BytesAndBrains: A Substrate for Networked Machine Learning

Matthew Love
Bytes and Brains LLC
matthewlove@bytesandbrains.ai

June 2026

Abstract

Machine learning increasingly runs outside the data center, and the strategies the field has produced for that setting (federated learning [1], gossip learning [2], split learning [3], peer-to-peer inference) do not compose: BytesAndBrains is the substrate they can share. Each strategy ships with its own reference code, its own RPC stack, and its own training loop, and the substrate that would let them sit on a common foundation does not exist in the open ecosystem. BytesAndBrains records a workload into an ONNX-backed [4] intermediate representation, partitions the recording across nodes, and owns the structured byte layer between them. The runtime is sans-IO [5]: a state machine driven by the host program, with every distributed behavior testable in-process. The extension surface is composed of *Components*, typed plug-ins that satisfy *Roles*, runtime contracts the substrate dispatches against. This paper positions BytesAndBrains relative to Flower [6], Ray [7], Substra [8], libp2p [9], TensorFlow Federated [10], Hivemind [11], and Spark [12], and identifies the points at which the substrate occupies a different layer of the stack.

1 Introduction

Machine learning runs in two operating envelopes. One is the data center: homogeneous clusters, fast interconnect, a single operator, one trust domain. The other is everything outside the data center: phones, sensors, hospital networks, on-premises fleets, and peer-to-peer overlays. The strategies the field has produced for the second envelope (federated learning [1], gossip learning [2], split learning [3], peer-to-peer inference) all address a single underlying question: how to run machine learning where the data already lives.

These strategies do not compose. Each ships with its own reference implementation: a one-off RPC stack, a paradigm-specific training loop, and a coordinator process or peer-sampling implementation hard-wired to one transport. A research group attempting to compare three strategies on the same dataset writes three networking stacks. The cost is borne in every empirical study and every reproducibility effort.

BytesAndBrains is the substrate these strategies share. It takes a description of a computation, partitions it across nodes, and owns the bytes between them. Federated, gossip, split, and peer-to-peer learning become bindings on the same surface. The remainder of this paper describes the design (Sections 3 through 7), names BytesAndBrains' boundaries (Section 8), positions it relative to existing systems (Section 9), and concludes (Section 10).

2 Background and Motivation

The substrate this paper describes does not exist in the open ecosystem. The layers around it do (Figure 1). PyTorch [13], TensorFlow [14], and JAX [15] provide tensor compute and model-authoring DSLs. Ray [7] and Spark [12] provide cluster scheduling and actor-based execution. libp2p [9] provides peer-to-peer transports, multiplexing, peer identity, and discovery primitives. Between these layers sits a missing component: a substrate that authors computation, partitions it across nodes that may be reachable only through networks it does not own, executes each partition without assuming what kind of process hosts it, and ships bytes between participants without committing to any particular transport.

The strategies the field has produced for the second envelope are technically valuable. Federated averaging [1] established the coordinator-driven round model and is the de facto starting point for cross-device learning.

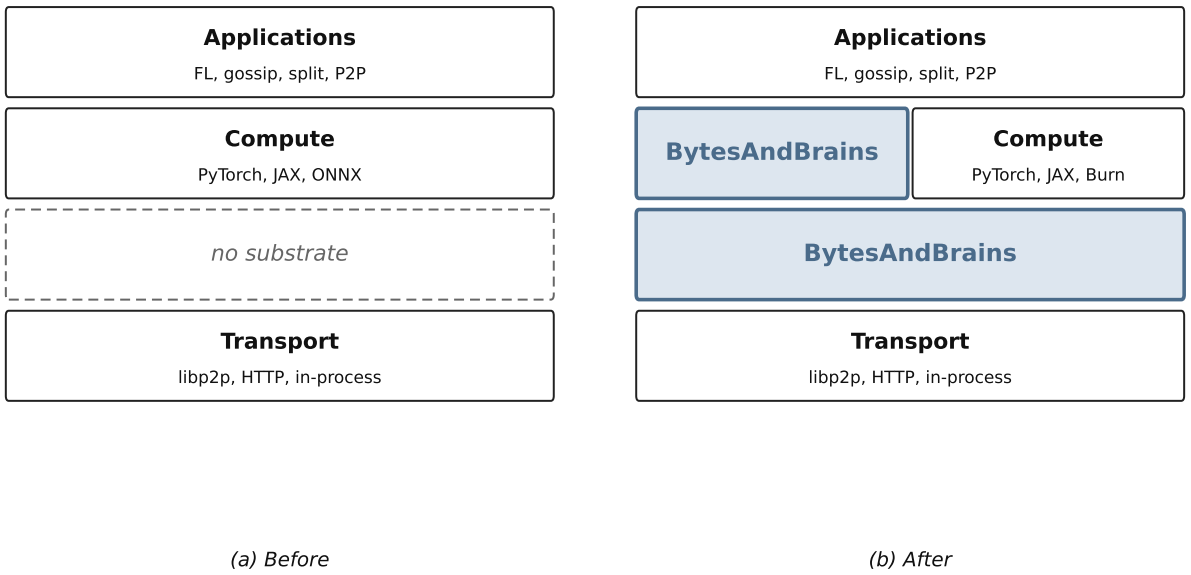

Figure 1: The layering claim. (a) PyTorch, JAX, and ONNX provide tensor compute; libp2p, HTTP, and in-process channels provide transport. Between them sits a missing substrate. (b) BytesAndBrains is both the authoring DSL (*Module*, *Graph*, *Roles*) and the substrate beneath it (IR, compiler, engine, wire envelope); tensor frameworks plug in as Backend Components.

Gossip learning [2] generalized it to a fully decentralized exchange with age-weighted merging. Split learning [3] partitions the model itself between participants. Push-sum aggregation [16] established asynchronous averaging primitives still used in production gossip protocols. Their reference implementations are standalone artifacts glued to bespoke RPC stacks and training loops; framework-level reuse across them requires substantial reimplementation.

A substrate that consolidates these strategies must satisfy specific constraints. It is a framework rather than a library, because partitioning is a global concern that user code cannot make in isolation. It uses a portable intermediate representation, so the same computation survives crossing language and process boundaries. It is sans-IO [5], so the host program retains exclusive control over its runtime and sockets. It owns the byte layer but stops there; existing transports are well-developed, and it does not invent its own. It is composable at the role level, so the building blocks of the field's strategies (compute backends, models, indices, aggregators, peer-sampling overlays, custom protocols) plug in as distinct, swappable components.

3 System Overview

BytesAndBrains performs three operations.

First, BytesAndBrains records the user's computation into a portable intermediate representation. A workload is authored as a *Module*, a typed handle whose recording method walks the description once and produces a graph in the IR. From that point forward, the IR is the program; the host language no longer participates in subsequent processing.

Second, the compiler partitions the recorded program across nodes. It reads the IR, decides which operations live on which node, and emits one partitioned program per node. Cross-node edges in the original recording become explicit communication operations in the per-node outputs, inserted by the compiler rather than the user (Figure 2).

Third, BytesAndBrains owns the byte layer between nodes. Every byte sent or received rides as a structured envelope, whose encoding, routing, and correlation the framework owns. The host owns the sockets that move the envelope. A transport adapter mediates the boundary.

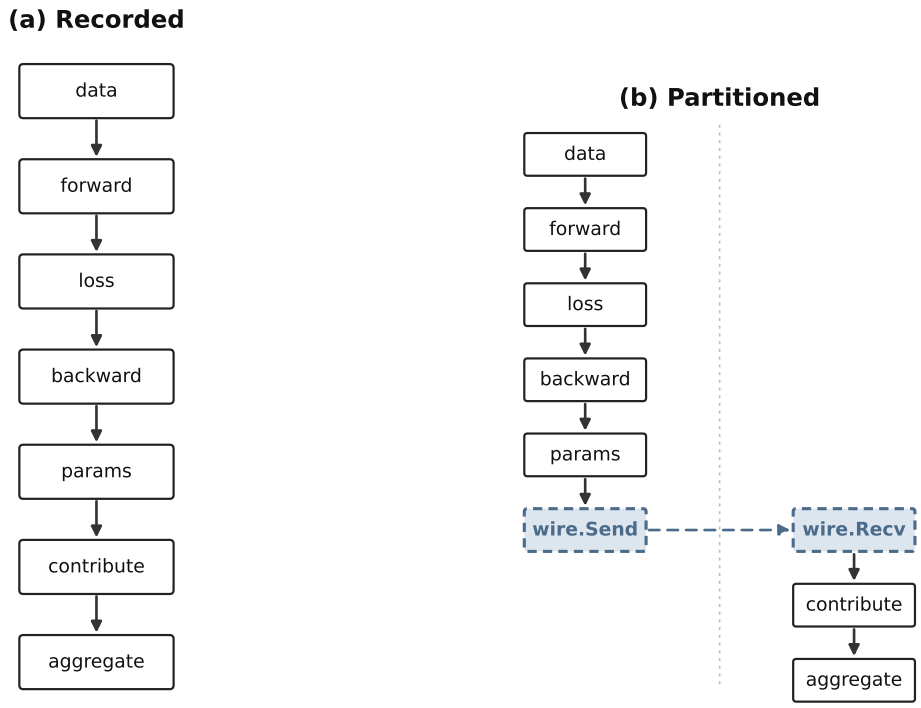

Figure 2: Graph partitioning across peers. (a) The recorded computation: one dataflow graph, no peer boundaries. (b) After compiler partitioning: the compiler decides which operations live on which node and inserts `wire.Send` / `wire.Recv` (dashed, slate-blue) at every cross-node edge. The same graph becomes two per-node programs that exchange typed values through one structured envelope.

The discipline that holds the design together is sans-IO. The library crate contains no asynchronous runtime, no shared sockets, and no thread-pool dependency. The runtime is a state machine. The host calls a poll function on it; the state machine drains pending inputs, walks the program to a fixed point, and returns outbound envelopes. The host ships those envelopes and feeds inbound bytes back. All other runtime decisions (event-loop selection, transport choice, reconnection policy, logging) belong to the host. BytesAndBrains is a coordination and authoring layer; Component composition handles backend, transport, security, and deployment-shape concerns.

4 Intermediate Representation

The choice of IR is load-bearing. A framework that partitions computation across nodes requires a representation it can analyze, rewrite, and serialize independently of the host language. BytesAndBrains uses ONNX [4].

ONNX defines a typed, directed acyclic graph IR and an opset of named operations. It serves as the de facto cross-framework format for trained models, with tooling and runtime support across PyTorch [13], TensorFlow [14], JAX [15], and ONNX Runtime. Three properties make it appropriate here.

Portability. The IR is a serializable protocol buffer. A recording produced by a host in one language can be read by a host in another. The authoring surface is in Rust, but the IR does not constrain that choice; alternative front-ends in Python or other languages remain admissible.

Community vocabulary. ONNX op definitions are agreed by a community of toolmakers and framework authors who care about how forward and backward passes are described, what a typed tensor means, and how control-flow nodes behave. A backend author who already understands ONNX implements one for BytesAndBrains without learning a new IR. Models authored elsewhere export to ONNX format; Backend Components consume ONNX nodes directly or translate to their preferred runtime (the reference `CpuBackend` executes ONNX nodes; an ExecuTorch backend would translate-and-cache; an ONNX Runtime backend would pass through directly).

Layered extension. BytesAndBrains defines additional opsets on top of ONNX's base (runtime intrinsics, inter-node value transfer, role-specific verbs) while serializing to a single graph proto. No parallel custom IR exists. The compiler is a series of pure functions over the graph proto. The wire envelope carries serialized graphs across the network. ONNX-aware tooling applies directly to recordings it produces.

Typed contract for cross-node values. The IR is typed end-to-end: every Component slot, every cross-node edge, every wire envelope carries a typed contract that both ends must agree on. The framework surfaces an error when an envelope arrives with a type the receiver does not recognize; it does not provide a mechanism for renegotiating types at runtime. Two nodes that exchange typed values agree on the type system at deployment time, and cross-version compatibility is a deployment concern, not an IR concern.

The cost of this choice is concrete. ONNX's opset is the community's opset; decisions baked into it cannot be relitigated locally. Versioning must coordinate with the broader ecosystem. These costs are accepted in exchange for portability and ecosystem reach.

5 Runtime Model

A node is a runtime container hosting an execution engine. The engine owns the per-node state required to run a partitioned program: the dispatch table mapping operations to bound Components, the slot table holding intermediate values, the frontier of operations ready to fire, the ingress queue for external events, the tracking for long-running asynchronous operations, and a small set of framework primitives (a scheduler, an event bus, a request tracker, and an outbound envelope queue).

The container is a state machine. The host calls a poll function on it. The function drains pending inputs, walks the program's directed acyclic graph to a fixed point, and returns a vector of outbound envelopes. External threads interact with the engine exclusively through the ingress queue, which constitutes the single thread-safe boundary. Within the poll cycle, every operation has exclusive access to engine state.

Three design properties follow.

Single-threaded dispatch, concurrent execution. The engine dispatches operations sequentially within a poll cycle. Each op borrows mutable references to shared framework primitives, and the single-threaded dispatch loop is what keeps the engine state machine straightforward to reason about. Op *execution* is not constrained to that loop. A Contract method that returns `ContractResponse::Later(cmd_id)` releases the engine to drain the rest of the frontier while the op runs to completion on whatever runtime the host supplies (a thread pool, an async executor, a GPU stream, a remote service). At this layer, BytesAndBrains is a networked scheduler: the engine orchestrates which op fires when, and the Component implementation chooses how to execute. Components are free to be fully threaded if their runtime supports it.

Sans-IO discipline. The library crate contains no asynchronous runtime dependency, no shared sockets, and no spawned tasks. Two consequences follow. Every system built on it is testable in-process: a harness instantiates multiple nodes in one process, connects them by in-memory channels, and exercises the full distributed behavior without opening sockets. The library also ports to any host runtime that can shuttle bytes and call poll, decoupling framework logic from the host's runtime choice.

Snapshot as a first-class concept. Components that maintain state implement two methods: serialize current state to bytes and reconstruct an instance from bytes. The compiler assigns each stateful instance a stable snapshot key at build time. A node-wide snapshot walks the dispatch table, captures each Component's

state, and emits a blob; a restore rebuilds the same shape. Snapshot captures runtime state for replay-based debugging of distributed failures, or for migration of a Node between hosts. A captured snapshot plus the recorded ingress stream reproduces the failing trajectory deterministically, and the same blob hands a Node off when a host is decommissioned.

Memory model. BytesAndBrains distinguishes external byte payloads (transport-owned, ephemeral) from internal byte ownership (framework-owned, durable). External boundaries entering the engine (network adapter ingress, application event push, async completion) take payloads as borrowed slices (`&[u8]`) and copy them into framework-owned storage inside the boundary call. Many transport stacks hand the framework a pointer into their own memory (libp2p reads, QUIC completion buffers, kernel ring buffers); the framework must not keep a reference past the call and must not assume the transport's allocator owns the buffer. The copy is the ownership transition, not an inefficiency. Inside the framework, byte ownership flows normally: a `Vec<u8>` moves between framework-owned slots, and an `Arc<...>` handle threads backend-managed tensor buffers across slots without deep copy. The single mandatory copy is at the boundary.

Error semantics at the boundary. External boundaries use fallible allocation (`try_reserve_exact`) and a per-node byte budget. On allocation failure or budget exceedance, the engine emits a typed bus event (`InfraEvent::WireReceiveError::AllocationFailed` or `::BudgetExceeded` on the wire surface; `InfraEvent::AppIngressError` with parallel variants on the application surface), drops the offending bytes, and continues processing other envelopes and events. The engine never panics, never aborts, and never stalls at the boundary. Inside the engine, normal Rust patterns apply: components are designed to play by the runtime contract, so a per-component allocation failure inside a Contract body is a process abort, not a framework-handled event. The fallibility line lives at the engine boundary itself. Per-source caps (per-event, per-invoke, per-completion) reject oversize single payloads with typed errors; the cumulative byte budget guards sustained load. The two layers compose: the per-source cap protects against single adversarial inputs, and the budget protects against ensemble pressure.

Observability. The runtime is instrumented through three surfaces. Tracing spans on `engine.poll` (and its sub-phases `engine.phase1_ingress`, `engine.phase2_frontier_drain`, `engine.phase5_completions`, `engine.phase8_outbound`) bound each poll cycle. Each operation invocation opens a structured `engine.invoke_one` span with `op.name`, `op.kind`, `op.domain`, `exec_id`, and `op_ref` fields, so traces filter and group by op kind or by specific dispatch instance. The event bus carries `NodeEvent`, `InfraEvent`, and `AppEvent` enums as the in-process observability surface; the host subscribes to drive metrics, logs, and external tracing without coupling to a specific telemetry backend.

Multiple targets per Node. A Node is a runtime container, not a Module. A single Node can install multiple Module targets, and those targets share the named Component bindings the host registered at construction. Authors compose distinct behaviors as separate Modules (a client target that trains and uploads, a server target that aggregates and rebroadcasts); the host installs each target onto the Node it wishes to run that behavior. Bindings are looked up by name, so two targets referencing the same Component name resolve to the same instance. The canonical example is federated learning, where one peer may take on both the client and the server role against a shared model: the Node installs a `client` Module and a `server` Module, both targets bind the same Backend, the same model parameter store, and the same PeerSelector, and the engine dispatches against the shared instances without coordination between the targets. The same shape covers gossip-and-coordinate hybrids, validator-and-participant peers, and any deployment where one runtime container plays multiple roles in the larger computation. Multi-target install is the seam; shared bindings are the mechanism.

6 The Byte Layer

When a partitioned program needs to send a value to another node, the framework wraps the value in a structured envelope and asks the host to ship it. When the host receives an envelope, it hands it back to the framework, which routes it to the destination. The envelope is the contract between framework and host.

Above it, BytesAndBrains owns dispatch and encoding. Below it, the host owns sockets, encryption, identity, and the network stack.

Two semantically distinct messaging planes share the envelope format. The *data plane* carries values across the cross-node edges the compiler inserted: encoding, request and response correlation, batching of multiple edges to a common peer in a single cycle, and routing to the correct destination slot. The user does not see this plane; partitioning produces it. The *control plane* carries protocol traffic for Components that speak to peers: gossip protocols, DHT-style overlays, consensus algorithms. Inbound envelopes for these protocols route to the appropriate Component; the Component owns encoding and decoding of its own messages. Both planes use the same envelope format. The envelope's protocol identifier distinguishes them.

The transport-adapter contract is small. Outbound: encode the envelope as framed bytes and ship through whatever channel the host has chosen. Inbound: decode framed bytes as an envelope and pass to the framework. The adapter consumes whatever peer identity, multiplexing, encryption, and discovery the underlying transport supplies; the framework does not assume any of them.

The choice of transport is a deployment concern. Peer-to-peer overlays, request-response HTTP, and in-process channels (for tests, simulators, or embedded multi-node deployments inside one host process) all satisfy the same contract. BytesAndBrains owns the envelope and terminates at the adapter boundary.

Identity and replay. The envelope carries a transparent sender identifier. BytesAndBrains does not authenticate that identifier; production deployments use transports with built-in identity (libp2p peer ids [9], mutual TLS, or equivalent). The envelope carries request and response correlation but no anti-replay nonce; the same transport layer supplies replay protection. Side-channel mitigation (constant-time codec implementations, padding, traffic shaping) lives in the Component that handles the sensitive data.

Backend-mediated tensor receive. A wire receive whose destination slot binds a Backend Component routes through that Backend rather than through a generic decoder. The engine caps the inbound fill's byte length against per-envelope caps, charges the length against the node's ingress byte budget, and hands the framework-owned bytes to the Backend by value. The Backend chooses the materialization strategy: zero-copy adoption when alignment permits, a buffer drawn from its own pool, or a fresh allocation. The resulting backend-native tensor is wrapped in an internal handle that carries the byte charge so the slot-table writer can release it on overwrite or eviction. This is the framework-to-Component handoff, not an external boundary; the borrowed-slice rule that governs transport ingress does not apply, because the Backend lives inside the framework ecosystem and plays by the runtime contract. The Backend interface (Section 7) names the single Contract method (`materialize_from_wire`) that participates in this hand-off and supplies a default that delegates to the global wire decoder registry, so backends without tensor pooling continue to work without modification.

6.1 Network reality

Production networks impose constraints BytesAndBrains names but does not solve in isolation. The framework provides composition seams; Components and protocols implement the policies.

NAT traversal. Peer-to-peer overlays solve NAT through hole-punching, relays, and rendezvous coordination [17, 9]. BytesAndBrains is transport-agnostic; libp2p adapters consume these mechanisms directly. The `Module::bootstrap` surface lets protocols express NAT-friendly behaviors (client-first-speak, server-driven heartbeat) as graph operations rather than transport-specific code.

Intermittent connectivity. Asynchronous federated rounds and disconnection-tolerant gossip are addressed by communication-efficient FL protocols [18] and production federated systems [19]. Hop-local back-off rides as a typed wire op and per-site liveness is tracked through the φ-accrual failure detector [20]. The PeerSelector role provides the seam where disconnect-aware cohort selection (drop, stash, requeue) is expressed; the Aggregator role determines what a partial round means.

Asymmetric bandwidth. Communication-efficient federated learning explicitly addresses uplink-dominated cost asymmetry through gradient compression, sketching, and structured updates [18, 21]. The Codec role provides the encode-decode seam where compression schemes compose with the wire layer; the Aggregator role consumes the decompressed payloads. BytesAndBrains ships no codec; it defines where one composes.

Async timeouts and backpressure. Tail-latency mitigation through per-call deadlines is established practice [22]. Dapper-style per-envelope deadlines [23] are derived at compile time from graph-walk chain depth and refined at runtime through Karn-Partridge RTT smoothing [24, 25]. Backpressure under congestion is well-studied in event-driven systems [26]: hop-local back-off rides as a typed wire op, receiver-side detection composes with φ-accrual liveness, and the PeerSelector role provides the seam where exponential-backoff and circuit-breaker policies are expressed.

7 Extension Surface

Pluggability distinguishes BytesAndBrains from a one-paradigm library. The extension surface is built from two abstractions: Components and Roles.

A **Component** is a typed plug-in. The only structural requirement is a lifecycle contract: a stable type identifier, a serialization method, and a reconstruction method. The engine dispatches operations to Components. Components are otherwise free to assume whatever internal shape their logic requires.

A **Role** is a runtime contract defining a category of behavior. Several role traits ship with the framework. A compute backend Role executes tensor operations. An index Role looks up values by key or similarity. An aggregator Role combines parameter shards from multiple peers. A data source Role produces batches. A peer-sampling overlay Role draws peers from a view of the network [27]. A custom protocol Role speaks its own opset to peers.

Each Role corresponds to a trait. A Component implements the Role traits it intends to satisfy. A single-Role Component is a single-purpose plug-in. A multi-Role Component exposes several surfaces from one piece of state (Figure 3). A Kademlia-style DHT [17] naturally implements three Roles: it is an index (values stored in the overlay are retrievable by key), a peer-sampling source (peers are drawn from its routing table), and a protocol (it speaks its own message types to peers). One piece of state, one piece of code, three Roles. The engine dispatches by Role without requiring knowledge of how many Roles any given Component carries.

The surface intentionally lacks a central registry, a plug-in protocol, or any compile-time ritual beyond trait implementation. The polymorphism is the trait system. Library authors ship Components in standalone crates. Users bind those Components to Roles at node construction. The framework remains small; the Components constitute the system.

Backend interface. The compute backend Role is the surface for tensor compute. A Backend Component declares an associated `Tensor` type (the backend's native tensor handle, which is typically an `Arc`-shared handle around a backend-managed buffer for cheap intra-node clones), implements either a per-operation method surface (one method per mandatory tensor primitive) or a whole-graph `execute` method, and supplies one Contract method that participates in the wire path: `materialize_from_wire(type_hash, bytes)` consumes the framework-owned wire bytes by value and returns a backend-native tensor. A default delegates to the global wire decoder registry, so backends without tensor pooling continue to work; backends that override pay the registry hop only at override time. This is the framework's single hook for routing cross-node tensor values through the Backend's allocator. Existing tensor runtimes (the reference CpuBackend, an ExecuTorch wrapper, a Burn integration, an ONNX Runtime adapter) implement the Backend Role by satisfying its trait contract; dispatch lands against the contract surface without coupling to any particular runtime.

Federated-learning composition seams. Production federated learning addresses gradient poisoning, membership inference, model inversion, and differential privacy as distinct threats [19, 21, 28, 29, 30]. The framework ships none of these defenses; it provides composition seams. `Compiler::bind_aggregator`

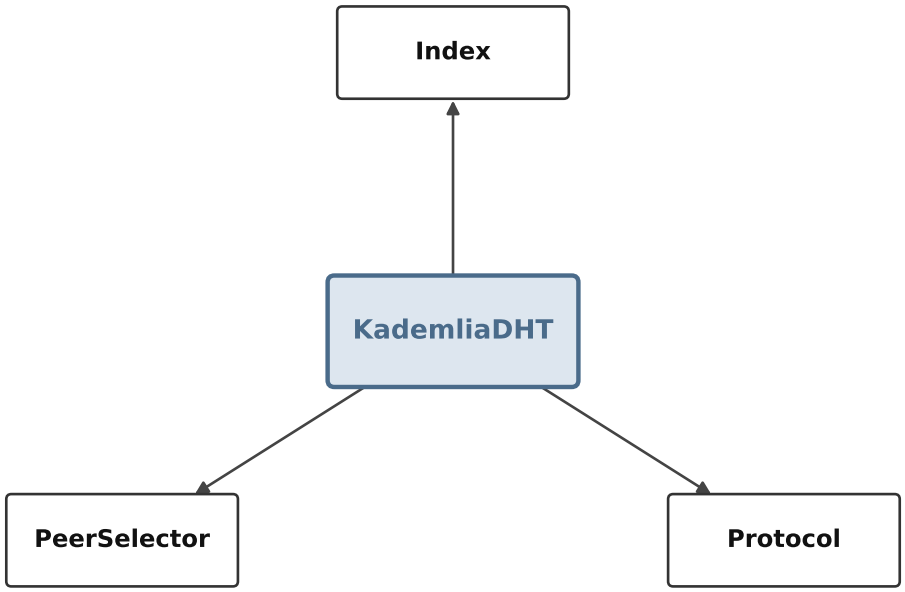

Figure 3: Multi-role Component composition. A Kademlia-style distributed hash table holds one piece of state (a routing table and value store) and implements three Role surfaces: `Index` (`add`, `search`, `remove`, `train`), `PeerSelector` (`select`), and `Protocol` (`dispatch_atomic`). The Compiler binds the same component instance at three slots (`bind_index`, `bind_peer_selector`, `bind_protocol`); the engine dispatches each call to the role-matching method set. The framework carries no opinion on how many Roles a Component implements.

is where Byzantine-robust reductions compose [28]. `Compiler::bind_codec` is where encrypt-aggregate-decrypt and DP noise compose. `Compiler::bind_peer_selector` is where attesting cohort gating composes. Deployments that need these defenses choose Component implementations that supply them; the framework is silent on policy.

Layered responsibility. BytesAndBrains names each concern, owns one layer of the stack, and exposes a composition seam for the rest.

Concern	Owner	Composition seam
Tensor compute	Backend Component	`Compiler::bind_backend`
Network transport	Host + adapter	Wire envelope
Identity / authentication	Transport adapter	Below the envelope
Gradient-poisoning defense	Aggregator Component	`Compiler::bind_aggregator`
Differential privacy	Codec pair	`Compiler::bind_codec`
Peer attestation	PeerSelector	`Compiler::bind_peer_selector`
NAT traversal	Transport + Protocol bootstrap	libp2p adapter + `Module::bootstrap`
Adaptive deadlines	Framework runtime	`with_per_hop_budget_ns` + `RttTracker`

8 Related Work

Backends as enablers. TFLite, ExecuTorch, Core ML, ONNX Runtime, and Burn are inference and training backends that integrate as Backend Components. They are enablers of the substrate, not competitors with it; the reference CpuBackend demonstrates the integration shape, and the same shape applies to accelerator and platform-specific runtimes.

Ray and PyTorch. Ray [7] provides actor-based distributed execution with a global object store, designed for homogeneous clusters with low-latency interconnect. PyTorch [13] provides the model-authoring DSL. Their combination is the canonical reference for distributed training inside a data center, with strong support from a single operator running a homogeneous fleet. The combination is unsuitable for heterogeneous, WAN-scale deployments: actors carry runtime weight, the object store assumes shared infrastructure, and placement is built around scheduling decisions rather than overlay topology. BytesAndBrains targets these gaps.

DeepSpeed, Megatron-LM, and PyTorch FSDP. These three systems address a layer of the stack distinct from the substrate. DeepSpeed's ZeRO [31] partitions optimizer states, gradients, and parameters across data-parallel workers to enable training models with parameter counts that exceed single-device memory. Megatron-LM [32] specializes in tensor and pipeline parallelism for transformer-style architectures. PyTorch FSDP [33] shards model parameters across data-parallel workers and gathers them on demand. All three assume a single trust domain, synchronous collective operations, and a low-latency interconnect (NVLink, InfiniBand, or equivalent). They are appropriate when the goal is to train a single large model as quickly as possible inside a homogeneous cluster. BytesAndBrains does not compete with them; it targets workloads where the assumptions of homogeneous hardware, low-latency interconnect, and single-trust-domain operation do not hold.

Flower. Flower [6] is a federated learning framework supporting custom strategies, with PyTorch and TensorFlow integration and a substantial set of supported aggregation algorithms. It is not a substrate. Flower selects one paradigm (federated learning with a coordinator) and ships strategies within that paradigm. Implementing gossip learning on top of Flower runs against the grain of its API; implementing peer-to-peer inference falls outside Flower's scope. For workloads bounded by the federated learning paradigm, Flower offers a more direct path than BytesAndBrains. For workloads composing across paradigms, BytesAndBrains provides the substrate Flower is not.

TensorFlow Federated. TensorFlow Federated [10] is Google's production federated learning stack. It exposes a high-level Federated Learning API and a Federated Core for expressing federated computations. Like Flower, it is single-paradigm: a coordinator-driven federated computation model with TensorFlow as the bound tensor backend. BytesAndBrains sits at a layer that accepts TFF-style federated rounds as one composition among others.

Hivemind. Hivemind [11] is a peer-to-peer distributed training library for PyTorch, with decentralized averaging, distributed mixture-of-experts, and DHT-coordinated training. It is paradigm-specific (PyTorch-bound, training-focused) and supplies its own RPC and DHT stack. BytesAndBrains sits at a layer that hosts Hivemind-style decentralized training as a composition: PeerSelector and Aggregator roles plus a custom protocol Component express the same coordination shape over the substrate's envelope, with the backend choice unbound.

libp2p. libp2p [9] provides transports, multiplexing, DHT-style discovery, and a peer-identity model. It does not provide an authoring DSL, an IR, a compiler, or a dispatch model. Building distributed machine learning on bare libp2p is a substantial engineering effort. BytesAndBrains treats libp2p as one valid transport adapter beneath the envelope, not as a competing layer.

Spark. Spark [12] solves an adjacent problem: bulk-synchronous data-parallel computation on a cluster, with a high-level functional API and a runtime that handles partitioning and shuffles. It is not designed for streaming workloads, heterogeneous fleets, or stateful protocols with control-plane traffic (gossip exchanges, peer sampling, leader election). The planning layers differ: BytesAndBrains plans against a typed graph with explicit cross-node edges, Spark against a dataset abstraction with bulk-synchronous stages. The two systems answer different shapes of question.

Substra. Substra [8] is a federated learning platform targeting regulated industries. It provides orchestration, asset management, audit trails, and coordinator-driven workflows. It is a platform with workflow primitives, not a runtime with authoring primitives. BytesAndBrains and Substra occupy different layers and could coexist: BytesAndBrains expressing the per-node computation Substra orchestrates.

In summary, BytesAndBrains occupies a layer of the stack not addressed by existing systems. It is a Rust substrate rather than a Python platform, sans-IO rather than actor-based, and owns the byte layer between nodes rather than the scheduling layer above them. It treats federated, gossip, split, and peer-to-peer strategies as placement-and-binding problems on a single substrate, rather than as distinct paradigms with their own coordinators and runtimes.

9 Conclusions

This paper has presented BytesAndBrains, the substrate for networked machine learning. It records a user's computation into an ONNX-backed [4] intermediate representation, partitions the recording across nodes through the compiler, executes each partition as a state machine, and ships bytes between participants through a single structured envelope. The runtime is sans-IO [5], single-threaded in dispatch within each node yet free in execution, and decoupled from any particular transport. Above it, the field's strategies compose as Components satisfying Role contracts. Below it, any transport that can ship bytes can carry the envelopes. BytesAndBrains occupies a layer of the stack not addressed by existing federated learning frameworks, distributed compute frameworks, or peer-to-peer overlays.

The framework compounds with composition: the more Backends, Codecs, Aggregators, PeerSelectors, and Protocols ship, the more strategies become expressible on it. The source repository at https://github.com/Bytes-Brains/bytesandbrains is the current state.

References

[1] H. Brendan McMahan et al. "Communication-Efficient Learning of Deep Networks from Decentralized Data". In: *Proceedings of Artificial Intelligence and Statistics (AISTATS)*. 2017. URL: https://arxiv.org/abs/1602.05629.

[2] István Hegedűs, Gábor Danner, and Márk Jelasity. "Gossip Learning as a Decentralized Alternative to Federated Learning". In: *Distributed Applications and Interoperable Systems (DAIS)*. 2019. URL: https://arxiv.org/abs/1906.05882.

[3] Praneeth Vepakomma et al. *Split learning for health: Distributed deep learning without sharing raw patient data*. 2018. URL: https://arxiv.org/abs/1812.00564.

[4] ONNX. *Open Neural Network Exchange specification*. URL: https://onnx.ai/onnx/.

[5] Sans-IO. *Network protocols, sans I/O*. URL: https://sans-io.readthedocs.io/.

[6] Daniel J. Beutel et al. *Flower: A Friendly Federated Learning Framework*. 2020. URL: https://arxiv.org/abs/2007.14390.

[7] Philipp Moritz et al. "Ray: A Distributed Framework for Emerging AI Applications". In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2018. URL: https://arxiv.org/abs/1712.05889.

[8] Mathieu N. Galtier and Camille Marini. *Substra: a framework for privacy-preserving, traceable and collaborative Machine Learning*. 2019. URL: https://arxiv.org/abs/1910.11567.

[9] libp2p. *A modular network stack*. URL: https://libp2p.io/.

[10] TensorFlow Federated. *An open-source framework for machine learning and other computations on decentralized data*. URL: https://www.tensorflow.org/federated.

[11] Hivemind. *Decentralized deep learning in PyTorch*. URL: https://github.com/learning-at-home/hivemind.

[12] Matei Zaharia et al. "Apache Spark: A Unified Engine for Big Data Processing". In: *Communications of the ACM* 59.11 (2016), pp. 56–65. DOI: 10.1145/2934664. URL: https://doi.org/10.1145/2934664.

[13] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2019. URL: https://arxiv.org/abs/1912.01703.

[14] Martín Abadi et al. "TensorFlow: A System for Large-Scale Machine Learning". In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2016. URL: https://arxiv.org/abs/1605.08695.

[15] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. 2018. URL: https://github.com/google/jax.

[16] David Kempe, Alin Dobra, and Johannes Gehrke. "Gossip-based computation of aggregate information". In: *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 2003, pp. 482–491. DOI: 10.1109/SFCS.2003.1238221. URL: https://doi.org/10.1109/SFCS.2003.1238221.

[17] Petar Maymounkov and David Mazières. "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric". In: *International Workshop on Peer-to-Peer Systems (IPTPS)*. 2002, pp. 53–65. DOI: 10.1007/3-540-45748-8_5. URL: https://doi.org/10.1007/3-540-45748-8_5.

[18] Jakub Konečný et al. *Federated Learning: Strategies for Improving Communication Efficiency*. 2017. URL: https://arxiv.org/abs/1610.05492.

[19] Keith Bonawitz et al. "Towards Federated Learning at Scale: System Design". In: *Proceedings of Machine Learning and Systems (SysML)*. 2019. URL: https://arxiv.org/abs/1902.01046.

[20] Naohiro Hayashibara et al. "The φ Accrual Failure Detector". In: *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS)*. 2004. URL: https://dl.acm.org/doi/10.1109/RELDIS.2004.1353004.

[21] Peter Kairouz et al. "Advances and Open Problems in Federated Learning". In: *Foundations and Trends in Machine Learning* 14.1–2 (2021). URL: https://arxiv.org/abs/1912.04977.

[22] Jeffrey Dean and Luiz André Barroso. "The Tail at Scale". In: *Communications of the ACM* 56.2 (2013), pp. 74–80. URL: https://research.google/pubs/pub40801/.

[23] Benjamin H. Sigelman et al. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Tech. rep. Google Research, 2010. URL: https://research.google/pubs/pub36356/.

[24] Phil Karn and Craig Partridge. "Improving Round-Trip Time Estimates in Reliable Transport Protocols". In: *Proceedings of ACM SIGCOMM*. 1987. DOI: 10.1145/55482.55484. URL: https://dl.acm.org/doi/10.1145/55482.55484.

[25] Vern Paxson et al. *Computing TCP's Retransmission Timer*. 2011. URL: https://datatracker.ietf.org/doc/html/rfc6298.

[26] Matt Welsh, David Culler, and Eric Brewer. "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services". In: *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*. 2001. URL: https://www.sosp.org/2001/papers/welsh.pdf.

[27] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. "T-Man: Gossip-based fast overlay topology construction". In: *Computer Networks* 53.13 (2009), pp. 2321–2339. DOI: 10.1016/j.comnet.2009.03.013. URL: https://doi.org/10.1016/j.comnet.2009.03.013.

[28] Dong Yin et al. "Byzantine-Robust Distributed Learning: Towards Optimal Statistical Rates". In: *Proceedings of the 35th International Conference on Machine Learning (ICML)*. 2018. URL: https://arxiv.org/abs/1803.01498.

[29] Reza Shokri et al. "Membership Inference Attacks Against Machine Learning Models". In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2017. URL: https://arxiv.org/abs/1610.05820.

[30] Martín Abadi et al. "Deep Learning with Differential Privacy". In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2016. URL: https://arxiv.org/abs/1607.00133.

[31] Samyam Rajbhandari et al. “ZeRO: Memory Optimizations Toward Training Trillion Parameter Models”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 2020. URL: https://arxiv.org/abs/1910.02054.

[32] Mohammad Shoeybi et al. *Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism*. 2019. URL: https://arxiv.org/abs/1909.08053.

[33] Yanli Zhao et al. “PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel”. In: *Proceedings of the VLDB Endowment* 16.12 (2023), pp. 3848–3860. URL: https://arxiv.org/abs/2304.11277.