deep visibility into both the workload and the hardware, presented in a way that is easily consumable. XProf excels at this.

#### Design

XProf consists of two primary components: collection and analysis.

1. **Collection:** XProf captures information from various sources: annotations in the user's JAX code, cost models for operations within the XLA compiler, and **purpose-built hardware profiling features within the TPU**. This collection can be triggered programmatically or on-demand, generating a comprehensive event artifact.
2. **Analysis:** XProf post-processes the collected data and creates a suite of powerful visualizations, accessed via a browser.

#### Key Strengths

The true power of XProf comes from its deep integration with the full stack, providing a breadth and depth of analysis that is a tangible benefit of the co-designed JAX/TPU ecosystem.

- **Co-designed with the TPU:** XProf exploits hardware features specifically designed for seamless profile collection, enabling a collection overhead of **less than 1%**. This allows profiling to be a lightweight, iterative part of development.
- **Breadth and Depth of Analysis:** XProf yields deep analysis across multiple axes. Its tools include:
    - **Trace Viewer:** An op-by-op timeline view of execution on different hardware units (e.g., TensorCore).
    - **HLO Op Profile:** Breaks down the total time spent into different categories of operations.
    - **Memory Viewer:** Details memory allocations by different ops during the profiled window.
    - **Roofline Analysis:** Helps identify whether specific ops are compute- or memory-bound and how far they are from the hardware's peak capabilities.
    - **Graph Viewer:** Provides a view into the full HLO graph executed by the hardware.

# A Comparative Perspective: The JAX/TPU Stack as a Compelling Choice

The modern Machine Learning landscape offers many excellent, mature toolchains. The JAX AI Stack, however, presents a unique and compelling set of advantages for developers focused on large-scale, high-performance ML, stemming directly from its modular design and deep hardware co-design.

While many frameworks offer a wide array of features, the JAX AI Stack provides specific, powerful differentiators in key areas of the development lifecycle:

- **A Simpler, More Powerful Developer Experience:** The "chainable gradient transformation paradigm" of **Optax** allows for more powerful and flexible optimization strategies that are declared once, rather than imperatively managed in the training loop.[1] At the system level, the "simpler single controller interface" of **Pathways** abstracts away the complexity of multi-pod, multi-slice training, a significant simplification for researchers.

- **Engineered for "Hero-Scale" Resilience:** The JAX stack is designed for extreme-scale training. **Orbax** provides "hero-scale training resilience" features like emergency and multi-tier checkpointing. This is complemented by **Grain**, which offers "full support for reproducibility with deterministic global shuffles and checkpointable data loaders". The ability to atomically checkpoint the data pipeline state (Grain) with the model state (Orbax) is a critical capability for guaranteeing reproducibility in long-running jobs.
- **A Complete, End-to-End Ecosystem:** The stack provides a cohesive, end-to-end solution. Developers can use **MaxText** as a SOTA reference for training, **Tunix** for alignment, and follow a clear, dual-path to production with **vLLM-TPU** (for vLLM compatibility) and **NSE** (for native JAX performance).

While many stacks are vastly similar from a high-level software standpoint, the deciding factor often comes down to **Performance/TCO**, which is where the co-design of JAX and TPUs provides a distinct advantage. This Performance/TCO benefit is a direct result of the "vertical integration across software and TPU hardware". The ability of the **XLA** compiler to fuse operations specifically for the TPU architecture, or for the **XProf** profiler to leverage hardware hooks for <1% overhead profiling, are tangible benefits of this deep integration.

For organizations adopting this stack, the "full featured nature" of the JAX AI Stack minimizes the cost of migration. For customers employing popular open model architectures, a shift from other frameworks to MaxText is often a matter of setting up config files. Furthermore, the stack's ability to ingest popular checkpoint formats like safetensors allows existing checkpoints to be migrated over without needing costly re-training.

The table below provides a mapping of the components provided by the JAX AI stack and their equivalents in other frameworks or libraries.

| **Function** | **JAX** | **Alternatives/equivalents in other frameworks**[5] |
|---|---|---|
| Compiler / Runtime | XLA | Inductor, Eager |
| Multipod Training | Pathways | Torch Lightning Strategies, Ray Train, Monarch (new). |
| Core Framework | JAX | PyTorch |
| Model authoring | Flax,<br>Max* models | torch.nn.*,<br>NVidia TransformerEngine, HuggingFace Transformers |

[5] Some of the equivalents here are not true 1:1 comparisons because other frameworks draw API boundaries differently compared to JAX. The list of equivalents is not exhaustive and there are new libraries appearing frequently.

| | | |
|---|---|---|
| Optimizers & Losses | Optax | torch.optim.*, torch.nn.*Loss |
| Data Loaders | Grain | Ray Data, HuggingFace dataloaders |
| Checkpointing | Orbax | PyTorch distributed checkpointing, NeMo checkpointing |
| Quantization | Qwix | TorchAO, bitsandbytes |
| Kernel authoring & well known implementations | Pallas / Tokamax | Triton/Helion, Liger-kernel, TransformerEngine |
| Post training / tuning | Tunix | VERL, NeMoRL |
| Profiling | XProf | PyTorch profiler, NSight systems, NSight Compute |
| Foundation model Training | MaxText, MaxDiffusion | NeMo-Megatron, DeepSpeed, TorchTitan |
| LLM inference | vLLM | vLLM, SGLang |
| Non-LLM Inference | NSE | Triton Inference Server, RayServe |

# Conclusion: A Durable, Production-Ready Platform for the Future of AI

The data provided in the table above draws to a rather simple conclusion - these stacks have their own strengths and weaknesses in a small number of areas but overall are vastly similar from the software standpoint. Both stacks provide out of the box turnkey solutions for pre-training, post-training adaptation and deployment of foundational models.

The JAX AI stack offers a compelling and robust solution for training and deploying ML models at any scale. It leverages deep vertical integration across software and TPU hardware to deliver class-leading performance and total cost of ownership.

By building on battle-tested internal systems, the stack has evolved to provide inherent reliability and scalability, enabling users to confidently develop and deploy even the largest models. Its modular and composable design, rooted in the JAX ecosystem philosophy, grants users unparalleled freedom and control, allowing them to tailor the stack to their specific needs without the constraints of a monolithic framework.

With XLA and Pathways providing a scalable and fault-tolerant base, JAX providing a performant and expressive numerics library, powerful core development libraries like Flax, Optax, Grain, and Orbax, advanced performance tools like Pallas, Tokamax, and Qwix, and a

robust application and production layer in MaxText, vLLM, and NSE, the JAX AI stack provides a durable foundation for users to build on and rapidly bring state-of-the-art research to production.