

Hybrid Rust+C++/CUDA ML Systems: Open-Source Examples and Architecture Patterns

Open-Source Examples of Rust Orchestration + C++/CUDA Compute

NVIDIA Dynamo (Distributed LLM Serving)

NVIDIA's **Dynamo** is a recent open-source framework that illustrates a Rust+Python hybrid approach for large-model inference. Rust is used for high-performance orchestration (frontend servers, scheduling, request routing), while Python (with C++/CUDA backends) is used for model execution ¹ ². Dynamo was designed to coordinate multi-GPU, multi-node workloads – originally focusing on inference, but its architecture is general. It supports **multiple backend engines** (e.g. vLLM, TensorRT-LLM, etc.) behind a common interface ³ ¹. The system consists of Rust-based components (like an OpenAI-compatible HTTP server and router) that communicate with **worker processes** running the actual model code (e.g. Python drivers using CUDA libraries) ².

Key features aligning with the question: - *Clear Orchestration vs Compute Separation*: Rust handles cluster coordination (etcd for state, NATS for messaging) and networking, whereas heavy ML computations (transformer forward passes, etc.) run in optimized C++/CUDA libraries via Python bindings ⁴ ². This separation is explicit – for example, Dynamo's Rust **frontend** accepts requests and routes them to **workers** running specialized GPU-bound engines ². - *Distributed and Multi-environment*: Dynamo was built to **scale across nodes/GPUs**, tackling the “orchestration gap” when a model is sharded over many GPUs ⁵. It uses etcd/NATS for cluster coordination, so it can run locally (single node) or span a cluster/cloud environment. In local mode it even supports a fallback to file-based coordination for simplicity ⁶ ⁷. - *Interfaces*: Dynamo offers a CLI-driven deployment (e.g. `python -m dynamo.frontend ...` to launch components) and a **RESTful web API** (OpenAI-compatible) for inference requests ². It also has a browser-based GUI for monitoring (noted in docs) and Kubernetes integration for cloud deployment ⁸. - *Documentation & Architecture*: NVIDIA provides design proposals and an architecture diagram. The **architecture** follows a **distributed microservices** pattern: a Rust service for the front-end and routing, and separate backend services for each model engine. This matches the hybrid approach: Rust orchestrates the workflow between services, while each service uses optimized libraries (like C++ CUDA kernels in TensorRT or vLLM's C++ code) for the heavy lifting. The design is clearly documented, emphasizing Rust “for performance” and Python “for extensibility” ⁹, which demonstrates the rationale for using Rust as the control plane and leveraging existing ML libraries for compute.

Assessment: Although Dynamo targets inference rather than training, its architecture is relevant – one could adapt it for training by having Rust coordinate distributed training jobs (scheduling all-reduce operations, syncing model shards, etc.) while C++/CUDA libraries (e.g. NCCL, cuBLAS, or PyTorch C++ API) do the math. Dynamo closely aligns with the use case's needs: multi-backend support, distributed operation, and a clear separation of concerns (Rust orchestrator vs. GPU compute engines) ³ ¹.

Burn (Rust ML Framework with Multi-Backend Support)

Burn is an open-source deep learning framework written in Rust that cleanly separates high-level orchestration from low-level compute via a *backend trait* system. The framework's core is Rust (for model definition, training loops, etc.), but it can swap in different compute backends – including ones based on CUDA/C++ libraries ¹⁰ ¹¹. For example, Burn supports backends like NVIDIA CUDA, AMD ROCm, WebGPU, and even **LibTorch** (PyTorch's C++ engine) ¹⁰. This means you can train a model in Burn with one line of Rust code, and underneath it may invoke CUDA kernels (via the chosen backend) or call into C++ libraries like *libtorch*. Crucially, Burn achieves this through a **clear module separation**: the high-level ops go through a `Backend` trait, and each backend (implemented in its own Rust crate) handles the compute details ¹². For instance, a `burn-tch` crate enables using PyTorch C++ for tensor ops, whereas `burn-wgpu` or `burn-cuda` use GPU kernels directly.

Notable points:

- *Orchestration vs Compute*: Burn's Rust code orchestrates the training flow (data loading, model iteration, etc.) with safety and concurrency benefits. The heavy number crunching is delegated to whichever backend is chosen – e.g. the CUDA backend will call CUDA primitives (likely via Rust CUDA or FFI to C CUDA), the libtorch backend calls C++/CUDA implementations in PyTorch ¹⁰ ¹³. This design allows leveraging battle-tested C++ ML routines while keeping the outer logic in Rust.
- *Distributed/Device Support*: Burn is designed to run on “as many hardware as possible.” It supports multi-GPU training on a single machine, and although full distributed multi-node training might still be evolving, its backend-agnostic approach means it could integrate with distributed libraries (for example, a libtorch backend could use PyTorch's DistributedDataParallel under the hood). Burn explicitly mentions cloud vs edge deployment flexibility – train in the cloud with a CUDA backend, then deploy on an edge device with a CPU or WebGPU backend without rewriting the model ¹⁴ ¹⁵.
- *Interfaces*: Burn is primarily a library (for use in Rust programs or CLI tools), with good **documentation** (“burn-book”) and examples, but no built-in web GUI or server. However, one could wrap a Burn model in a CLI or web API easily. The key is its focus on code modularity and performance.
- *Documentation & Architecture*: Burn's docs outline its architecture: most of the code is generic over a `Backend` trait, enabling **swappable compute backends** ¹². The separation is so strict that, for example, the autodiff (autograd) is implemented as a **decorator** that can wrap any backend to provide gradient capabilities ¹⁶. This is a strong architectural pattern ensuring that “orchestration” logic (like the training loop and autograd bookkeeping) is written once in Rust, and you can **plug in different compute engines** (CUDA, CPU, etc.) beneath it ¹². This aligns well with the use case of adaptable ML architecture: training a Gaussian Process regressor could use a dense linear algebra backend today, and tomorrow a GPU-accelerated kernel backend – simply by implementing the appropriate `Backend`.

Assessment: Burn closely matches the vision of using Rust for orchestration with pluggable high-performance compute. It supports distributed training on multiple devices (and potentially multiple nodes with some extension) and has clear module boundaries. It demonstrates the technical feasibility of a Rust orchestrator driving C++/CUDA work: for example, using Burn's libtorch backend essentially makes Rust a controller over the optimized Torch C++ core. The project is well-documented and active, indicating a good reference for architecture and design patterns in hybrid ML systems.

Tch-rs (Rust Bindings to PyTorch C++ API)

The **tch-rs** library is not a full “system” but a crucial building block: it allows Rust programs to use **LibTorch** (PyTorch's C++/CUDA backend) for model training and inference ¹⁷. The crate provides thin wrappers so that Rust code can call into the highly-optimized PyTorch C++ kernels ¹⁷. In practice, this means you can write Rust code that defines a neural network, train it, and behind the scenes all tensor ops are executed by PyTorch's C++ engine (with GPU acceleration). This is a prime example of Rust

orchestrating heavy ML work done in C++: the Rust side handles things like model structure, loop control, and multi-threading (if needed), and offloads math to libtorch.

Relevance to the use case:

- *Orchestration/Compute Split*: By design, tch-rs keeps as close as possible to the original PyTorch API, just in Rust form ¹⁷. Rust takes the role of orchestrator – you could manage multiple training threads, distribute tasks, handle data pipelines – while libtorch provides the **scientific computing routines** (autograd, matrix ops, convolutions on GPU, etc.). This gives near C++ performance with Rust’s safety. Notably, one can utilize PyTorch’s *DistributedDataParallel* or other multi-GPU features via tch-rs, thereby achieving distributed training (local or across nodes) without writing any C++ manually. Essentially, tch-rs leverages PyTorch’s battle-tested distributed training under the hood. - *Multi-backend*: PyTorch supports CPU and CUDA seamlessly; by using tch-rs, a Rust program can similarly train on CPU or GPU by selecting the device. This isn’t a multiple *framework* backend scenario, but it is multi-hardware. (If needed, one could even call into different libraries via separate FFI, though tch itself focuses on libtorch.) - *Interfaces*: Projects using tch-rs are typically CLI applications or libraries. For instance, you could build a CLI tool in Rust for training a Gaussian Process model that internally uses tch for heavy tensor ops (perhaps implementing the GP training algorithm using torch tensor ops). While tch-rs doesn’t provide a GUI or server by itself, it’s straightforward to build a web API around it (e.g., using **Axum** or Actix-web in Rust to handle HTTP, and tch for model computations). This showcases that a **Rust service can embed C++ ML compute** in-process. - *Documentation & Example Code*: The tch-rs repo has examples (such as image classification, training small models) and the API mirrors PyTorch, so PyTorch’s extensive documentation mostly carries over. Its simplicity (just a binding layer) exemplifies the technical suitability of the Rust+C++ approach: it proves that Rust can manage complex C++ ML libraries safely. As noted in one tutorial, “tch-rs lets you do inference and training in Rust using the same underlying engine as PyTorch — but with Rust’s safety” ¹⁸.

Assessment: For a custom ML system, tch-rs could serve as the foundation where Rust is the orchestrator and PyTorch’s C++ is the compute engine. It doesn’t come with higher-level orchestration of clusters out-of-the-box (that would be up to your code), but its existence validates using Rust in place of Python for orchestrating ML training. A system built on tch-rs could implement, say, a Gaussian Process regressor training routine in Rust, and use GPU linear algebra from PyTorch for the actual computations. This approach directly leverages the rich PyTorch ecosystem (distributed training, efficient Tensor core kernels) under a Rust orchestrator.

HAL (Hyper Adaptive Learning with ArrayFire)

HAL is an older but illustrative project demonstrating a Rust orchestrator working with a C++ CUDA library. It uses Rust for the ML logic and ArrayFire (a C++ GPU-accelerated library) for numeric computations ¹⁹. HAL supports multi-GPU and multi-backend execution: specifically, it can run on **CUDA, OpenCL, or CPU** by delegating to ArrayFire’s backends ²⁰. Rust code in HAL defines neural network structures (like LSTMs, autoencoders) and the training loop, and when a tensor operation is needed, it calls ArrayFire’s library via Rust bindings. This model is quite relevant as it cleanly separates concerns:

- *Clear Module Separation*: HAL’s design required installing ArrayFire separately or via submodule, and the Rust side calls into it for operations like matrix multiplications, FFTs, etc. ¹⁹ ²¹. Thus, all orchestrating logic (model iterations, loop control, multi-GPU coordination) is in Rust, benefiting from Rust’s safety and concurrency, while all heavy linear algebra and GPU kernels execute in highly optimized C/C++ code (ArrayFire internally uses CUDA, cuBLAS, etc.). This is akin to Rust being a conductor and C++ being the musicians performing the intensive computations.

- *Distributed/Parallel Support*: HAL included **multi-GPU support** on a single node (both data-parallel and model-parallel aspects were under development) ²⁰. It did not natively implement multi-node distribution, but since ArrayFire can be used on any device visible to the process, one could manage a cluster by running multiple HAL instances and coordinating them (externally or via MPI). In concept, extending HAL's approach to distributed training would involve a Rust orchestrator process on each node (calling ArrayFire locally) and some message passing between them (which Rust could handle via MPI bindings or custom networking).
- *Interfaces*: As a library and set of examples, HAL was primarily CLI-driven (running examples via `cargo run`). It didn't provide web APIs or GUIs, but it wasn't hard to wrap one around it. The focus was more on the internal architecture to maximize performance. With Rust's safe multi-threading, one could spawn threads to utilize multiple GPUs concurrently from one process.
- *Documentation & Architecture*: The README highlights features and clearly lists the **capabilities provided by Rust vs the ArrayFire backend** – e.g., “Multi GPU support”, “OpenCL + CUDA + Parallel CPU support” ²⁰. It also credits ArrayFire's team and shows how to configure the backend via a JSON file ²². The architecture resembles an “**embedded orchestrator**” pattern: the Rust program itself is the orchestrator and directly calls a native library. This pattern is effective when you want minimal IPC overhead – Rust and C++ execute in the same process. HAL's approach proved viable and achieved good performance, but being an academic/older project, it's less actively maintained. Still, it's a relevant open-source example of Rust + CUDA synergy for training models.

Assessment: HAL demonstrates the technical suitability of Rust for orchestrating GPU computing. Even though it's from Rust's earlier days, it managed to integrate with a C++ CUDA library and run complex training tasks. This reinforces the idea that Rust can replace higher-level languages (like Python) in managing ML workflows, while still leveraging C++ for what it does best (number crunching on hardware). Modern projects have more ergonomic frameworks, but HAL's design (Rust + ArrayFire) is a straightforward blueprint for a hybrid system.

Additional Mentions

- **GraphPipe/ONNX with Rust:** Some projects wrap inference engines (like ONNX Runtime, TensorRT) with Rust APIs. For example, Microsoft's ONNX Runtime has Rust bindings, meaning a Rust orchestrator can load a model and let ONNX Runtime (C++ and CUDA underneath) do the compute. These are similar in spirit to tch-rs – Rust is the coordinator, C++/CUDA does the heavy lifting.
- **GPU Cluster Managers:** Tools like **GPUStack** (while primarily Python-based) show the importance of multi-backend and multi-interface support. GPUStack is an open-source GPU cluster manager that supports deploying models to various inference engines and hardware ²³ ²⁴. It provides a web UI, CLI, and API for orchestration. Although GPUStack itself uses Python for orchestration, one could envision a Rust equivalent providing the same – in fact, Dynamo and others are moving in that direction.
- **Rust-CUDA and WGPU:** An alternative to relying on C++ for GPU work is using Rust's emerging GPU libraries. Projects like **Rust-CUDA** and **wgpu** allow writing GPU code in Rust. For instance, **Burn** can use WGPU (via Rust) for compute. However, these are still maturing. In the interim, using established C++ CUDA libraries (cuBLAS, cuSolver, or entire frameworks like oneAPI, ArrayFire, etc.) via Rust FFI is often the most pragmatic route. The examples above leverage that approach.

Technical Suitability of Rust for Orchestration vs C++/CUDA for Compute

Using **Rust for orchestration** and **C++/CUDA for heavy ML tasks** is a technically sound strategy, combining the strengths of both ecosystems. Here's why each is suitable for its role:

- **Rust as Orchestrator:** Rust offers performance close to C++ with strong memory safety guarantees and fearless concurrency. In complex ML training systems, this means the coordinating logic (managing threads, data loading, networking between nodes, etc.) can run efficiently without garbage collection pauses or race conditions. Rust “delivers the low-level control needed for performance-critical components — but without the memory-safety issues” common in large C++ codebases ²⁵. This makes it ideal for orchestrating distributed training, where subtle bugs or crashes in the control logic can be catastrophic. Rust's reliability and modern tooling also make the system more maintainable. Moreover, Rust integrates well with Python (via PyO3) and C/C++ (via FFI), which means it can serve as the glue in a multi-language system. As one analysis put it, teams increasingly use Rust to replace the “performance-critical 10% of code” while still using Python or others for higher layers ²⁶. In our context, Rust is that performant orchestrator ensuring that GPUs across the cluster are kept busy and synchronized. It can handle multi-threading and asynchronous I/O (e.g. using Tokio) to stream data and coordinate tasks, far outperforming Python in those aspects (no GIL to worry about).
- **C++/CUDA for Compute:** C++ (especially with CUDA) remains **the king of high-performance ML kernels**. Most state-of-the-art ML libraries (PyTorch, TensorFlow, cuDNN, etc.) are written in C/C++ for a reason – they can directly leverage GPU instructions, optimized BLAS libraries, and fine-tuned memory management. By using C++/CUDA for heavy lifting, we ensure we're not reinventing highly optimized algorithms in a slower or less optimized language. Rust could theoretically be used for GPU kernels (and projects exist to do so), but the CUDA ecosystem and tools are far more mature. Thus, tapping into C++/CUDA gives immediate access to years of performance engineering. Additionally, many advanced ML techniques (like Tensor Core utilization, efficient low-level parallelism) are available in C++ libraries. A Rust orchestrator can call these and get “C++-level speed” where it matters ²⁷ ²⁸. The trade-off is that one must interface across language boundaries, but Rust's FFI is quite straightforward and zero-cost for C calls, so the overhead is minimal. In practice, as seen in the examples (tch-rs, Burn, HAL), this combination yields near-native performance with the benefits of Rust managing the show.
- **Synergy:** The reason this hybrid is powerful is that modern ML workloads demand both *flexibility* and *performance*. Python is flexible but slow; pure C++ is fast but cumbersome at scale. Rust hits a sweet spot: it's much safer and more ergonomic than C++ for high-level orchestration (preventing memory leaks, data races in multi-threaded trainers, etc.), and it can still call into C++/CUDA for raw speed where needed. In fact, many cutting-edge AI systems quietly pair Python with C++ behind the scenes – replacing Python with Rust for the coordination layer simply makes the system more robust without sacrificing speed. **Rust does not replace C++ in compute-heavy inner loops; it enables using C++ effectively by controlling it from a safe environment.** This separation of concerns improves the overall system reliability.

In summary, Rust is technically very well-suited as the orchestration layer (fast, safe, concurrent) and C++/CUDA remains the go-to for number-crunching (unmatched low-level performance, huge existing ML library support). The hybrid approach takes advantage of each for what they do best, and open-source projects are already validating this strategy (as shown above). As one article succinctly noted: “Most

'Python speed' is actually C++ speed, wrapped in Python" ²⁶ – by using Rust as the wrapper instead, we get that speed with stronger guarantees and better scalability.

Architecture Patterns for Such a System

Designing a distributed training system with Rust orchestrating and C++/CUDA compute can follow several architecture patterns. The best choice depends on requirements like scalability, fault tolerance, and development complexity. Below are some patterns and how they fit this use case, along with justification:

- 1. Client-Server (Master-Worker) Architecture:** This is a straightforward pattern where a central **Rust** process (master) coordinates work, and multiple worker processes (which could be Rust programs linking C++ libs, or pure C++ programs) perform the ML training tasks. The master sends out tasks (e.g. "train this batch" or "compute this gradient chunk") to workers and aggregates results (gradients, metrics). This model is relatively simple to implement and reason about ²⁹. It clearly separates roles: the server orchestrates, the clients compute. It's suitable for distributed setups where you might have one node act as parameter server or job scheduler and the others do heavy compute. The **advantage** here is clarity and centralized control (e.g., easier to implement synchronous SGD or parameter server for Gaussian Processes). However, the master can become a bottleneck or single point of failure. In practice, frameworks like PyTorch's DDP use a form of this (each process is a worker and a rank 0 worker or an external scheduler coordinates startup). A Rust master can be very efficient at handling coordination (using async IO to handle many worker connections concurrently). Use this pattern if your training can be centrally coordinated and you want simplicity.
- 2. Microservices Architecture:** In a microservices approach, the system is broken into several independent services (which can be deployed and scaled separately) ³⁰ ³¹. For example, you might have a **Training Service** (Rust orchestrator) that handles high-level scheduling of epochs and evaluation, a **Compute Service** (perhaps a thin server around the C++/CUDA kernels) that accepts tasks like "compute gradient for batch X" or "evaluate kernel matrix for GP", and maybe other services like a **Data Service** (streaming data shards), and a **Monitoring Service** (tracking metrics, progress). These services communicate via APIs or message queues. **Pros:** This adds modularity and scalability – each component can be scaled horizontally; if, say, the compute service is the bottleneck, you can run more of them behind a load balancer ³² ³¹. It also improves resilience: one service crash doesn't bring down the whole system – e.g. if a worker service goes down, the orchestrator can detect and restart it, while others keep running ³¹. **Cons:** Increased complexity in deployment and communication. You'll need to design robust APIs between services and handle network latency. That said, Kubernetes or container orchestration can manage a lot of this. For our use case, microservices make sense if the system is large-scale or multi-tenant (for instance, a service managing different ML experiments or multiple models in parallel). A real-world analog is Kubernetes + gRPC: Rust could be used to implement each microservice (ensuring memory safety), and C++ libraries are used within the services for compute. This pattern best serves cases where you want **flexibility and robust scaling**, at the cost of additional engineering overhead.
- 3. Actor Model (Distributed Actors):** The actor model treats components of computation as **actors** – independent entities with their own state and mailbox. Actors communicate by sending messages asynchronously. In Rust, there are actor frameworks (like Actix or **Riker**); while not as famous as Akka (Scala) or Orleans (.NET), they exist. An actor-based architecture for distributed training might have actors like **ParameterServerActor**, **WorkerActor** (one per GPU or per data

shard), etc., where sending a message triggers an operation (e.g., a worker actor receives a “compute gradient” message). The actor model can naturally extend to multiple nodes if the actor framework supports networking (or you build it by assigning actors to nodes). **Pros:** It simplifies reasoning about concurrency – no shared mutable state by design, only message passing, which is great for avoiding race conditions. It also can dynamically scale: you could spawn more worker actors on different machines and the system logic doesn’t change much. PyTorch’s **Monarch** (a distributed training toolkit) uses an actor-like approach internally, showing this model maps well to ML training (each training process is akin to an actor) ³³. **Cons:** Implementing a truly distributed actor system from scratch in Rust is non-trivial – you’d either use an existing crate or need to handle message routing, actor discovery, etc. Latency can also be a concern if not managed, since everything is message-based. However, Rust’s speed and safety again help here – high message throughput and no data races in actors. This pattern is justified when the system demands **high concurrency and dynamic behavior** (e.g., scaling up/down workers on the fly, or having many fine-grained actors such as one per dataset shard or model partition). It provides a structured way to build a complex distributed system without hard-coding a client-server hierarchy. In essence, the actor model could make the orchestrator itself distributed: instead of one master, you have a collection of coordinator actors possibly on each node (which could eliminate the single point of failure while still coordinating via messages).

Choosing a pattern: For a **first implementation**, a client-server model might be easiest to get working for distributed training – one Rust master orchestrating many worker processes (which could just be the same binary started with different roles, or a separate C++ binary if needed). This is akin to how Horovod or Parameter-Server approaches work. As the system grows, you might refactor toward microservices (for better modularity – e.g., separate services for training vs inference in a unified platform) or incorporate actor frameworks to manage complexity. The microservices approach is often used in production ML platforms (e.g., a scheduling service, a training service, a model serving service all separate), and Rust’s ability to create small, efficient services that communicate (via REST or RPC) suits that well. On the other hand, the actor model could shine in a scenario where the boundary between local and remote execution is fluid – for instance, using an actor library, your code could almost treat local vs remote actors the same, simplifying distributed logic.

Regardless of the pattern, the **modular separation of concerns** remains key. Rust’s strong compile-time guarantees encourage a modular design (e.g., separate crates for “orchestration logic”, “ML compute FFI bindings”, “communication protocol”, etc.). Meanwhile, C++/CUDA will likely reside in its own modules (shared libraries or services) that Rust calls into. All the mentioned patterns can accommodate that separation. For example, in a microservice design you might have a `train-worker` service (Rust + C++ library) and a `train-orchestrator` service (pure Rust coordinating via gRPC); in an actor model, you might have one actor type that internally uses C++ kernels for compute, and another actor type making high-level decisions.

Justification Summary: The client-server pattern is *simple and clear* – best when the system is not extremely large-scale ³⁴. Microservices introduce *flexibility and fault isolation*, suitable for complex or large-scale systems that may evolve ²⁹. The actor model offers a *clean concurrency model and scalability* at the design level, which can simplify certain types of distributed logic (especially dynamic interactions), though it requires more sophisticated infrastructure. Given the use case (training Gaussian Process models and beyond, potentially on clusters), one might start with a client-server (a Rust CLI tool that can launch distributed jobs via MPI or a custom coordinator). As needs grow (multiple different models, user requests for training via an API, etc.), evolving toward a microservices architecture with a Rust web API orchestrator and separate worker services could be beneficial. Throughout, the decision to use Rust and C++/CUDA remains justified: Rust provides the *“safe and predictable performance”* needed in the

orchestration layer ³⁵, and C++/CUDA delivers the raw speed for mathematical computations – together enabling a system that is both robust and high-performance.

Sources:

- NVIDIA Dynamo – Rust orchestrator with multi-backend GPU inference ¹ ²
- Burn ML Framework – Rust training framework with swappable CUDA/CPU backends ¹² ¹⁰
- tch-rs – Rust bindings leveraging PyTorch C++ (libtorch) for heavy compute ¹⁷
- HAL (Rust + ArrayFire) – multi-GPU ML in Rust, using CUDA/OpenCL C++ library for math ²⁰ ¹⁹
- Techn0tz Blog – Why Rust excels at orchestrating heavy workloads (memory safety, concurrency) ²⁵ ²⁶
- Software Architecture Basics – Notes on client-server vs microservices trade-offs ²⁹ ³⁰

¹ ² ³ ⁴ ⁵ ⁶ ⁷ ⁸ ⁹ GitHub - ai-dynamo/dynamo: A Datacenter Scale Distributed Inference Serving Framework

<https://github.com/ai-dynamo/dynamo>

¹⁰ ¹¹ ¹² ¹³ ¹⁴ ¹⁵ ¹⁶ GitHub - tracel-ai/burn: Burn is a next generation tensor library and Deep Learning Framework that doesn't compromise on flexibility, efficiency and portability.

<https://github.com/tracel-ai/burn>

¹⁷ GitHub - LaurentMazare/tch-rs: Rust bindings for the C++ api of PyTorch.

<https://github.com/LaurentMazare/tch-rs>

¹⁸ Image Classification in Rust with Tch-rs (Torch bindings) - Djamware

<https://www.djamware.com/post/690864cde87a290bcfebeebe/image-classification-in-rust-with-tchrs-torch-bindings>

¹⁹ ²⁰ ²¹ ²² GitHub - jramapuram/hal: Rust based Cross-GPU Machine Learning

<https://github.com/jramapuram/hal>

²³ ²⁴ GitHub - gpustack/gpustack: GPU cluster manager for optimized AI model deployment

<https://github.com/gpustack/gpustack>

²⁵ ²⁶ ²⁷ ²⁸ ³⁵ How Rust Is Quietly Becoming the New Backbone of AI, ML, Data Science, Big Data, and Quantum Computing | Techn0tz

<https://manjushaps.github.io/Rust-Series-FutureTech/>

²⁹ ³⁰ ³¹ ³² ³⁴ Building the Future- Choosing Between Client-Server, Monolithic, and Microservices. | by Kulan Thamuditha | Medium

<https://medium.com/@kulanthamuditha66/building-the-future-choosing-between-client-server-monolithic-and-microservices-2a194677f8eb>

³³ Introducing PyTorch Monarch

<https://pytorch.org/blog/introducing-pytorch-monarch/>