Heterogeneous Distributed Training with Planetary Compute Protocol

Phil Rohr, Thomas Bach, Ahmet Celebi, Atabey Ünlü

January 12, 2026

Abstract

Training large-scale foundation models on heterogeneous, geodistributed hardware requires a fundamental decoupling of model semantics from runtime execution. Standard distributed training paradigms implicitly rely on homogeneous clusters with low-latency interconnects and tightly coupled runtime environments. We present the Planetary Compute Protocol (PCP), a system for fault-tolerant, low-communication training across diverse hardware accelerators. PCP compiles the entire training step—including forward propagation, reverse-mode automatic differentiation, and optimizer state transitions—into a standalone, immutable Virtual Machine FlatBuffer (VMFB) artifact via IREE and StableHLO. This approach eliminates the dependency on heavyweight framework runtimes (e.g., PyTorch) on worker nodes, reducing workers to generic tensor execution environments. We implement a hierarchical supervision tree where a Shepherd coordinates global state via a custom TCP/Cap'n Proto protocol, while local Supervisors manage ephemeral Worker processes to isolate hardware faults. By integrating DiLoCo-style federated optimization with this compiler-centric architecture, PCP enables reliable training over public internet connections using heterogeneous compute endpoints (NVIDIA CUDA, AMD ROCm, Apple Metal) without semantic drift.

1 Introduction

Modern large-scale training pipelines are increasingly limited not by raw FLOPs, but by the system's assumptions baked into standard distributed optimization. Synchronous data parallelism—frequent all-reduce of gradients across workers—implicitly requires low-latency, high-bandwidth interconnects and stable cluster membership. These conditions are mismatched with today's compute reality: heterogeneous accelerators, preemptible capacity, and geographically distributed "islands" of hardware connected by high-latency links. A growing line of work therefore targets low-communication training, where workers perform many local steps and synchronize only intermittently. DiLoCo exemplifies this approach: it amortizes communication over an inner loop (typically AdamW) and performs a momentum-based outer update, substantially reducing synchronization frequency while preserving model quality and robustness to churn [1, 2]. Recent results strengthen this perspective, including scaling-law evidence that DiLoCo-style training remains reliable as model size increases [3], and a set of complementary algorithmic variants that further reduce bandwidth via sparsification, alternative inner optimizers, or avoiding all-reduce altogether [4, 5, 6, 7].

However, communication efficiency is only part of the problem. In most deep learning stacks, training semantics (forward execution, differentiation, optimizer updates, and state management) are realized by a heavyweight host runtime. This runtime-centric design complicates portability across heterogeneous backends and makes distributed systems tightly coupled to framework-specific control logic. Compiler infrastructures such as MLIR motivate a different decomposition: represent models and transformations in a reusable IR, then lower systematically to diverse hardware targets [8]. In parallel, compiler- and source-transformation approaches to automatic differentiation show that AD can be expressed as a static program transformation rather than a runtime service, enabling differentiated programs to be optimized and lowered like ordinary code [9, 10]. More recent MLIR work expands the space of transformation and code-generation workflows, reinforcing the viability of IR-driven, backend-agnostic pipelines [11].

We introduce the Planetary Compute Protocol (PCP), a graph-to-graph training system that compiles a full training step—forward pass, reverse-mode differentiation, and optimizer update—into immutable executables distributed to workers, visualized in Figure 1. PCP separates model specification from execution by using an MLIR-based canonical representation (via StableHLO, which provides serialization/versioning compatibility guarantees) and by performing differentiation and optimizer embedding at compile time rather than in a host runtime [12, 8, 10]. PCP then pairs these compiled artifacts with a fault-tolerant orchestration layer intended for unreliable, internet-scale settings, aligning system design with the algorithmic premise of infrequent synchronization [1, 3].

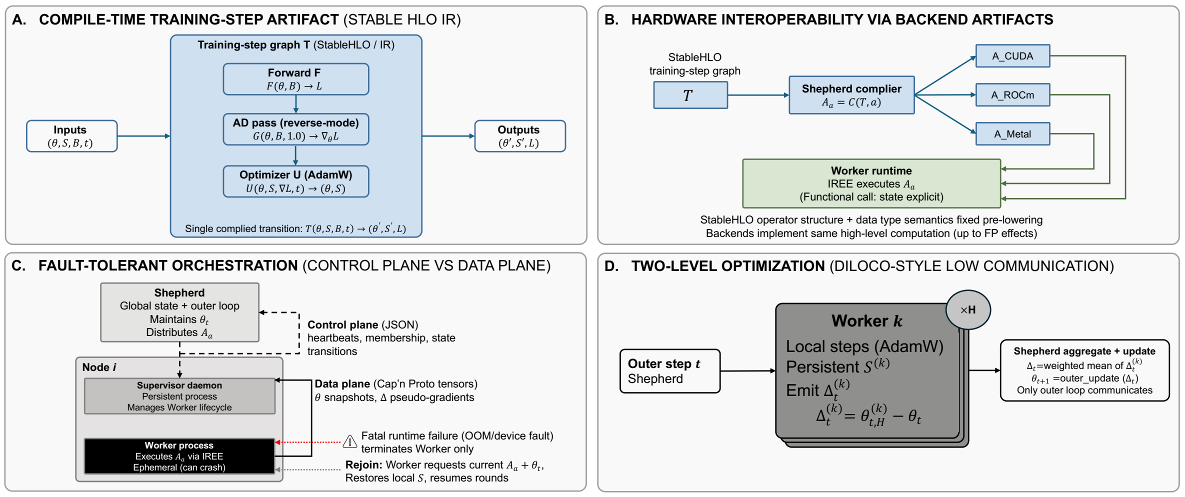
A. COMPILE-TIME TRAINING-STEP ARTIFACT (STABLE HLO IR)
Training-step graph T (StableHLO / IR)
Forward F
$F(\theta, B) \to L$
AD pass (reverse-mode)
$G(\theta, B, 1.0) \to \nabla_\theta L$
Optimizer U (AdamW)
$U(\theta, S, \nabla L, t) \to (\theta, S)$
Inputs
(θ, S, B, t)
Outputs
(θ', S', L)
Single complied transition: $T(\theta, S, B, t) \to (\theta', S', L)$
B. HARDWARE INTEROPERABILITY VIA BACKEND ARTIFACTS
StableHLO training-step graph
T
Shepherd complier
$A_a = C(T, a)$
A_CUDA
A_ROCm
A_Metal
Worker runtime
IREE executes A_a
(Functional call: state explicit)
StableHLO operator structure + data type semantics fixed pre-lowering
Backends implement same high-level computation (up to FP effects)
C. FAULT-TOLERANT ORCHESTRATION (CONTROL PLANE VS DATA PLANE)
Shepherd
Global state + outer loop
Maintains θ_t
Distributes A_a
Control plane (JSON)
heartbeats, membership, state transitions
Node i
Supervisor daemon
Persistent process
Manages Worker lifecycle
Data plane (Cap'n Proto tensors)
θ snapshots, Δ pseudo-gradients
Worker process
Executes A_a via IREE
Ephemeral (can crash)
Fatal runtime failure (OOM/device fault) terminates Worker only
Rejoin: Worker requests current $A_a + \theta_t$,
Restores local S, resumes rounds
D. TWO-LEVEL OPTIMIZATION (DILOCO-STYLE LOW COMMUNICATION)
Outer step t
Shepherd
Worker k
×H
Local steps (AdamW)
Persistent $S^{(k)}$
Emit $\Delta_t^{(k)}$
$\Delta_t^{(k)} = \theta_{t,H}^{(k)} - \theta_t$
Shepherd aggregate + update
Δ_t=weighted mean of $\Delta_t^{(k)}$
θ_{t+1} =outer_update (Δ_t)
Only outer loop communicates

Figure 1: **Figure 1: Overview of the PCP (Portable Compiled Protocols) Framework.** (A) **Compile-time training-step artifacts:** The training logic is lowered into a unified StableHLO intermediate representation (IR). A single transition T encapsulates the forward pass, reverse-mode automatic differentiation, and optimizer state updates (e.g., AdamW). (B) **Hardware interoperability:** The Shepherd compiler lowers the StableHLO graph into backend-specific VMFB artifacts (CUDA, ROCm, Metal). These artifacts are executed by the IREE runtime, maintaining functional consistency across heterogeneous hardware. (C) **Fault-tolerant orchestration:** The system decouples the control plane (asynchronous heartbeats and lifecycle management via JSON) from the data plane (high-throughput tensor exchange via Cap'n Proto). A persistent Supervisor daemon manages the lifecycle of ephemeral Worker processes, allowing for rapid recovery from device-level or OOM failures without interrupting the global state. (D) **Two-level optimization:** To minimize communication overhead, the system utilizes a DiLoCo-style orchestration. Workers perform H local optimization steps on their respective data shards before communicating only the weight deltas (Δ_t) for global aggregation and outer-loop momentum updates at the Shepherd level.

Contributions. (1) A compiler-centric training-step artifact that fixes training semantics prior to backend lowering, improving portability across heterogeneous devices; (2) a fault-tolerant orchestration design that decouples control-plane reliability from data-plane execution; and (3) an integration of DiLoCo-style low-communication optimization with statically compiled training steps, reducing dependence on framework runtimes while preserving consistent update semantics [1, 10].

2 Method

The Planetary Compute Protocol (PCP) uses a graph-to-graph compilation pipeline that separates model specification from hardware execution. Input models, provided as MLIR source files [12, 8], serve as the canonical form for subsequent transformations. PCP then constructs a single training-step graph that explicitly encodes **(i)** forward propagation, **(ii)** reverse-mode automatic differentiation, and **(iii)** optimizer updates (e.g., AdamW). This unified graph is Just-in-Time (JIT) compiled with IREE into a Virtual Machine FlatBuffer (VMFB) executable [13]. By distributing an immutable training-step artifact per backend, PCP standardizes training semantics across heterogeneous runtimes (e.g., NVIDIA

CUDA, AMD ROCm, Apple Metal), improving numerical comparability by fixing operator structure prior to hardware-specific lowering.

PCP coordinates distributed execution using a hierarchical Shepherd–Supervisor–Worker architecture designed for fault tolerance and efficient communication. The Shepherd maintains global model parameters and distributes the compiled VMFB artifacts to participating nodes. Workers execute the artifact against their local hardware abstraction layer and persist the optimizer state required by inner-loop updates (e.g., first and second moments for AdamW [2]), as well as any DiLoCo-specific [1] local state. Each node also runs a Supervisor process that isolates control-plane responsibilities from the data plane, enabling failure detection and automatic Worker respawning without terminating the global job. Synchronization between the Shepherd and Workers uses a lightweight TCP protocol with Cap'n Proto serialization to exchange pseudo-gradients efficiently over high-latency public networks.

2.1 Graph-to-Graph Compilation Pipeline

PCP enforces a strict separation between model specification and execution through a multi-stage graph compilation pipeline. Unlike eager-execution frameworks, where differentiation and optimizer updates are orchestrated by a host runtime (e.g., Python), PCP constructs a single, stateful training-step graph in IR before hardware lowering. Concretely, an inference-only forward graph is transformed into a trainable artifact by explicitly representing both differentiation and parameter updates as IR subgraphs.

Let θ denote model parameters, S denote persistent optimizer state, t denote the current timestep, and B denote a minibatch. The forward computation F maps (θ, B) to a scalar loss:

$$F(\theta, B) \to L \tag{1}$$

Given the forward graph, PCP constructs a gradient graph G that computes parameter gradients via reverse-mode automatic differentiation, seeded with an initial gradient of 1.0:

$$G(\theta, B, 1.0) \to \nabla_\theta L \tag{2}$$

PCP then embeds the optimizer update rule U (e.g., AdamW) as explicit IR operations operating on $(\theta, S, \nabla_\theta L, t)$, utilizing the timestep for bias correction to produce updated parameters and state:

$$U(\theta, S, \nabla_\theta L, t) \to (\theta, S) \tag{3}$$

These components are composed into a unified training-step transition function:

$$T(\theta, S, B, t) \to (\theta, S, L) \tag{4}$$

This yields a self-contained artifact that consumes the current training state, a minibatch, and the scalar timestep, and returns the updated state and loss, without requiring an external runtime to schedule differentiation or apply updates.

Prior to lowering, the system applies a stabilization pass (Sanitizer) to the IR. This pass injects ϵ-guards into logarithmic and division operations and normalizes floating-point literals (e.g., standardizing `NaN` representations), ensuring consistent numerical behavior across heterogeneous backends. Supplementary Algorithm 2 details the graph-to-graph construction procedure that transforms a forward StableHLO module into a single training-step artifact by composing reverse-mode AD and optimizer state transitions prior to lowering.

2.2 Reverse-Mode Automatic Differentiation on StableHLO

PCP avoids reliance on runtime autograd by implementing automatic differentiation (AD) as a compile-time transformation pass over StableHLO. Given a forward graph G_{fwd}, the AD pass constructs a backward graph G_{bwd} by **(i)** cloning the necessary forward operations into the gradient scope to provide primal values, **(ii)** exposing the incoming loss gradient as a function argument (allowing the orchestrator to seed it, typically with 1.0), and **(iii)** traversing operations in reverse topological order. For each operation

$$y = f(x_1, \ldots, x_n), \tag{5}$$

the pass applies a statically registered vector–Jacobian product (VJP) rule to produce adjoints $\bar{x}_i$ from $\bar{y}$.

To handle fan-out, the compiler accumulates gradient contributions for each SSA value and emits explicit elementwise sums when multiple consumers generate distinct adjoint terms, enforcing the multivariable chain rule. To handle broadcasting and shape-changing operators, the compiler normalizes adjoints by synthesizing the required reductions (e.g., summation over broadcast dimensions), ensuring that each adjoint tensor matches the shape of its corresponding primal value. To minimize peak memory usage on consumer-grade GPUs, the compiler automatically injects activation rematerialization for high-rank tensors during the backward pass generation. Pseudocode for PCP's reverse-mode AD transformation, including adjoint accumulation and broadcast/shape normalization, is given in Supplementary Algorithm 3.

2.3 Stateless Hardware-Agnostic Execution

PCP achieves hardware interoperability by decoupling training semantics from device-specific execution through a standardized compilation artifact. For each backend $a \in A$ (e.g., $\{\mathrm{CUDA}, \mathrm{ROCm}, \mathrm{Metal}\}$) present in the active worker pool, the Shepherd compiles the StableHLO training-step graph T into a backend-specific VMFB artifact:

$$A_a = C(T, a) \tag{6}$$

Each A_a packages lowered device code and an execution schedule for the corresponding runtime. Because operator structure and dtype semantics are fixed at the StableHLO level prior to lowering, backends are constrained to implement the same high-level computation up to floating-point effects.

Workers execute A_a via the IREE runtime in a functional style: state is explicit in the call interface rather than implicit in a framework runtime. At step t, a Worker invokes A_a with (θ_t, S_t, B_t) and receives $(\theta_{t+1}, S_{t+1}, \ell_t)$. The Worker binary therefore remains a generic tensor host: it embeds no domain-specific logic for layers, differentiation, or optimizer behavior beyond executing the received artifact. Changing training behavior (e.g., replacing AdamW with Lion) reduces to distributing an updated graph and corresponding artifacts, without modifying Worker executables.

2.4 Fault-Tolerant Distributed Orchestration

Edge environments exhibit frequent, independent node failures, so PCP treats orchestration as a first-class component of training. PCP uses a hierarchical Shepherd–Supervisor–Worker design that separates the control plane from the data plane. The Shepherd coordinates global state (e.g., the DiLoCo outer loop), maintains the current global parameters θ_t, and distributes the backend-specific training artifact A_a to each node. On every node, a persistent Supervisor daemon manages the lifecycle of an ephemeral Worker process that executes the compiled training step. This indirection ensures that fatal runtime failures (e.g., CUDA out-of-memory or device-level faults) terminate only the Worker, while the Supervisor—and its session with the Shepherd—remains intact.

Communication uses a framed TCP protocol with two logical classes of messages. Control-plane messages (e.g., heartbeats, handshakes, membership changes, and state transitions) are encoded as lightweight JSON to simplify debugging and introspection. Data-plane payloads carrying tensors—such as parameter snapshots θ_t and pseudo-gradients Δ_t—are serialized with Cap'n Proto to avoid per-field parsing overhead and to support efficient deserialization. In implementations where control and data share a single JSON envelope, the tensor payload is transported as an opaque byte buffer (e.g., Base64-encoded) and decoded only at the aggregation boundary, keeping intermediate routing and bookkeeping independent of tensor layout.

Fault tolerance is implemented as a reactive recovery loop driven by the Supervisor. At startup, the Supervisor establishes a persistent connection to the Shepherd, spawns the Worker, and monitors it as a child process. If the Worker exits unexpectedly, the Supervisor records the failure, applies a bounded backoff, and respawns a fresh Worker. Rejoining is lightweight: the respawned Worker requests the current artifact A_a and global parameters θ_t, restores any required local state (e.g., optimizer state S from durable storage or a known initialization), and resumes participation in subsequent aggregation rounds. As a result, node crashes reduce throughput temporarily rather than corrupting global progress, enabling PCP to tolerate unreliable hardware and preemptible/spot capacity without terminating the training job. A complete end-to-end pseudocode description of the Shepherd–Supervisor–Worker protocol, including artifact distribution, worker lifecycle management, and DiLoCo-style synchronization, is provided in Supplementary Algorithm 1.

2.5 Distributed Optimization Algorithm - DiLoco

PCP's distributed training procedure adapts DiLoCo-style low-communication optimization to a statically compiled, runtime-minimal execution model. In contrast to synchronous data parallelism—where Workers exchange gradients every step—PCP separates local optimization from global synchronization. Each Worker performs H inner optimization steps independently and communicates only once per outer iteration, reducing synchronization frequency and making training practical over high-latency, geo-distributed links.

Let θ_t denote the global parameters at outer step t. The system orchestrates K Workers indexed by $k \in \{1, \ldots, K\}$. At the start of each outer step, the Shepherd broadcasts θ_t and each Worker initializes its local parameters to the global state:

$$\theta_{t,0}^{(k)} \leftarrow \theta_t \tag{7}$$

Each Worker also maintains persistent local optimizer state $S^{(k)}$ (e.g., AdamW moments), which is not reset across outer iterations.

Inner loop (local). For inner step $h \in \{0, \ldots, H-1\}$, to handle worker churn, data is not statically sharded. Instead, the Shepherd dynamically assigns a dataset offset and length to Worker k at the start of the round. Worker k streams this assigned chunk locally to form minibatch $B_{t,h}^{(k)}$ and computes a stochastic gradient:

$$g_{t,h}^{(k)} = \nabla_\theta L\Big(\theta_{t,h}^{(k)};\, B_{t,h}^{(k)}\Big) \tag{8}$$

Parameters are updated by applying H steps of AdamW using the persistent state $S^{(k)}$:

$$\Big(\theta_{t,h+1}^{(k)},\, S_{t,h+1}^{(k)}\Big) \leftarrow \mathrm{AdamW}\Big(\theta_{t,h}^{(k)},\, S_{t,h}^{(k)},\, g_{t,h}^{(k)};\, \eta_{\text{inner}},\, \lambda\Big) \tag{9}$$

where η_{inner} is the inner learning rate and λ is the weight decay coefficient. Persisting $S^{(k)}$ across communication rounds preserves local momentum/curvature information and avoids repeatedly "cold-starting" the optimizer each outer step.

Outer loop (communication + aggregation). After H inner steps, Worker k calculates the accumulated pseudo-gradient as the difference between the initial global parameters and the final local state:

$$\Delta_t^{(k)} = \theta_t - \theta_{t,H}^{(k)} \tag{10}$$

(Using this sign convention, $\Delta_t^{(k)}$ represents the aggregated gradient step required to produce the local update.) Workers transmit $\Delta_t^{(k)}$ to the Shepherd, which aggregates them (e.g., a weighted mean by local batch count w_k):

$$\Delta_t = \frac{\sum_{k=1}^{K} w_k\, \Delta_t^{(k)}}{\sum_{k=1}^{K} w_k} \tag{11}$$

The Shepherd then performs an outer update with Nesterov momentum. Let v_t be the global velocity, μ the momentum coefficient, and η_{outer} the outer step size. One standard Nesterov form is:

$$v_{t+1} \leftarrow \mu v_t + \Delta_t, \qquad \theta_{t+1} \leftarrow \theta_t + \eta_{\text{outer}}\,(\mu v_{t+1} + \Delta_t) \tag{12}$$

This two-level procedure treats the Workers' independent inner-loop trajectories as noisy but informative descent directions that can be combined with infrequent communication. In PCP, the Worker's inner-loop computation is executed via the compiled training-step artifact (Fault-Tolerant Distributed Orchestration Section), ensuring that the same update semantics are applied consistently across heterogeneous backends; only the outer-loop aggregation and broadcast couple Workers through the network. Supplementary Algorithm 1 provides full pseudocode for the combined orchestration and optimization loop (Shepherd outer updates with Worker inner steps and pseudo-gradient aggregation).

3 Evaluation and Analytical Validation

3.1 Scope and evaluation philosophy

This report primarily presents a systems design and implementation approach rather than a fully benchmarked training study. Because sustained cluster time was not available within the report timeline, we do not claim competitive end-to-end throughput or long-horizon convergence against mature baselines. Instead, we provide (i) an analytical characterization of communication costs and scaling behavior, and (ii) a set of falsifiable validation checks that establish whether the compiled training-step artifact and orchestration protocol behave as specified.

Concretely, we focus on four properties that can be validated without long training runs: **(1) semantic closure of the training step** (the transition $T(\theta, S, B, t) \to (\theta', S', L)$ is fully contained in the artifact interface), **(2) backend portability boundary** (what is fixed at StableHLO vs what may vary numerically after lowering), **(3) fault-isolation semantics** (Supervisor contains worker crashes without global restart), and **(4) communication scaling** as a function of model size and local inner-loop length H.

NanoChat workload (model and training configuration). For a lightweight end-to-end validation workload, we use the NanoChat setup: a GPT-2–style decoder-only Transformer with 8 layers and embedding dimension 1024, totaling 92,274,688 trainable parameters. Unless stated otherwise, experiments use a global batch size of 8 sequences and a context length of 2048 tokens. This configuration is large enough to exercise the full training-step pipeline (forward pass, reverse-mode AD, and optimizer state updates) while remaining tractable for short-horizon systems validation on the available hardware.

3.2 Short-horizon loss decrease with NanoChat on 8×H100

To provide a minimal end-to-end learning signal under limited evaluation time, we ran a short PCP distributed training sequence of the NanoChat workload on a homogeneous 8×H100 setup and logged the average loss per completed round. Over 15 rounds, the average loss decreases from 11.0623 to 10.5599 (an absolute reduction of 0.5024, i.e., 4.54% relative). The loss decreases on 13 of the 14 round-to-round transitions, with a single small increase (+0.0011) consistent with stochastic minibatch variation.

This short-horizon result in Figure 2 supports the systems-level claim that PCP executes the compiled training-step transition repeatedly across multiple workers and applies optimizer updates without numerical instability. It is intended as a sanity check of training dynamics rather than evidence of long-horizon convergence or throughput competitiveness against mature baselines.

3.3 Analytical communication profile under DiLoCo-style synchronization

PCP communicates intermittently: Workers run H local inner steps and then transmit an update once per outer round. Let P be the number of parameters and b the bytes per parameter (e.g., b=2 for fp16/bf16, b=4 for fp32). If each outer round transmits a full parameter snapshot θ_t (broadcast) and a full pseudo-gradient Δ_t (upload), then the dominant data-plane payload per worker per round is:

$$\text{bytes/round/worker} \approx 2Pb + \text{overhead}. \tag{13}$$

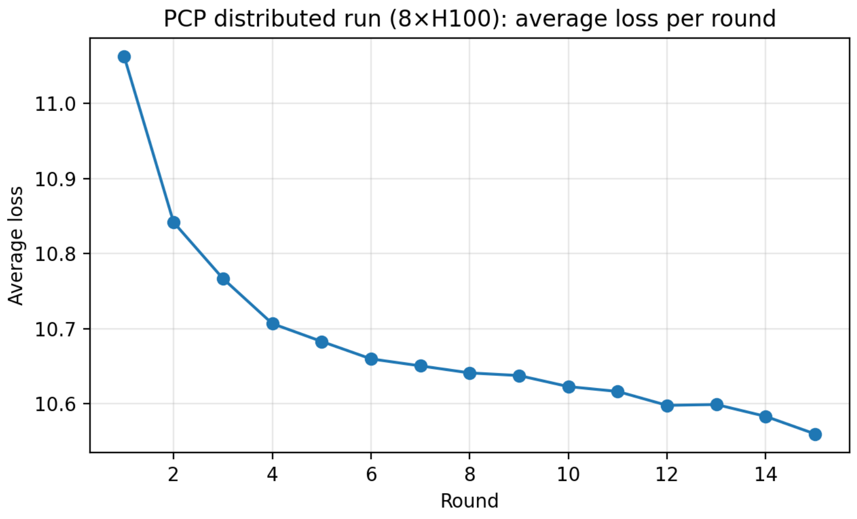

Figure 2: NanoChat short-horizon loss trace under PCP on 8×H100. Average loss decreases from 11.0623 to 10.5599 over 15 rounds (4.54% relative reduction), with one minor upward fluctuation consistent with stochasticity.

Amortized per local inner step, this yields:

$$\text{bytes/local-step/worker} \approx \frac{2Pb}{H}. \tag{14}$$

This makes the core scaling claim explicit and falsifiable: for fixed (P, b), increasing H reduces network traffic per unit of local compute approximately as $1/H$, while preserving a simple $O(P)$ message size per outer round.

For reference, synchronous data-parallel training (e.g., gradient all-reduce each step) communicates $O(Pb)$ per step (up to topology-dependent constants), implying that PCP reduces communication per local step by approximately a factor of H in regimes where full-state synchronization is used only once per outer round.

3.4 Training-step semantic closure and correctness obligations

A key design goal of PCP is that training semantics are *closed* within the compiled artifact boundary. The Worker executes a single function with explicit state:

$$T(\theta, S, B, t) \rightarrow (\theta', S', L),$$

so correctness reduces to verifying that (i) state is updated only through this transition, and (ii) the transition implements the intended forward, reverse-mode AD, and optimizer logic.

Because this report does not include a full empirical evaluation, we specify the minimal correctness checks required to validate the artifact in a reproducible manner:

- **Interface invariants:** shapes/dtypes of (θ, S) remain consistent across steps; loss L is finite; timestep t increments monotonically.
- **Transition determinism under fixed inputs:** for a fixed backend, fixed seed, and identical minibatches, repeated executions produce identical outputs up to floating-point nondeterminism sources (e.g., non-deterministic kernels).
- **Reference equivalence (recommended):** compare one-step outputs (θ', S', L) against a reference implementation of the same StableHLO graph (or a framework reference) on the same inputs; report relative error tolerances.
- **Gradient sanity (recommended):** finite-difference checks on a small model to validate the AD pass on a sampled subset of parameters.

These checks are straightforward to run as unit/integration tests and directly falsify semantic errors in the artifact construction pipeline.

3.5 Cross-backend portability boundary

PCP fixes operator structure, differentiation logic, and optimizer updates at the StableHLO level before lowering. This provides a clean portability boundary: any backend-specific differences should arise only from (i) floating-point implementation details, (ii) lowering choices (fusion, scheduling), and (iii) nondeterministic kernel behavior, not from divergent high-level semantics.

To make this claim falsifiable, the appropriate cross-backend validation protocol is:

- run identical batches and seeds for N steps on two backends;
- compare the loss sequence $\{L_t\}$ and update norms $\{\|\Delta\theta_t\|\}$;
- report max and median relative deviations, and parameter drift $\|\theta_t^a - \theta_t^b\|/\|\theta_t^a\|$ at the final step.

This report does not claim specific tolerances; rather, it provides the measurement procedure needed to quantify portability for a given model/dtype regime.

3.6 Fault isolation semantics and measurable recovery KPIs

PCP uses a Supervisor process to isolate faults to ephemeral Worker processes. The scientific claim here is not that failures never occur, but that their *effect is bounded*: a worker crash should not require global restart, and the system should return to a stable training state once the Worker is respawned and resynchronized.

This yields measurable KPIs for future evaluation:

- **detection latency:** time from Worker failure to Supervisor detection;
- **time-to-rejoin:** failure $\rightarrow$ Worker respawn $\rightarrow$ artifact fetch $\rightarrow$ parameter sync $\rightarrow$ resumed participation;
- **progress impact:** fraction of rounds missed and throughput degradation during recovery;
- **safety invariant:** global parameters θ_t advance monotonically with no rollback or corruption due to partial failures.

We report recovery time as the interval from worker exit to rejoin: median $\approx$5s, p95 $\approx$6s, measured over 8 restarts.

Summary. In the absence of long-horizon empirical results, we provide an analytical communication model and a concrete validation protocol that makes PCP's claims measurable: semantic closure at the artifact boundary, portability checks across backends, and fault-isolation semantics with well-defined recovery KPIs.

4 Discussion and Conclusion

This work introduces the Planetary Compute Protocol (PCP), a compiler-centric approach to distributed training that separates training semantics from worker runtimes. PCP constructs a single stateful transition $T(\theta, S, B, t) \rightarrow (\theta', S', L)$ by embedding reverse-mode automatic differentiation and optimizer updates directly into StableHLO, then lowers this transition into immutable, backend-specific VMFB artifacts via IREE. Execution is orchestrated by a Shepherd–Supervisor–Worker hierarchy in which workers act as generic tensor execution hosts (no framework autograd/optimizer logic) and Supervisors respawn ephemeral workers on failures without restarting the global job. Within the limited cluster time available, the experiments primarily validate system invariants: repeated execution of the compiled training step, stable state updates without NaNs/Infs, portability across CUDA and ROCm devices, and recovery behavior under worker interruptions.

The key contribution is a shift from “distributed training as a runtime property” to “distributed training as compiled semantics”. Fixing the forward pass, differentiation, and optimizer state transitions at the IR level prior to backend lowering reduces opportunities for semantic drift across heterogeneous accelerators and removes the need to deploy heavyweight framework stacks on every node. This design aligns with low-communication optimization (DiLoCo-style inner/outer loops), making synchronization intermittent and therefore more compatible with geo-distributed links and unstable membership. The resulting artifact boundary is also a clean mechanism for evolving training behavior: changing an optimizer or update rule becomes an artifact update rather than a worker rewrite, which is important for long-lived systems spanning multiple organizations and hardware pools.

The main weakness is that the evaluation does not yet establish long-horizon convergence, scaling behavior, or competitive end-to-end throughput against mature distributed training baselines under matched compute budgets. Several report metrics remain placeholders (cross-backend deviation tolerances, recovery-time distributions, per-round payloads), and strict determinism is not guaranteed across heterogeneous floating-point implementations even when high-level semantics are fixed. In addition, while execution is intentionally data-agnostic, robust operation under strongly non-IID data and cross-party governance requires explicit contracts for evaluation, monitoring, and aggregation policies that are not fully specified in the current report.

Future work should therefore focus on making the system operationally complete and quantitatively characterized. This includes formalizing a lightweight data/evaluation interface (metadata reporting plus standardized evaluation hooks) to make non-IID aggregation measurable without centralizing data; conducting systematic failure-injection studies (OOMs, device resets, disconnects, churn) with explicit KPIs for time-to-rejoin and progress under partial outages; adding artifact lifecycle controls at the protocol boundary (provenance, signing, registry-backed rollout/rollback) so updates are auditable and reversible; and instrumenting structured telemetry for both system health and learning dynamics to detect divergence, stale nodes, and abnormal update magnitudes early. With these additions, the same compiled-transition approach can be extended beyond supervised learning to distributed agentic and reinforcement learning pipelines by compiling policy/value updates into portable artifacts while maintaining intermittent, fault-tolerant synchronization.

References

[1] Arthur Douillard, Qixuang Feng, Andrei A. Rusu, Rachita Chhaparia, Yani Donchev, Adhiguna Kuncoro, Marc'Aurelio Ranzato, Arthur Szlam, and Jiajun Shen. Diloco: Distributed low-communication training of language models, 2023.

[2] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[3] Zachary Charles, Gabriel Teston, Lucio Dery, Keith Rush, Nova Fallen, Zachary Garrett, Arthur Szlam, and Arthur Douillard. Communication-efficient language model training scales reliably and robustly: Scaling laws for diloco, 2025.

[4] A. Sarfi, B. Thérien, J. Lidin, and E. Belilovsky. Communication-efficient LLM pre-training with sparseloco, 2025.

[5] B. Thérien, X. Huang, I. Rish, and E. Belilovsky. Muloco: Muon is a practical inner optimizer for diloco, 2025.

[6] J. Kolehmainen, N. Blagoev, J. Donaghy, O. Ersoy, and C. Nies. Noloco: No-all-reduce low communication training method for large models, 2025.

[7] B. Peng, J. Quesnelle, D. Rolnick, A. Lotter, U. H. Adil, and E. La Rocca. A preliminary report on distro. Technical report, 2024.

[8] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques A. Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: scaling compiler infrastructure for domain specific computation. In Jae W. Lee, Mary Lou Soffa, and Ayal Zaks, editors, *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, pages 2–14. IEEE, 2021.

[9] M. Vákár and T. Smeding. CHAD: Combinatory homomorphic automatic differentiation, 2021.

[10] Mai Jacob Peng and Christophe Dubach. Lagrad: Statically optimized differentiable programming in MLIR. In Clark Verbrugge, Ondrej Lhoták, and Xipeng Shen, editors, *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction, CC 2023, Montréal, QC, Canada, February 25-26, 2023*, pages 228–238. ACM, 2023.

[11] Martin Paul Lücke, Oleksandr Zinenko, William S. Moses, Michel Steuwer, and Albert Cohen. The MLIR transform dialect: Your compiler is more powerful than you think. In Johannes Doerfert, Tobias Grosser, Hugh Leather, and P. Sadayappan, editors, *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization, CGO 2025, Las Vegas, NV, USA, March 1-5, 2025*, pages 241–254. ACM, 2025.

[12] StableHLO Project. StableHLO. GitHub repository, 2025.

[13] The IREE Authors. IREE. Computer software, GitHub repository, 2019.

Supplementary Information

This section provides pseudocode for PCP's end-to-end distributed protocol and compiler pipeline. Algorithm 1 describes the runtime orchestration (Shepherd–Supervisor–Worker) coupled with DiLoCo-style outer/inner updates. Algorithm 2 describes construction of the single training-step StableHLO module (forward + reverse-mode AD + optimizer). Algorithm 3 details the reverse-mode AD transformation used to generate the backward graph.

Notation. θ denotes model parameters and S persistent optimizer state (e.g., AdamW moments). H is the number of inner (local) steps per outer round, and T is the number of outer rounds. Workers may run on heterogeneous backends $a \in \{\mathrm{CUDA}, \mathrm{ROCm}, \mathrm{Metal}\}$. $\Delta^{(k)}$ is the pseudo-gradient (weight delta) returned by worker k after H inner steps.

Algorithm 1 Planetary Compute Protocol (PCP): Distributed Training with Compiled Artifacts

```
Require: Initial global parameters $\theta^{(0)}$, outer rounds $T$, inner steps $H$
Require: Worker pool $\{w_k\}_{k=1}^{K}$ with heterogeneous backends $a_k$
Require: Compiler toolchain $\mathcal{C}$ (StableHLO → IREE → VMFB)
Require: Inner optimizer AdamW hyperparameters $\Omega_{\text{AdamW}}$
Require: Outer optimizer hyperparameters $(\mu, \eta_{\text{outer}})$ for Nesterov momentum

1: procedure SHEPHERD(Model)
2:     $\mathcal{M}_{\text{train}} \leftarrow$ BUILDTRAININGARTIFACT($Model, \Omega_{\text{AdamW}}$)        ▷ Alg. 2
3:     $\mathcal{B} \leftarrow$ BACKENDSPRESENT($\{a_k\}_{k=1}^{K}$)
4:     for all $a \in \mathcal{B}$ do
5:         $\mathcal{A}_a \leftarrow \mathcal{C}(\mathcal{M}_{\text{train}}, a)$        ▷ Compile backend-specific VMFB
6:     end for
7:     BROADCASTARTIFACTS($\{\mathcal{A}_a\}_{a\in\mathcal{B}}$)
8:     $v^{(0)} \leftarrow 0$
9:     for $t = 1 \ldots T$ do
10:        $\mathcal{W}_t \leftarrow$ SELECTACTIVEWORKERS($\{w_k\}_{k=1}^{K}$)
11:        for all $w_k \in \mathcal{W}_t$ do
12:            $(\text{off}_k, \text{len}_k) \leftarrow$ ASSIGNDATACHUNK($t, k$)
13:            SENDROUNDCONFIG($w_k, \theta^{(t-1)}, t, H, \text{off}_k, \text{len}_k$)
14:        end for
15:        $\{(\Delta_t^{(k)}, n_k)\}_{w_k\in\mathcal{W}_t} \leftarrow$ COLLECTDELTAS($\mathcal{W}_t$)
16:        $\Delta_t \leftarrow \dfrac{\sum_{w_k\in\mathcal{W}_t} n_k \Delta_t^{(k)}}{\sum_{w_k\in\mathcal{W}_t} n_k}$        ▷ Weighted mean aggregation
17:        $v^{(t)} \leftarrow \mu v^{(t-1)} + \Delta_t$
18:        $\theta^{(t)} \leftarrow \theta^{(t-1)} + \eta_{\text{outer}}\big(\mu v^{(t)} + \Delta_t\big)$
19:    end for
20: end procedure

21: procedure SUPERVISOR(backend $a$)
22:     CONNECTCONTROLPLANE()        ▷ Persistent session + heartbeats
23:     while JOBACTIVE() do
24:         if WORKERDEAD() or HEALTHCHECKFAIL() then
25:             BACKOFFSLEEP()
26:             SPAWNWORKERPROCESS($a$)
27:         end if
28:     end while
29: end procedure

30: procedure WORKER(backend $a$)
31:     $\mathcal{A}_a \leftarrow$ RECEIVEARTIFACT($a$)
32:     $S \leftarrow 0$        ▷ Persistent local optimizer state
33:     while TRAINING() do
34:         $(\theta, t, H, \text{off}, \text{len}) \leftarrow$ RECEIVEROUNDCONFIG()
35:         $\theta_0 \leftarrow \theta$
36:         $n \leftarrow 0$
37:         for $h = 1 \ldots H$ do
38:             $(x, y) \leftarrow$ NEXTBATCH(off, len)
39:             $(\theta, S, \ell) \leftarrow$ EXECUTE($\mathcal{A}_a, \theta, S, x, y, t$)
40:             $n \leftarrow n + |(x, y)|$
41:         end for
42:         $\Delta \leftarrow \theta_0 - \theta$
43:         SENDDELTA($\Delta, n$)
44:     end while
45: end procedure
```

Algorithm 2 Graph-to-Graph Construction: Forward StableHLO → Training-Step Module

```
Require: Forward module $\mathcal{M}_{\text{fwd}}$ (StableHLO)
Require: Optimizer specification $\Omega$ (e.g., AdamW hyperparameters)
Ensure: Training module $\mathcal{M}_{\text{train}}$ implementing $T(\theta, S, B, t) \to (\theta', S', L)$

1: function BuildTrainingArtifact($\mathcal{M}_{\text{fwd}}, \Omega$)
2:     $\mathcal{G}_{\text{fwd}} \leftarrow$ ParseStableHLO($\mathcal{M}_{\text{fwd}}$)
3:     RenameEntryPoint($\mathcal{G}_{\text{fwd}}$, main, forward_pass)

4:     $\mathcal{G}_{\text{fwd}} \leftarrow$ SanitizerPass($\mathcal{G}_{\text{fwd}}$)          ▷ $\epsilon$-guards, literal normalization, etc.

5:     $\mathcal{G}_{\text{bwd}} \leftarrow$ BuildGradientGraph($\mathcal{G}_{\text{fwd}}$)          ▷ Alg. 3
6:     AppendFunction($\mathcal{G}_{\text{bwd}}$, backward_pass)

7:     $\mathcal{G}_{\text{bwd}} \leftarrow$ RematerializePass($\mathcal{G}_{\text{bwd}}$)          ▷ Optional: reduce peak activation memory

8:     CreateEntryPoint(train_step($\theta, S, B, t$) $\to$ ($\theta', S', L$))
9:     $g \leftarrow$ Call(backward_pass, $\theta, B$)
10:    $(\theta', S') \leftarrow$ EmitOptimizerOps($\Omega, \theta, S, g, t$)
11:    $L \leftarrow$ Call(forward_pass, $\theta, B$)
12:    return SerializeStableHLO(train_step, forward_pass, backward_pass)
13: end function
```

Algorithm 3 StableHLO Reverse-Mode AD Pass (VJP + Adjoint Accumulation)

```
Require: Forward function $\Phi$ (DAG of StableHLO ops producing scalar loss $L$)
Ensure: Gradient function $\nabla\Phi$ producing $\nabla_\theta L$

1: function BuildGradientGraph($\Phi$)
2:     $\mathcal{G}_{\text{grad}} \leftarrow$ CreateEmptyFunction()
3:     adj $\leftarrow \emptyset$          ▷ Map: SSA value $\mapsto$ accumulated adjoint
                    ▷ Pass 1: clone forward ops so primals exist in gradient scope
4:     for all $op \in$ TopologicalSort($\Phi$) do
5:         $op' \leftarrow$ Clone($op$)
6:         Append($op', \mathcal{G}_{\text{grad}}$)
7:         RecordPrimalMapping($op \to op'$)
8:     end for
                    ▷ Pass 2: backpropagate adjoints in reverse topological order
9:     adj[$L$] $\leftarrow 1$
10:    for all $op \in$ ReverseTopologicalSort($\Phi$) do
11:        $\bar{Y} \leftarrow$ LookupAdjoints(Outputs($op$), adj)
12:        if $\bar{Y} = \emptyset$ then
13:            continue
14:        end if
15:        $(\bar{X}_1, \ldots, \bar{X}_n) \leftarrow$ EmitVJP($op, \bar{Y}$)
16:        for $i = 1 \ldots n$ do
17:            $\bar{X}_i \leftarrow$ NormalizeAdjointShape($\bar{X}_i$, Shape(Inputs($op$)$_i$))
18:            adj[Inputs($op$)$_i$] $\leftarrow$ Accumulate(adj[Inputs($op$)$_i$], $\bar{X}_i$)          ▷ Fan-out: sum multiple contributions
19:        end for
20:    end for
21:    return $\mathcal{G}_{\text{grad}}$
22: end function
```