Verifiable constraints on frontier training via proofs of compartmentalization

Daniel Reuter [1] **Luke Marks** [2] **Aurelio Carlucci** [3] **Jonathan Ng** [4] **James Petrie** [5 6] **Jason Hausenloy** [7] **Adam Karvonen** [4] **Mauricio Baker** [3]

Abstract

Laws and international agreements on frontier AI development may only be enforceable if the compliance of AI developers is verifiable. To protect the privacy of developers and their ability to perform work other than frontier AI development, it is important that we can verify the absence of frontier AI development without requiring that developers reveal confidential information, and without preventing them from performing inference. We propose a property of workloads called *compartmentalization* that we argue inference workloads naturally possess, but that frontier training workloads necessarily do not. We then introduce protocols for verifying whether a datacenter's workloads are compartmentalized called *proofs of compartmentalization*, and argue that a datacenter could not pass our protocols while simultaneously performing a frontier training run, even using adversarially designed training algorithms.

1. Introduction

It may soon be desirable to put in place laws or international agreements on frontier AI development. AI developers may be unlikely to support such laws or agreements if verifying compliance requires invasive procedures that threaten their privacy or their ability to perform work other than frontier AI training. Therefore, an important open problem is:

> How can an AI developer demonstrate to external auditors that it is not using its datacenters for frontier training while maintaining the confidentiality of its models, code, and data, and preserving its ability to do inference with existing AIs?

[1]Oxford Martin AIGI [2]Martian [3]University of Oxford [4]Independent [5]Future of Life Institute [6]Oxford Hardware AI Governance Lab [7]Center for AI Safety. Correspondence to: Luke Marks <luke@withmartian.com>.

Second Workshop on Technical AI Governance Research (TAIGR) @ ICML 2026, Seoul, South Korea. 2026. Copyright 2026 by the author(s).

We make progress on this problem by proposing the following:

1. A property of workloads called *compartmentalization* that we argue inference workloads possess but that frontier training workloads necessarily do not.
2. Protocols that prove the compartmentalization of a developer's workloads without needing to share confidential information.

We say that a workload is *compartmentalized* if it can be broken down into a limited number of cheap computations called *compartments* that each have small outputs and take as input only (i) information committed by a fixed date (e.g., models or code), and (ii) a small amount of uncommitted information (e.g., a prompt). This prevents frontier training because parameter updates are too large to be inputs to or outputs of compartments, and thus cannot be applied during training steps. By contrast, inference workloads are easily broken down into computationally cheap chat sessions with small inputs and outputs, making them compartmentalized.

We introduce two protocols called *Proofs of Compartmentalization* (PoComps), that if implemented with sufficient physical and hardware security would allow developers to prove that their network traffic consists entirely of inputs and outputs to compartmentalized workloads without revealing model weights, data, or code. Prior work has studied similar problems and proposed protocols for addressing them (Harack et al., 2025; Baker et al., 2025; Cankaya et al., 2026), but to our knowledge, we are the first to give a protocol for verifying the absence of frontier training in which the developer is not required to reveal confidential information.

2. Problem statement

We consider two parties, a *developer* and an *auditor*. The developer operates a datacenter using which it will perform work, and the auditor monitors the datacenter, intending to catch the developer if it performs a certain kind of work. We refer to the datacenter as the *monitored system*, and describe the period in which the auditor monitors the datacenter as the *monitoring period*.

The developer and auditor have an informal agreement stip-

ulating that the developer is not to use the monitored system for frontier training. We want a protocol that notifies the auditor if the developer violates the agreement, but in such a way that the developer is not required to reveal any confidential information, and can perform work other than frontier training. In particular, the developer should be able to run inference with models it has already trained.

To make the informal agreement verifiable, the developer and auditor will additionally agree to a formal proxy for whether some work is frontier training. Only this proxy can be verified, so the extent to which the auditor can get assurance of the developer's compliance with the agreement is the extent to which the proxy is able to detect whether the developer is doing a frontier training run. We introduce and argue for the robustness of one such proxy in Section 3, and then give protocols for verifying that proxy in Section 4.

3. Compartmentalization

We would like to design a protocol that runs over the monitoring period, rejecting if the developer runs frontier training workloads on the monitored system, and accepting otherwise. In particular, the developer running inference workloads should not cause the protocol to reject. A significant challenge in writing such a protocol is that "frontier training" is not a well-specified property of workloads. As a result, no procedure can directly check whether a workload is frontier training. However, we argue that frontier training and inference are fundamentally different in such a way that a simple property securely distinguishes them even without a robust specification for frontier training.

Consider that training accumulates information over many training steps in the weights of the model being trained. This accumulated information—the parameter updates—needs to be an input to future training steps if the updated model is sampled from in those steps. As a result, each training step in a training run depends on the outputs of all previous training steps. By contrast, inference workloads are naturally broken down into chat sessions that are mostly independent of one another in the sense that they do not need to take as input the outputs of other chat sessions. Each session takes as input the model, prompt, and inference stack, and outputs the generated tokens. A prompt may include tokens output by other chat sessions, but the size of the inputs originating from other chat sessions will be small, because prompts are small (relative to parameter updates).

Although the model weights and inference stack are large, and thus the inputs to each chat session are large, the developer had access to both before it ran the workload, and thus they could have been committed before the workload was ran. If we allow the developer to commit to information they knew before the monitoring period and count only the uncommitted information, then each chat session's inputs and outputs are small, and thus the extent to which a given chat session can depend on the output of other chat sessions is small.

Even counting only information that the developer did not commit to before running a training workload, the inputs and outputs to each training step are still large. This is because parameter updates are necessary inputs to training steps and cannot have been committed, as they are computed within the workload. Put another way, inference workloads can be broken down into small computations (which we refer to as "compartments") that each take as input only a small amount of uncommitted information, and output only a small amount of information, but training cannot. We therefore say that inference is *compartmentalized* and that training is not.

We formalize compartmentalization using the notion of a (C, X, Y)-*compartment*. A computation is a (C, X, Y)-compartment if it (i) receives at most X bits of uncommitted information as input, (ii) costs at most C H100-equivalent (H100e) hours to run, and (iii) produces at most Y bits of of output. We call a workload (C, X, Y, N)-*compartmentalized* if it can be decomposed into at most N (C, X, Y)-compartments. The effects of each of the parameters (C, X, Y, N) can be understood as follows:

- X limits how much information produced by other compartments (e.g., parameter updates) can be used by one compartment.
- Y limits how much information one compartment can provide to later compartments.
- C prevents a compartment from recreating the accumulated results of a large computation (e.g., parameter updates resulting from training) within one compartment.
- N prevents the workload from evading the other bounds by using arbitrarily many compartments (e.g., by having compartments represent individual arithmetic operations).

3.1. Setting the compartmentalization parameters

We first set the parameters (C, X, Y, N) such that they generously admit inference with DeepSeek-V4-Pro (DSV4-Pro) (DeepSeek-AI, 2026), and then show that these values exclude training. Although inference workloads could be compartmentalized in many ways, we calibrate (C, X, Y, N) to admit assigning one chat session per compartment. This allows us to set (C, X, Y, N) more tightly than if they were set with the intention of supporting compartments containing several chat sessions, or with the intention of having an

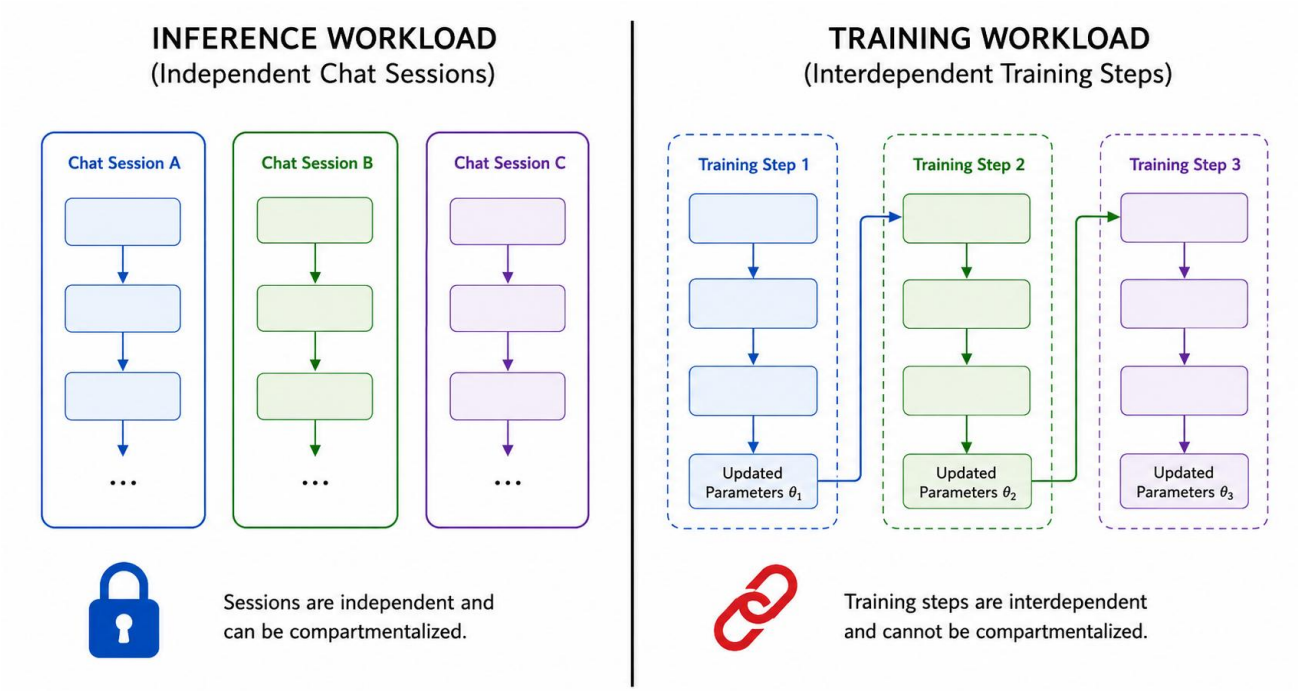

Figure 1. Inference workloads are naturally compartmentalized, while training workloads are necessarily not. Chat sessions can be treated as separate compartments because each session depends only on committed information plus a small prompt and produces a small output. By contrast, training steps are interdependent: each step must pass parameter updates to later steps, creating large cross-step dependencies that violate the input/output bounds required for compartmentalization.

individual chat session spread over several compartments, as this would require loading the KV cache for chat sessions as inputs to compartments, necessitating that X be large. We assume that the developer has committed an inference stack and the DSV4-Pro weights. The uncommitted inputs to compartments of an inference workload are therefore only the prompts, and the outputs only the generated tokens.

We set (C, X, Y, N) as follows:

- C is set to allow for generating one full context window of tokens. Public benchmarks report approximately 900 tokens/s on an 8x B200 machine (Lambda, 2026), which—measured by FP8 peak—is approximately 18 H100es (NVIDIA, 2026). Because DSV4-Pro has a context window of one million tokens, this works out to approximately 5.5 H100e hours. We set $C = 10$ H100e hours to account for headroom.
- X is set to allow for loading a full context window of input tokens into each compartment. DSV4-Pro has a vocabulary size of 129,000, so each token has a size of 2.25 bits. A full context window is therefore 2.25MB. We set $X = 10$MB to account for headroom.
- Y is set relative to X. We assume a $10 : 1$ input to output token ratio, giving $Y = 1$MB to account for headroom.
- N is set to admit full utilization of a large datacenter over the monitoring period, which we set to one calendar day. Stargate Abilene is expected to house approximately one million H100es (Epoch AI, 2026), giving 24 million H100e hours per monitoring period. Because $C = 10$, we would need $N = 2,400,000$ to allow for full utilization of the projected Stargate Abilene capacity. We therefore set $N = 10,000,000$ to account for headroom.

Now consider pretraining DSV4-Pro while needing to prove (C, X, Y, N)-compartmentalization. After training begins, later training steps must depend on parameter updates output by earlier training steps. A BF16 parameter update to DSV4-Pro, a one trillion parameter model, is approximately a terabyte — five OOMs larger than X, and six OOMs larger than Y. As a result, (C, X, Y, N) exclude standard pretraining, because a compartment could never take a parameter update as input, or output a parameter update. Thus compartments are prevented from depending one another in the way that is required for frontier training.

3.2. Adversarially designed training algorithms

The parameters set in Section 3.1 exclude standard training, but what about alternative algorithms that are designed to

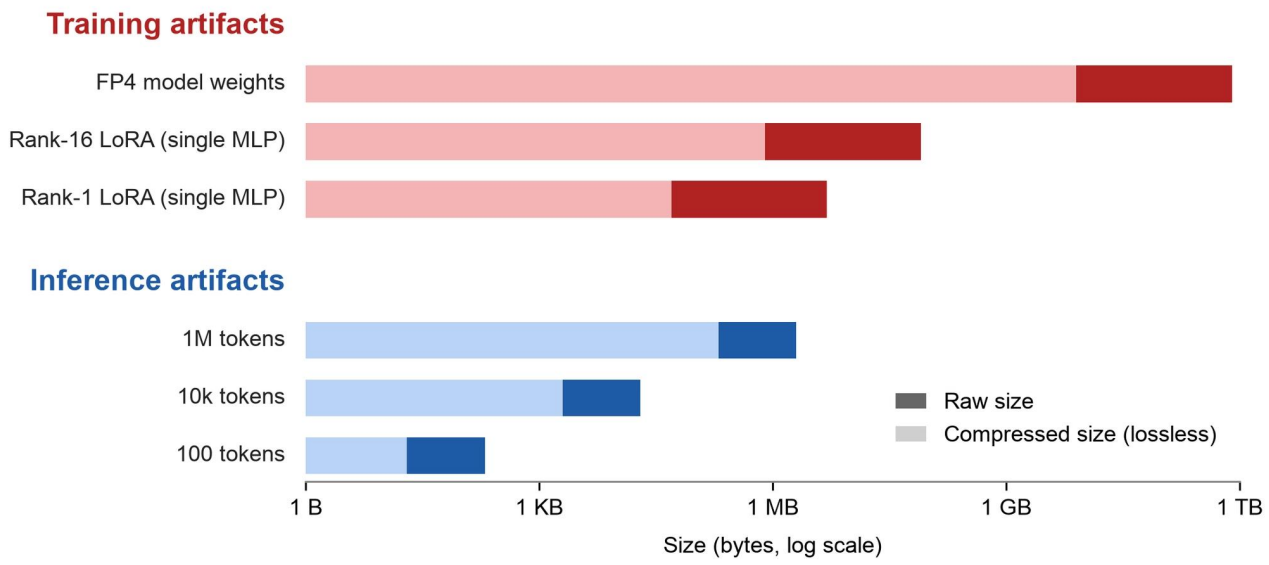

Figure 2. Sizes of artifacts used when training or serving DeepSeek-V4-Pro. Dark bars show raw sizes of artifacts used during inference, pretraining, and post-training; light bars show compressed sizes under current state-of-the-art lossless compression algorithms for each artifact type.

be compartmentalized? In principle, someone might discover an algorithm very different from conventional training, and this algorithm may not require loading large parameter updates. In Appendix A.4, we consider whether compartmentalization would constrain several unconventional algorithms were they to become performant. Because such algorithms replacing conventional training seems unlikely in the near future, this section focuses on modifications to current algorithms that could make frontier training compartmentalized.

First, model parameters and parameter updates are compressible at reported ratios of up to 50-100x (Han et al., 2016; Brown et al., 2026), bringing the size of parameter updates closer to X and Y. But text is compressible too: Wikipedia compresses more than tenfold (Delétang et al., 2024), and distributions like code may compress further (Hindle et al., 2012; Casalnuovo et al., 2018), allowing X and Y to be reduced—perhaps by a factor of ten—without prohibiting max-context inference. For further discussion of compression, see Appendix A.3 and B. In addition to input compression, requiring that X be several OOMs smaller than 10MB may be a reasonable requirement due to many tasks being solvable on smaller input budgets than an entire context window.

Second, a model can be updated using a parameter update much smaller than the entire model. For example, the cumulative size of FP4 rank-one LoRAs on all experts in a single layer of DeepSeek-V4-Pro would be 6MB: smaller than X. However, in instances where LoRAs have matched the performance of full fine-tuning on narrow distributions, they have been applied at every layer, bringing the total size of the LoRA parameters to 600MB for DSV4-Pro (Schulman & Lab, 2025). Additionally, LoRAs have few degrees of freedom. On SFT datasets such as Tulu3 and OpenThoughts3, Schulman & Lab (2025) find that low-rank LoRAs diverge from the full fine-tuning loss curve once the adapter runs out of capacity. Biderman et al. (2024) also find that on broad post-training distributions, low-rank adapters "substantially underperform" full fine-tuning. In Appendix A.1, we consider an attack in which the developer selectively loads LoRAs that are stored off-site for particular inputs, which may enable overcoming the limited capacity of low-rank of LoRAs, though this is empirically unsubstantiated.

Third, low-communication training algorithms such as DiLoCo (Douillard et al., 2024) reduce the total communication volume of training runs. While this may reduce the size of the inputs that would have to be passed to each compartment in a training run, such algorithms do not enable loading the parameter updates resulting from training using a smaller number of bits, except to the extent that they leverage compression on parameter updates. Therefore we do not consider them to pose a notable threat independent of compression.

Finally, the datacenter could use inference to produce outputs such as synthetic data or reward signal that help a small amount of unmonitored external compute train powerful models more efficiently. We discuss this attack in Appendix

A.2.

4. Proofs of compartmentalization

This section introduces *proofs of compartmentalization* (PoComps). A PoComp is an interaction between a developer and an auditor in which the auditor is convinced that the workloads run by the developer on the monitored system during the monitoring period are compartmentalized according to agreed parameters.

We introduce two PoComps with different security guarantees. Pod-PoComp, introduced in Section 4.1, gives a coarser guarantee but has a simple physical interpretation: a pod's computation between two memory erasures is one compartment. Task-PoComp, introduced in Section 4.3, gives a tighter guarantee by checking a random sample of smaller compartments.

Both protocols use the following conventions. The monitoring period is $[0, T]$ and is divided into audit epochs. Before the monitoring period, the developer makes a commitment c_0 to information $\mathcal{I}_0$ such as model weights, code, and data. The monitored system consists of *pods*: large units of compute such as server racks. Each pod's external communication links are observed by auditor-trusted gateways that produce signed digests of the traffic they observe.

Gateways parse pod ingress and egress into *messages*. A message is a pair $m_j = (\mathsf{desc}_j, x_j)$, where x_j is its contents and desc_j is its descriptor, recording the pod, gateway, direction, sequence number, audit epoch, start and end times, and length. Let

$$\widehat{m}_j = (\mathsf{desc}_j, H(x_j))$$

be the corresponding content-hashed message, and write $\mathrm{len}(\widehat{m}_j)$ for the length recorded in its descriptor. For a set S of content-hashed messages, let $\mathrm{new}_{\mathcal{I}_0}(S)$ denote the number of bits charged as uncommitted input under a public rule fixed at setup. For gateway g, pod p, and audit epoch e, the *gateway digest* is

$$D_{g,p,e} = (\mathsf{root}_{g,p,e}, \sigma^D_{g,p,e}),$$
$$\mathsf{root}_{g,p,e} = \mathsf{Root}(g, p, e, \widehat{m}_1, \ldots, \widehat{m}_n).$$

where $(\widehat{m}_1, \ldots, \widehat{m}_n)$ is the content-hashed messages output by that gateway for that pod and epoch ordered by sequence number, and $\sigma^D_{g,p,e}$ is the gateway's signature on $(g, p, e, \mathsf{root}_{g,p,e}, n)$. Let $\mathcal{D}_e = \{D_{g,p,e}\}_{g,p}$ denote the public gateway digests for epoch e. Gateways reject traffic they cannot parse into messages with unambiguous descriptors. We assume that at most B bits of traffic can enter or leave each pod over the monitoring period without being recorded by that pod's gateway.

4.1. Pod-PoComp: compartmentalization via pod boundaries

4.1.1. Overview and assumptions

Goal. If Pod-PoComp passes, then for each pod, the computation performed between consecutive erasures is one compartment. Gateway-observed ingress during between those erasures is the compartment's uncommitted input and gateway-observed egress is its output.

Definition 1. Fix an erasure schedule $0 = t_0 < t_1 < \cdots < t_m = T$. A workload is (C, X, Y)-*pod-compartmentalized* if, for every pod and every interval (t_i, t_{i+1}), the computation performed by each pod in each interval costs at most C, has uncommitted ingress of size at most X, and produces egress of size at most Y. The computation done by a pod between erasures is a compartment.

Memory erasure. Pod-PoComp forces compartments to begin and end by erasing the pod's memory at each time t_i. Any information produced before an erasure that is required by the pod after that erasure must either leave through a gateway and re-enter as later ingress, or survive in state that was not actually erased. Let R be the amount of state per pod that can survive an erasure. We assume that the auditor knows an upper bound C_{pod} on the compute capacity of each pod over $[0, T]$, and that the length of the period between erasures T_{erase} is chosen so that $C_{\mathrm{pod}} T_{\mathrm{erase}} \leq C$. For more details on memory erasure, see Appendix C.

Relation. The Pod-PoComp relation $\mathcal{R}_{\mathrm{pod}}$ is the predicate proved by the developer during a Pod-PoComp. Its public statement u_{pod} contains $(T_{\mathrm{erase}}, C_{\mathrm{pod}}, C, X, Y)$, c_0, the erasure schedule, and the gateway digests. Its witness w_{pod} contains $\mathcal{I}_0$ and its opening to c_0, the content-hashed messages opened from the gateway digests, any auxiliary information needed to evaluate $\mathrm{new}_{\mathcal{I}_0}$, and proof that each pod's memory was erased on schedule. We define $(u_{\mathrm{pod}}, w_{\mathrm{pod}}) \in \mathcal{R}_{\mathrm{pod}}$ iff:

- $\mathcal{I}_0$ opens c_0;
- for each $D_{g,p,e}$, the witness gives the content-hashed messages ordered by sequence number whose descriptors are consistent with (g, p, e), recompute $\mathsf{root}_{g,p,e}$, and make $\sigma^D_{g,p,e}$ a valid gateway signature on the root and sequence length;
- every message content x used as witness data is consistent with its content-hashed message $\widehat{m} = (\mathsf{desc}, H(x))$;
- the memory of each pod was erased at each t_i;
- for every interval (t_i, t_{i+1}), if S_{in} and S_{out} are the witnessed ingress and egress content-hashed mes-

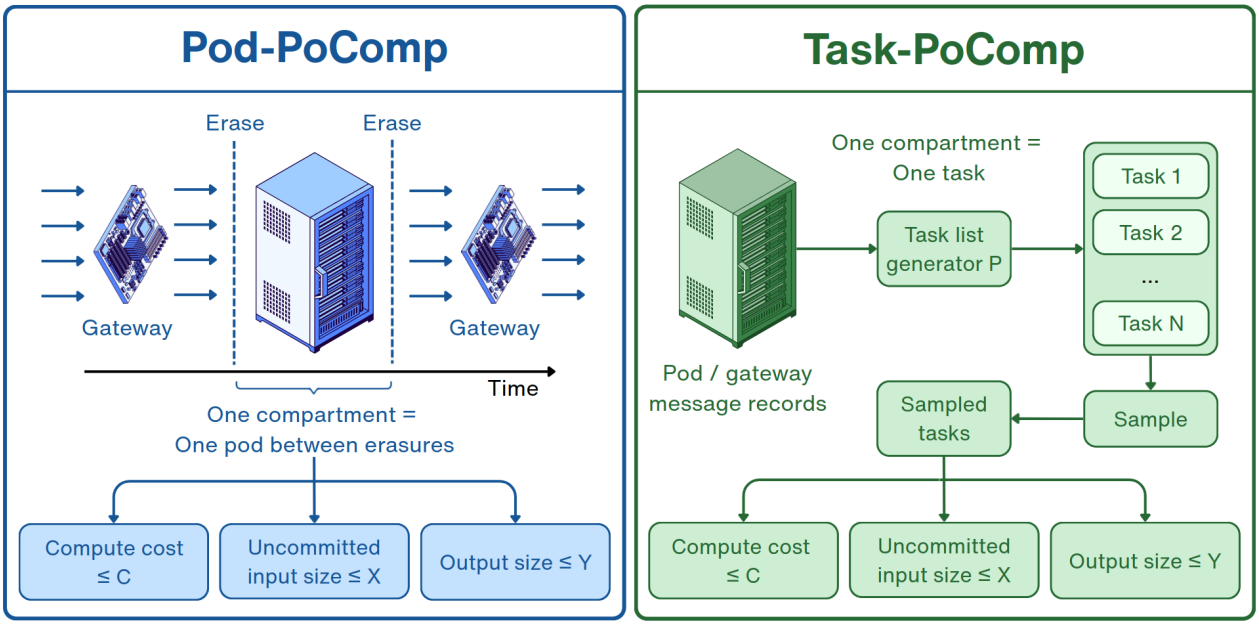

Figure 3. Overview of the two PoComp protocols. In Pod-PoComp, a pod's computation between memory erasures is treated as one compartment. In Task-PoComp, gateway message records are explained by tasks, which are sampled and proved to satisfy (C, X, Y).

sages in that interval, then $\mathrm{new}_{\mathcal{I}_0}(S_{\mathrm{in}}) \leq X$ and $\sum_{\widehat{m} \in S_{\mathrm{out}}} \mathrm{len}(\widehat{m}) \leq Y$.

4.1.2. Protocol

Setup. The developer and auditor fix public parameters

$$(T_{\mathrm{erase}}, C_{\mathrm{pod}}, C, X, Y, B, R, \Pi, \mathcal{R}_{\mathrm{pod}}),$$

where Π is the proof system used to prove membership in $\mathcal{R}_{\mathrm{pod}}$.

Execution. The developer schedules work freely. Gateways record all pod ingress and egress, publish signed gateway digests, and reject traffic that cannot be represented as messages. At times separated by T_{erase}, every pod erases the agreed memory regions and produces an erasure certificate.

Auditing. At the end of each epoch, the developer proves in zero knowledge that a witness w_{pod} exists such that $(u_{\mathrm{pod}}, w_{\mathrm{pod}}) \in \mathcal{R}_{\mathrm{pod}}$.

Verification. The auditor rejects unless the proof verifies.

4.1.3. Security properties

Soundness. Assume the gateway signatures, hash function, commitment scheme, and proof system are sound, that unrecorded communication for each pod is at most B bits over the monitoring period, and that at most R bits of state survive each erasure on any pod. If Pod-PoComp accepts, then except with negligible probability, each pod's computation between erasures is a compartment with compute cost at most C, uncommitted input at most $X + B + R$, and output at most $Y + B + R$. When $B = R = 0$, acceptance implies (C, X, Y)-pod-compartmentalization.

Confidentiality. The auditor receives only public parameters, gateway digests, and a zero-knowledge proof.

4.2. Discussion

Pod-PoComp is attractive because its soundness argument is mostly physical: if the pod's entire state is erased, then any inputs to the computation following the erasure must be observed by the gateway, and all outputs must be observed by the gateway. Its limitation is that X and Y must accommodate all ingress and egress handled by a pod between two erasures. For high-throughput inference, this may force X and Y to be too large to rule out frontier training.

For example, suppose each pod is an NVL72 rack. Linearly extrapolating from reported throughput for DeepSeek-V4-Pro on 8x B200 HGX systems (Lambda, 2026), an NVL72 could process roughly 33,000 input tokens/s and generate 8,200 output tokens/s at a 4:1 input-output ratio. If erasing an NVL72 takes about five hours and erasure occurs once per day, the pod could process about 2.3 billion input tokens and emit about 560 million output tokens between erasures. At 17 bits/token, Pod-PoComp would need roughly $X = 5\,\mathrm{GB}$ and $Y = 1\,\mathrm{GB}$. This leaves only a few orders of

magnitude between allowed compartment I/O and the size of a parameter update to a frontier model, and the margin shrinks further if updates compress well. Task-PoComp avoids this failure mode by allowing the developer to declare smaller units of computation as compartments.

4.3. Task-PoComp: compartmentalization via task boundaries

Task-PoComp asks the developer to explain the gateway digests as a list of small deterministic computations called tasks, and then checks a random sample of those tasks. This lets X and Y be calibrated to one inference request rather than to all pod ingress and egress between erasures. We examine whether tasks being deterministic is a reasonable assumption in Appendix 2.

4.3.1. Overview and assumptions

Tasks. The protocol uses a fixed deterministic interpreter $\mathsf{Exec}_{\mathcal{I}_0}$ which maps ingress messages to egress messages or rejects. Fix a pod p and epoch e. Let $\widehat{\mathcal{M}}^{\text{in}}_{p,e}$ and $\widehat{\mathcal{M}}^{\text{out}}_{p,e}$ be the ingress and egress content-hashed messages opened from the gateway digests for that pod and epoch. A *task* is a tuple $(S_{\text{in}}, S_{\text{out}})$, where $S_{\text{in}} \subseteq \widehat{\mathcal{M}}^{\text{in}}_{p,e}$, and $S_{\text{out}} \subseteq \widehat{\mathcal{M}}^{\text{out}}_{p,e}$. It claims that $\mathsf{Exec}_{\mathcal{I}_0}$, run on the contents of the messages named by S_{in} ordered by sequence number, produces output bitstrings whose hashes and lengths match the messages named by S_{out}, also ordered by sequence number. A task is *well-sized* if $\text{new}_{\mathcal{I}_0}(S_{\text{in}}) \leq X_{\text{task}}$ and $\sum_{\widehat{m} \in S_{\text{out}}} \text{len}(\widehat{m}) \leq Y_{\text{task}}$.

Task lists. A *task list* for a pod and epoch is a list of tasks. It is *I/O-comprehensive* if every ingress content-hashed message in $\widehat{\mathcal{M}}^{\text{in}}_{p,e}$ is named by at least one task and every egress content-hashed message in $\widehat{\mathcal{M}}^{\text{out}}_{p,e}$ is named by exactly one task.

Task list generation. To prevent the developer from choosing the task list after having observed the outputs of tasks, task lists must be generated by a committed deterministic program P. Before the monitoring period, the developer commits to P as c_P. For each epoch, the developer chooses a string A of length at most ℓ_A and publishes a binding commitment c_A before audit randomness is drawn. For each pod, P takes the ingress messages, and A and outputs a task list. The string A permits the developer to choose among task lists not determined by ingress alone, which is necessary because factors such as hardware failure may not be determined by ingress.

Relation. The Task-PoComp relation $\mathcal{R}_{\text{task}}$ is the predicate proved by the developer during a Task-PoComp. Its public statement u_{task} contains the task parameters $(C_{\text{task}}, X_{\text{task}}, Y_{\text{task}}, N_{\text{task}}, p, \ell_A)$, $\mathsf{Exec}_{\mathcal{I}_0}$, the commitments c_0, c_P, c_A, the gateway digests $\mathcal{D}_e$, and a sampling seed ρ generated after c_A is fixed. Its witness w_{task} contains $\mathcal{I}_0$, P, A, openings of c_0, c_P, c_A, openings of the relevant gateway digests to ordered content-hashed messages, ingress contents, and any auxiliary witness data needed to evaluate P, $\text{new}_{\mathcal{I}_0}$, and $\mathsf{Exec}_{\mathcal{I}_0}$. We define $(u_{\text{task}}, w_{\text{task}}) \in \mathcal{R}_{\text{task}}$ iff:

- the commitments c_0, c_P, c_A open correctly, the content-hashed messages open the public gateway digests as in $\mathcal{R}_{\text{pod}}$, and all supplied ingress contents hash to the corresponding content-hashed messages;
- $|A| \leq \ell_A$, and running P on the ingress messages gives a task list that is I/O-comprehensive for every pod, contains at most N_{task} tasks in total across the epoch, and contains only well-sized tasks;
- for the tasks sampled from the task list using ρ and probability p, each sampled task costs at most C_{task} according to $\mathsf{Exec}_{\mathcal{I}_0}$ and has an output whose length and hash match exactly the egress content-hashed messag named by that task.

4.3.2. Protocol

Setup. The developer and auditor run the setup from Pod-PoComp. They additionally fix task-level parameters $(C_{\text{task}}, X_{\text{task}}, Y_{\text{task}}, N_{\text{task}})$, a sampling probability p, the length bound ℓ_A, the interpreter $\mathsf{Exec}_{\mathcal{I}_0}$, the committed task-list program P, and the relation $\mathcal{R}_{\text{task}}$. Here N_{task} bounds the total number of tasks per epoch.

Execution. The epoch proceeds as in Pod-PoComp: the developer schedules work freely, while gateways publish signed gateway digests of pod ingress and egress.

Auditing. At the end of each epoch, after the gateway digests are published, the developer publishes c_A and the auditor samples ρ. The developer then proves membership in $\mathcal{R}_{\text{task}}$ for the sampled statement.

Verification. The auditor rejects unless the proof verifies.

4.3.3. Security properties

Definition 2. The gateway-observed messages are (δ, K)-*sound task-compartmentalized* if, except with probability at most δ, deleting at most K bits of egress from the messages committed by the gateway digests leaves egress messages whose remaining content-hashed messages are produced by well-sized tasks that each run in at most C_{task} compute, with at most N_{task} tasks per epoch.

Soundness. Suppose Pod-PoComp and Task-PoComp both accept for N_{epoch} epochs. Under the assumptions above, the gateway-observed messages are (δ, K)-sound task-compartmentalized for

$$K \leq \left\lceil \frac{\ln(1/\delta)}{p} \right\rceil Y_{\text{task}} + N_{\text{epoch}} \ell_A + B,$$

up to negligible error. The first term comes from sampling: once the gateway digests and c_A fix the task lists, if more than $\lceil \ln(1/\delta)/p \rceil$ invalid tasks remain, the chance that none are sampled is at most δ. Since each task is well-sized, those invalid tasks account for at most that many copies of Y_{task} output bits. The second term charges the per-epoch discretionary string A. The final term charges output that leaves through channels not observed by gateways.

Confidentiality. The auditor receives only public parameters, gateway digests, commitments, sampling randomness, and zero-knowledge proofs.

4.3.4. Discussion

Task-PoComp removes the main quantitative weakness of Pod-PoComp: X_{task} and Y_{task} scale with the size of one claimed task rather than with all traffic between two memory erasures. Its costs are a large zero-knowledge proof, a committed task-list generator P, and the sampling slack in K_{dig}. Running Pod-PoComp additionally ensures that gateway-observed egress not explained by Task-PoComp was at least accepted by Pod-PoComp.

5. Practical security parameters

For Task-PoComp, the main statistical parameter is the sampling rate p. Table 1 gives BOTECs for a one-year monitoring period with daily epochs, $Y_{\text{task}} = 1\,\text{MB}$, and $\delta = 2^{-40}$. We set $\ell_A \leq 1\,\text{MB}$ per epoch, which gives the developer substantial freedom in task-list generation but contributes only $365\,\text{MB}$ per year.

With $p = 10^{-3}$, $\ell_A = 1\,\text{MB}$ per day, and a sustained unobserved-channel bound of $1\,\text{kb/s}$, the total one-year slack is roughly $28 + 0.37 + 3.9 \approx 32\,\text{GB}$. With $p = 3 \times 10^{-3}$, it falls to roughly $14\,\text{GB}$. This is small relative to a BF16 update to a one-trillion-parameter model, which is about $2\,\text{TB}$, but it is close to the 20–40 GB one would get by applying a 50–100× compression factor to such an update. Thus, for year-long assurances against aggressively compressed training state, the physical side-channel target must be closer to $1\,\text{kb/s}$ than to single-digit kb/s, or the sampling rate must be increased further. Over a single day, by contrast, even $9\,\text{kb/s}$ contributes only $97\,\text{MB}$ to B, so the same hardware can support much tighter short-window guarantees.

The other practical bottleneck is proving cost. If the datacenter is sized as in Section 3.1, $N_{\text{task}} = 10^7$ tasks/day is enough to cover full utilization with $C_{\text{task}} = 10$ H100e-hours. Sampling at $p = 10^{-3}$ requires proving about 10^4 task replays per day; sampling at $p = 3 \times 10^{-3}$ requires about 3×10^4. If every task were near the worst-case compute cap, the native replay work alone would be p times the datacenter's work, i.e. 24,000–72,000 H100e-hours/day before zero-knowledge overhead. Current general-purpose zkVMs are far slower than native GPU tensor programs: for example, SP1 Turbo reports about 96 seconds to prove a 100M-cycle loop on a single GPU node, and RISC Zero's documentation emphasizes that zkVM proving time scales with VM cycles and that floating-point operations are emulated rather than native (Roy, 2025; RISC Zero, 2026). Specialized ZKML is improving quickly—zkGPT reports GPT-2 inference proofs in under 25 seconds—but this is still far from cheaply proving frontier-scale, million-token inference (Qu et al., 2025). Consequently, the parameter set above is a security target, not a claim that full Task-PoComp replay is already cheap. Near-term deployments should either sample relatively small deterministic tasks, use specialized ZKML circuits for the inference stack, or rely on Pod-PoComp for coarse compartmentalization while treating Task-PoComp as the proof-system bottleneck.

Pod-PoComp has a different cost profile. Rather than in proving the cost the of tasks, the expensive operation is erasure. Amodo Design (2026) gives fill times of about 8.6 minutes for one GPU's 190 GiB HBM and 6.7 minutes for one host's 490 GiB DRAM, but almost five hours for a host with 15.36 TB of local disk. Their estimated challenge phase samples 0.1% of labels on a GB200-scale system in about eight minutes. Since devices can run erasure sessions in parallel, HBM and DRAM are not the limiting factor; local storage is. A daily five-hour erasure window costs about 21% utilization, while a weekly five-hour erasure window costs about 3%. If storage cannot be removed from the monitored pod, the erasure schedule must be chosen with this utilization cost in mind.

Gateway logging is comparatively cheap. A conservative signed message record with a 32-byte content hash, a 64-byte signature, and descriptors, timestamps, and lengths fits well under 256 bytes. If 10^7 tasks/day each generate one ingress and one egress record, raw gateway logs are under 5.2 GB/day before compression. This is small compared with checkpoint storage and can be retained or recursively committed without affecting workload scheduling.

6. Conclusion

This paper proposes compartmentalization as a verifiable proxy for the absence of frontier training. The motivating intuition for why compartmentalization distinguishes frontier

Parameter	Example value	BOTEC consequence
Failure probability	$\delta = 2^{-40}$ per year	$\ln(1/\delta) \approx 27.7$
Task output budget	$Y_{\mathtt{task}} = 1\,\mathrm{MB}$	Matches the inference-calibrated value from Section 3.1
Sampling rate	$p = 10^{-3}$	$\approx 10^4$ samples/day if $N_{\mathtt{task}} = 10^7$; sampling slack $\approx 28\,\mathrm{GB}$
High-assurance sampling	$p = 3 \times 10^{-3}$	$\approx 3 \times 10^4$ samples/day; sampling slack $\approx 9.3\,\mathrm{GB}$
Task-list discretion	$\ell_A \leq 1\,\mathrm{MB}$/epoch	$365\,\mathrm{MB}$ over daily epochs for one year
Side-channel target	$b \leq 1\,\mathrm{kb/s}$	$B \leq 10.8\,\mathrm{MB}$/day and $B \leq 3.9\,\mathrm{GB}$/year

Table 1. Example Task-PoComp parameters for a one-year monitoring period. Storage units are decimal.

training from inference is simple: inference can be decomposed into many small computations whose outputs and uncommitted inputs are small, while training accumulates large learned state that must persist across steps. By bounding the compute, uncommitted input, output, and number of allowed compartments, we can therefore exclude frontier training.

PoComps can verify that a developer's workloads are compartmentalized, and that they therefore cannot have been frontier training. In Pod-PoComp, a compartment is the computation done between two memory erasures. Pod-PoComp gives a simple, but coarse guarantee of compartmentalization. In Task-PoComp, the developer proves that each pod's ingress and egress traffic is explained by a list of tasks for which each tasks is its own compartment. Tasks can be very small, allowing us to more tightly bound the input, output, and cost of each task. In both protocols, zero-knowledge proofs allow the auditor to verify the relevant relation without learning model weights, prompts, code, or user data. We intend for PoComps to be the technical means by which future laws or international agreements on frontier AI developerment could be verified in a way that is convenient and minimally invasive for AI developers.

References

Amodo Design. Memory wiping – a first test. https://amododesign.com/notes/memory-wipes-a-first-test/, June 2026. Accessed 2026-06-30.

Arun, A., St. Arnaud, A., Titov, A., Wilcox, B., Kolobaric, V., Brinkmann, M., Ersoy, O., Fielding, B., and Bonneau, J. Verde: Verification via refereed delegation for machine learning programs, 2025. URL https://arxiv.org/abs/2502.19405.

Baker, M., Kulp, G., Marks, O., Brundage, M., and Heim, L. Verifying international agreements on ai: Six layers of verification for rules on large-scale ai development and deployment. Technical Report WR-A4077-1, RAND Corporation, 2025. URL https://www.rand.org/pubs/working_papers/WRA4077-1.html.

Biderman, D., Portes, J., Ortiz, J. J. G., Paul, M., Greengard, P., Jennings, C., King, D., Havens, S., Chiley, V., Frankle, J., Blakeney, C., and Cunningham, J. P. Lora learns less and forgets less, 2024. URL https://arxiv.org/abs/2405.09673.

Brown, D., Rivera, J.-P., Hendrycks, D., and Mazeika, M. Aggressive compression enables llm weight theft. 2026. doi: 10.48550/arXiv.2601.01296. URL https://arxiv.org/abs/2601.01296.

Cankaya, N., Kryś, J., Ng, J., Marks, L., and Krueckel, F. Fingerprinting all ai cluster i/o without mutually trusted processors. Research paper, Oxford Martin AI Governance Initiative, April 2026. URL https://aigi.ox.ac.uk/publications/fingerprinting-all-ai-cluster-i-o-without-mutually-trusted-processors/. Published April 28, 2026.

Casalnuovo, C., Sagae, K., and Devanbu, P. Studying the difference between natural and programming language corpora. 2018. doi: 10.48550/arXiv.1806.02437. URL https://arxiv.org/abs/1806.02437.

Chen, S., Jie, Z., and Ma, L. LLaVA-MoLE: Sparse mixture of LoRA experts for mitigating data conflicts in instruction finetuning MLLMs. 2024. doi: 10.48550/arXiv.2401.16160. URL https://arxiv.org/abs/2401.16160.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. PaLM: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023. URL https://jmlr.org/papers/v24/22-1144.html.

Davis, L., Haller, L., Alfarano, A., and Santolucito, M. Lattice deduction transformers. 2026. doi: 10.48550/arXiv.2605.08605. URL https://arxiv.org/abs/2605.08605.

DeepSeek-AI. DeepSeek-V4: Towards highly efficient million-token context intelligence. Technical report, DeepSeek-AI, 2026.

Delétang, G., Ruoss, A., Duquenne, P.-A., Catt, E., Genewein, T., Mattern, C., Grau-Moya, J., Wenliang, L. K., Aitchison, M., Orseau, L., Hutter, M., and Veness, J. Language modeling is compression. 2024. doi: 10.48550/arXiv.2309.10668. URL https://arxiv.org/abs/2309.10668.

Douillard, A., Feng, Q., Rusu, A. A., Chhaparia, R., Donchev, Y., Kuncoro, A., Ranzato, M., Szlam, A., and Shen, J. Diloco: Distributed low-communication training of language models. 2024. doi: 10.48550/arXiv.2311.08105. URL https://arxiv.org/abs/2311.08105.

Epoch AI. OpenAI Stargate Abilene, 2026. URL https://epoch.ai/data/ai-data-centers/directory/openai-stargate-abilene. Updated: 2026-06-26. Accessed: 2026-06-29.

Feng, W., Hao, C., Zhang, Y., Han, Y., and Wang, H. Mixture-of-LoRAs: An efficient multitask tuning method for large language models. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pp. 11371–11380, Torino, Italia, May 2024. ELRA and ICCL. URL https://aclanthology.org/2024.lrec-main.994/.

Gao, C., Chen, K., Rao, J., Liu, R., Sun, B., Zhang, Y., Peng, D., Guo, X., and Subrahmanian, V. MoLA: MoE LoRA with layer-wise expert allocation. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 5112–5127, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.findings-naacl.284. URL https://aclanthology.org/2025.findings-naacl.284/.

Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *International Conference on Learning Representations*, 2016. doi: 10.48550/arXiv.1510.00149. URL https://arxiv.org/abs/1510.00149. Oral presentation.

Harack, B., Trager, R. F., Reuel, A., Manheim, D., Brundage, M., Aarne, O., Scher, A., Pan, Y., Xiao, J.,

Loke, K., Adan, S. N., Bas, G., Caputo, N. A., Morse, J. C., Ahuja, J., Duan, I., Egan, J., Bucknall, B., Rosen, B., Araujo, R., Boulanin, V., Lall, R., Barez, F., Alvira, S., Katzke, C., Atamli, A., and Awad, A. Verification for international ai governance. Technical report, Oxford Martin AI Governance Initiative, July 2025. URL https://aigi.ox.ac.uk/wp-content/uploads/2025/07/Verification_for_International_AI_Governance.pdf.

Hindle, A., Barr, E. T., Gabel, M., Su, Z., and Devanbu, P. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pp. 837–847, Piscataway, NJ, USA, 2012. IEEE Press. URL https://dl.acm.org/doi/10.5555/2337223.2337322.

Jolicoeur-Martineau, A. Less is more: Recursive reasoning with tiny networks. 2025. doi: 10.48550/arXiv.2510.04871. URL https://arxiv.org/abs/2510.04871.

Kim, K., Kotha, S., Liang, P., and Hashimoto, T. Pretraining under infinite compute, 2025. URL https://arxiv.org/abs/2509.14786.

Kunwar, P., Vu, M. N., Gupta, M., Abdelsalam, M., and Bhattarai, M. TT-LoRA MoE: Using parameter-efficient fine-tuning and sparse mixture-of-experts. In *The International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '25, pp. 1332–1350. ACM, 2025. doi: 10.1145/3712285.3759888. URL https://doi.org/10.1145/3712285.3759888.

Lambda. How to deploy DeepSeek-V4-Pro on Lambda, 2026. URL https://lambda.ai/inference-models/deepseek-ai/deepseek-v4-pro. Accessed: 2026-06-29.

Li, D., Ma, Y., Wang, N., Ye, Z., Cheng, Z., Tang, Y., Zhang, Y., Duan, L., Zuo, J., Yang, C., and Tang, M. MixLoRA: Enhancing large language models fine-tuning with LoRA-based mixture of experts. 2024. doi: 10.48550/arXiv.2404.15159. URL https://arxiv.org/abs/2404.15159.

Lin, Y., Han, S., Mao, H., Wang, Y., and Dally, W. J. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *International Conference on Learning Representations*, 2018. doi: 10.48550/arXiv.1712.01887. URL https://arxiv.org/abs/1712.01887.

Malladi, S., Gao, T., Nichani, E., Damian, A., Lee, J. D., Chen, D., and Arora, S. Fine-tuning language models with just forward passes, 2024. URL https://arxiv.org/abs/2305.17333.

Meituan LongCat Team. LongCat-Flash technical report, 2025. URL https://arxiv.org/abs/2509.01322.

NVIDIA. *CUDA Programming Guide: CUDA Graphs*. NVIDIA, 2026. URL https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/cuda-graphs.html. Accessed 2026-04-25.

PyTorch Contributors. PyTorch documentation: Reproducibility. https://docs.pytorch.org/docs/stable/notes/randomness.html, 2025. Accessed 2026-04-25.

Qiang, X., Chen, H., Sun, S., Leng, J., Liu, X., and Guo, M. DASH: Deterministic attention scheduling for high-throughput reproducible LLM training, 2026. URL https://arxiv.org/abs/2601.21824.

Qu, W., Sun, Y., Liu, X., Lu, T., Guo, Y., Chen, K., and Zhang, J. zkGPT: An efficient non-interactive zero-knowledge proof framework for LLM inference. In *34th USENIX Security Symposium (USENIX Security 25)*, pp. 2045–2063, Seattle, WA, August 2025. USENIX Association. ISBN 978-1-939133-52-6. URL https://www.usenix.org/conference/usenixsecurity25/presentation/qu-zkgpt.

Rinberg, R., Carrell, A. M., Henniger, S., Carlini, N., and Warr, K. Haiku to Opus in just 10 bits: LLMs unlock large compression gains. 2026. doi: 10.48550/arXiv.2604.02343. URL https://arxiv.org/abs/2604.02343.

RISC Zero. Guest optimization guide. https://dev.risczero.com/api/zkvm/optimization, 2026. Accessed 2026-06-30.

Roberts, A., Chung, H. W., Mishra, G., Levskaya, A., Bradbury, J., Andor, D., Narang, S., Lester, B., Gaffney, C., Mohiuddin, A., et al. Scaling up models and data with t5x and seqio. *Journal of Machine Learning Research*, 24(377):1–8, 2023. URL https://jmlr.org/papers/v24/23-0795.html.

Roy, U. SP1 Turbo: The world's fastest zkvm just got faster. https://blog.succinct.xyz/sp1-turbo/, January 2025. Accessed 2026-06-30.

Sarfi, A., Thérien, B., Lidin, J., and Belilovsky, E. Communication efficient LLM pre-training with SparseLoCo. 2025. doi: 10.48550/arXiv.2508.15706. URL https://arxiv.org/abs/2508.15706.

Schulman, J. and Lab, T. M. Lora without regret. *Thinking Machines Lab: Connectionism*, 2025. doi: 10.64434/tml.20250929. https://thinkingmachines.ai/blog/lora/.

Song, Y., Swamy, G., Singh, A., Bagnell, J. A., and Sun, W. The importance of online data: Understanding preference fine-tuning via coverage. 2024. doi: 10.48550/arXiv.2406.01462. URL https://arxiv.org/abs/2406.01462.

TensorFlow Developers. TensorFlow api documentation: tf.config.experimental.enable_op_determinism. https://www.tensorflow.org/api_docs/python/tf/config/experimental/enable_op_determinism, 2024. Accessed 2026-04-25.

Wang, H., Hao, S., Dong, H., Zhang, S., Bao, Y., Yang, Z., and Wu, Y. Offline reinforcement learning for LLM multi-step reasoning. 2024. doi: 10.48550/arXiv.2412.16145. URL https://arxiv.org/abs/2412.16145.

Wu, M., Anand, A., Verma, S., and Mezini, M. Efficient post-training of LLMs for code generation with offline reinforcement learning. 2026. doi: 10.48550/arXiv.2605.28409. URL https://arxiv.org/abs/2605.28409.

Xie, P., Zhang, X., and Chen, S. RepDL: Bit-level reproducible deep learning training and inference, 2025. URL https://arxiv.org/abs/2510.09180.

Xu, J., Lai, J., and Huang, Y. MeteoRA: Multiple-tasks embedded LoRA for large language models. 2024. doi: 10.48550/arXiv.2405.13053. URL https://arxiv.org/abs/2405.13053.

Zhang, Z., Li, J., Lan, Y., Wang, X., and Wang, H. An empirical study on prompt compression for large language models. In *Building Trust Workshop at ICLR 2025*, 2025. doi: 10.48550/arXiv.2505.00019. URL https://arxiv.org/abs/2505.00019.

Zheng, S., Zheng, X., Chang, L.-w., and Zhai, J. UniEP: Unified expert-parallel MoE megakernel for LLM training, 2026. URL https://arxiv.org/abs/2604.19241.

A. Residual attack surface

We outline plausible attacks that could enable a developer to perform a frontier training run despite having to prove the compartmentalization of their workloads:

- **Mixture-of-adapters:** the developer routes requests to small expert adapters that can be loaded within X. Although a given adapter may only provide a narrow capabilities increase, by choosing a performant adapter for each request, the developer can effectively make their AIs more capable.
- **Offline post-training:** the developer generates rollouts used to do backward passes on some compute that does not have to prove that its work is compartmentalized, enabling the developer to assist in a large training run.
- **Compression-based attacks:** the developer compresses objects like gradients and weights to prevent training from violating X.
- **Unconventional algorithms:** The developer uses an algorithm very different from current training algorithms, and this algorithm can create frontier AI systems entirely using compartmentalized workloads.

More sophisticated attacks are likely possible, but these represent our current concerns. None of these attacks seem sure to work, though it is difficult to rule them out.

A.1. Mixture-of-adapters

This attack exploits the fact that small adapters (e.g., LoRAs) can be similarly sized to large prompts (e.g., a rank-1 LoRA on the experts in a single layer is approximately equivalent to a few million tokens), meaning that permitting large prompts requires also permitting inference with small adapters. While loading a single small adapter is unconcerning, a developer could store many adapters off-site and route individual requests to the most helpful adapter. In this way, the developer has many additional capabilities stored, but only a small number can be active at any given time, because loading many small adapters simultaneously would violate X.

Adapter mixtures appear in the literature, though mostly on small models (Chen et al., 2024; Li et al., 2024; Xu et al., 2024). TT-LoRA MoE chooses a single rank-5 expert per-input and is able to match the performance of a rank-16 adapter, but only includes results for Llama-3.2-1B on supervised classification tasks (Kunwar et al., 2025). MoLA is able to incrementally outperform a rank-64 LoRA using an adapter mixture, but they route to the top-2 rank-8 LoRAs at each layer for a given token, which would likely violate X (Gao et al., 2025). Mixture-of-LoRAs improves relative to a baseline adapter the same size as an individual expert in the mixture, meaning that an attack based on Mixture-of-LoRAs would likely be incrementally more powerful than using no mixture (Feng et al., 2024). Most papers on adapter mixtures load adapters for many layers, meaning that a developer using these methods in a mixture of adapters attack would require that X was set permissively enough that several adapters could be loaded. If X were that large, it would likely be to accommodate tens of millions of input tokens per task, or perhaps non-text modalities. There is not currently any empirical evidence to support mixture-of-adapters being significantly more useful than individual adapters, though the attack still seems plausible.

A.2. Off-policy post-training

The developer could collude with compute not subject to our constraints, generating rollouts used to post-train a model on the colluding compute. Because the required capacity of the colluding compute would be less than the required capacity to do the entire post-training run (only backward passes would be required), and because the colluding compute can be heavily distributed, it may not be feasible to prevent attacks like this simply by enforcing that all compute is subject to PoComps.

The feasibility of this attack depends largely on the usefulness of offline post-training. Because any model the developer uses for inference can only be updated with at most X bits plus whatever updates can be computed with C compute, this training would be off-policy, and potentially fully offline if it is not tractable to load both updates and inputs simultaneously. In current practice, it is unusual to do RLVR-style coding or math post-training in a strongly off-policy way. However, Wang et al. (2024) show positive results for an offline method on coding and math tasks on Qwen2.5-Math-1.5B and DeepSeekMath-7B-Instruct, though Wu et al. (2026) more recently show mixed results for an offline method applied to Qwen-2.5-Coder-7B.

One reason to be skeptical of this attack is that off-policy RL is in theory less effective on long-horizon tasks due to the behavior policy and target policy's distributions diverging more as the episode length grows. Given that solving difficult tasks currently requires unusually long episode lengths, we might expect this to be true for difficult tasks in the future also, perhaps more so as spending on inference compute increases. Off-policy RL is also less sample-efficient on long-horizon tasks: Song et al. (2024) show that offline methods require global coverage of the behavior policy's trajectories, implying that the amount of off-policy we should expect to need grows quickly with episode length. The effects of episode length are important, because the kinds of tasks we are likely to want models to get better at require long trajectories to complete, but this might fundamentally

contend with the regimes off-policy training is effective in.

In a more sophisticated version of this attack, the developer could train and load new adapters as the student policy diverges from the otherwise frozen policy, keeping the training sufficiently on-policy for the student policy to acquire new capabilities. As with the fully offline version of this attack, it is less feasible at longer episode lengths, as the size of the update required to prevent the policies from diverging too far grows with the episode length at which the behavior policy is trained.

A.3. Compression-based attacks and low-communication training

A natural thought is to compress parameter updates such that they can be loaded within X, or output within Y, enabling training. However, compression only benefits an adversarial developer to the extent that parameter updates are more compressible than inputs, because X can be set to the expected size of compressed inputs. Thus, if inputs compress by $\mathrm{com}_{\mathrm{input}}$ and data resulting from training compresses by $\mathrm{com}_{\mathrm{train}}$, we can measure how much an adversarial developer benefits from compression by $\frac{\mathrm{com}_{\mathrm{train}}}{\mathrm{com}_{\mathrm{input}}}$. If $\mathrm{com}_{\mathrm{input}}$ were larger than $\mathrm{com}_{\mathrm{train}}$, then compression favors the auditor.

Existing gradient-compression methods are effective. Quantization alone can compress gradients 16-32x relative to FP32, and Lin et al. (2018) report 270-600x compression without accuracy loss via a range of techniques including sparsification and quantization. However, loading a single compressed gradient is unlikely to be useful to an adversarial developer, as an individual gradient cannot typically be used to increase the capabilities of a model. Rather, a developer would have to aggregate several gradients into meaningful parameter updates and load those, which likely compress to an extent closer to weights than individual gradients. Han et al. (2016) report 35-49x reductions for AlexNet/VGG-style models, and Brown et al. (2026) give 16x-100x compression as an estimate for what an attacker might be able to achieve for FP16 Qwen2 models given that the compressed model can be fine-tuned to recover the original model's performance.

Low-communication training algorithms like SparseLoCo (Sarfi et al., 2025) use compression techniques similar to Lin et al. (2018) combined with DiLoCo-style synchronization, achieving a 3600x communication reduction over a naive AdamW baseline. However, these algorithms do not actually decrease the number of bits needed to load a parameter update as an input to a task, except to the extent to which they leverage compression on parameter updates. Consider the effect of DiLoCo-style synchronization on the compartmentalization of training: we compute many gradients in one task, aggregating them into a single parameter update before synchronizing with other workers. While it requires less communication to synchronize the aggregated parameter updates than it would to synchronize on every gradient, the aggregated parameter updates are many OOMs too large to fit within X, so DiLoCo-style synchronization did not help compartmentalize training.

Although parameter updates and weights are highly compressible, there is evidence that text is compressible at similar ratios. Rinberg et al. (2026) show that domain-adapted LoRAs and LLM-based arithmetic coding can be used to lossily compress text 33x, and their interactive Question-Asking protocol achieves lossy compression up to 250-1667x on some tasks. Zhang et al. (2025) survey prompt compression techniques, finding that on long-context evaluations, compression ratios of 10x can come with performance gains. In the figure below, we average all of their long-context evaluations and plot them against the performance measured at different compression ratios.

These numbers show that the ratio $\frac{\mathrm{com}_{\mathrm{train}}}{\mathrm{com}_{\mathrm{input}}}$ does not obviously favor an adversarial developer, meaning that compression does not necessarily have negative security implications. This could change if future methods achieved larger compression ratios for parameter updates and progress on input compression was slow. We further consider the compressibility of code and related structured data in Appendix B.

A.4. Unconventional algorithms

We consider three case studies of unconventional learning algorithms, and ask whether they could be compartmentalized if they were run at the scale of frontier training runs. These case studies are intended to provide evidence for whether PoComps are likely to constrain learning algorithms that are very different from conventional training algorithms.

A.4.1. Evolutionary algorithms

An evolutionary algorithm maintains candidate genomes, evaluates their fitness, and uses the results to generate new candidates. We break these algorithms down into three steps: mutation, evaluation, and selection, and consider whether these steps can be combined or broken down further in such a way that evolutionary algorithms are compartmentalized.

Mutation. The uncommitted inputs are the parent genomes, a small random seed, and possibly some auxiliary information, which together produce a child genome. The parent genomes may be compressible against the committed information (e.g. by representing the parent genomes as diffs against committed genomes), but ultimately seem like the largest source of uncommitted information in mutation, particularly if the genomes are large (e.g. they are neural network weights). Thus whether X constrains evolution-

ary algorithms depends primarily on the genome size and potentially on the number of parent genomes. A mutation operation that requires reading from all parent genomes at once would require that all parent genomes be present as uncommitted inputs, but a mutation operation that could read the parent genomes in sequence could be broken down into several tasks, each requiring only one parent genome.

In existing algorithms, mutation is very computationally cheap, so it seems unlikely that C would constrain mutation in any way except by preventing the parent genomes from being recomputed. Future algorithms might benefit from expensive mutation operations, in which case C might constrain the kinds of mutation operations that are allowed.

Evaluation. The uncommitted inputs are the candidate genome and any inputs needed for the evaluation function (e.g. inputs from an environment, a random seed, etc.) These inputs seem likely to be smaller than the mutation inputs unless the size of the genome is very small, in which case the inputs could dominate. This is because mutation may require several parent genomes. Evaluation does not violate X when the candidate genome and evaluation inputs are small.

The compute cost is the cost of running the candidate genome on the evaluation. Evaluation satisfies C only if a candidate genome can be evaluated for less than C, or at least broken up into several tasks of cost less than C. It may not be possible to break down evaluations into several tasks if doing so would require passing lots of uncommitted information (e.g. neural network activations) to other tasks. However, if compact representations like tokens can be passed between tasks, then C plausibly does not affect evaluation at all.

Selection. The uncommitted inputs are a set of tuples (genome, evaluation result). There are selection strategies that can keep the number of tuples that need to be loaded small (e.g. tournament selection, pairwise elimination, tree reductions), but others require loading many tuples (e.g. full-covariance CMA-ES, quality-diversity methods, anything with a replay-buffer), potentially making the inputs larger than X if the genome size is large.

Conclusion. Evolutionary algorithms over small genomes can be run within C and X even at enormous scale. Evolutionary algorithms over large genomes will be constrained by X, particularly during mutation and selection. C is likely to be a constraint to the same extent it is during training: it stops cutting down on uncommitted inputs by preventing genomes from being recomputed from committed information. It seems unlikely that C constrains evolutionary algorithms in other ways.

A.4.2. Small recursive reasoners

Recursive reasoning models with less than 10m parameters are able to significantly outperform larger models on puzzle tasks like Sudoku (Jolicoeur-Martineau, 2025; Davis et al., 2026). However, superior performance on language and coding tasks has not been demonstrated. We consider whether it would be possible to make training these smaller recursive models compartmentalized in the case that they were to approach the general capabilities of conventional language models via scaling.

Recursive reasoning models are trained using nearly identical algorithms to conventional language models – the important difference is in the size of the parameter updates. A 1.8M-parameter lattice deduction transformer is about 3.6MB and can outperform trillion parameter language models on puzzle tasks. Parameter updates on models this small are tiny – much smaller than 10MB, which is what we suggested calibrating X to in Section 3.1. As a result, parameter updates could easily fit within X, meaning that training a model this small can be compartmentalized. However, these models are far from outperforming language models on economically useful tasks, and they may need to be much larger to do so. For an economically useful recursive reasoning model to be able to be trained despite compartmentalization, its parameter updates would have to be at least 5 OOMs smaller than parameter updates to conventional language models (assuming that X is 10MB). This would mean matching the performance of trillion parameter models on economically useful tasks using a 10M parameter recursive model.

A.5. Zeroth-order methods

As an alternative to loading parameter updates as inputs to compartments, a compact transcript could be loaded from which the parameter updates could be recomputed inside that compartment. MeZO is the cleanest example: it performs zeroth-order optimization in which each gradient step is replaced by a finite-difference estimate along a pseudo-random direction (Malladi et al., 2024). Because the direction is regenerated from a shared seed, only the scalar step coefficient needs to be stored per update. Malladi et al. (2024) report that for a 66B model, the full training trajectory can be represented as a seed plus $20{,}000 \times 2$ bytes, under 0.1 MB, well below X.

This makes MeZO a potential method for circumventing X, but it ultimately does not seem useful for making training compartmentalized. First, reconstruction requires replaying the full optimization trajectory within a compartment. Each MeZO step requires two forward passes on the full model (one at each side of the finite-difference estimate), so 20,000 steps is 40,000 forward passes, which would dramatically exceed C. Second, the transcript grows linearly with step

count: at 2 bytes per step, 125K steps fills a 250 KB budget and 1M steps fills a 2 MB budget, bounding the length of training runs that can be resumed. Third, MeZO has been evaluated primarily on narrow downstream tasks, and no published work has demonstrated that zeroth-order methods achieve competitive performance on broad post-training distributions.

B. Compressibility of code

Code is unusually structured, and can be more compressible than generic text as a result. If the developer primarily wants to run coding inference tasks, we could shrink X by compressing inputs, achieving a more favorable $\frac{\mathrm{com}_{\mathrm{train}}}{\mathrm{com}_{\mathrm{input}}}$ ratio. We consider some approaches that could enable large compression gains for code and related data.

Committing large codebases. Because we don't impose constraints on the size of committed information, a developer could simply commit codebases they want their AIs to work on. Future inference inputs can then reference large amounts of code using tiny amounts of uncommitted information. One such input could be: in repository R at commit h, inspect file f, and fix failing test t. A more realistic version of this prompt could be tiny, perhaps less than 50 bytes, even though the model retrieves and reasons about a large fraction of the repository. Structured data related to the committed codebase could also be highly compressible. For example, this would enable using diffs against a committed codebase to compress changes. Boilerplate that is expected to be used across many codebases could also be committed. Diffs against committed boilerplate could enable writing entirely new large codebases without violating X.

Concise rewriting. Rinberg et al. (2026) use succinct rewriting to lossily compress generic text. A similar approach may work for losslessly compressing code. Before giving some code as an inference input, it could be rewritten to be smaller but verified to behave identically, making the compression effectively lossless. This code could be written in an almost obfuscated "code golf" manner, and perhaps later "decompressed" to a more readable state after being loaded as an input.

Bootstrapping code from tiny specs. Code may sometimes be more concise to describe in natural language. A developer could first write a lengthy spec, and then use the succinct rewriting approach from Rinberg et al. (2026) to lossily compress the spec. This compressed spec could be given as input, and a summary of the developer's AI attempt at writing code compliant with the spec could be emitted as an inference output, using which the lossily compressed spec could be revised, enabling an interactive protocol for lossily compressing code.

C. Periodic erasure and residual state

Pod-PoComp treats a pod's computation between two consecutive erasures as one compartment. The gateway digests account for the bits that cross pod boundaries, but by themselves they do not prevent a pod from carrying learned state across a long sequence of audit epochs. The role of periodic erasure is to ensure that any state crossing an erasure boundary is either (i) committed in I_0, (ii) observed as later gateway ingress, or (iii) charged to the residual-state bound R. This appendix specifies the erasure check that supplies the memory-erasure component of the witness w_{pod} for the relation R_{pod}.

The motivating failure mode is that a pod could use local memory to accumulate parameter updates over many intervals and emit those updates only after the fact. Erasure breaks this dependence across intervals: any uncommitted information produced before t_i that is needed after t_i must either leave and re-enter through a gateway, or survive in state that is explicitly counted by R.

C.1. Protocol-level erasure check

Schedule and targets. Before the monitoring period, the developer and auditor fix the same public erasure schedule

$$0 = t_0 < t_1 < \cdots < t_m = T$$

used by Pod-PoComp. For each pod p and erasure boundary t_i, they also fix an erasure target $\mathcal{W}_{p,i}$: a manifest of memory regions that must be overwritten or otherwise invalidated at t_i. The target should cover all state whose persistence could allow information produced in (t_{i-1}, t_i) to influence computation in (t_i, t_{i+1}) without passing through a gateway. Depending on the deployment, this may include host DRAM, accelerator HBM, local storage inside the monitored pod, DMA queues, caches, registers, allocator metadata, and task-relevant device buffers.

The parties also fix a conservative bound $R_{p,i}$ on the number of uncommitted bits that may survive the erasure at boundary t_i on pod p. This includes state outside $\mathcal{W}_{p,i}$, state inside $\mathcal{W}_{p,i}$ that cannot be reliably overwritten or checked, and any recovery state that remains resident across the erasure. The parameter R used in the Pod-PoComp soundness statement can be instantiated as a uniform upper bound on $R_{p,i}$ over all pods and erasure boundaries. State that is immutable and committed in I_0 need not be charged to R, but mutable firmware, recovery images, or configuration state should be charged unless they are either erased or fixed by an attestation tied to I_0.

Erasure event. At each erasure boundary t_i, the gateway, runs an erasure event for each pod p:

1. It cuts off ordinary pod ingress and egress except for

the erasure channel, records the cut-off time, and sends a fresh challenge $\chi_{p,i}$.

2. The pod quiesces workload processes, drains or disables relevant DMA paths, releases task memory where possible, and begins overwriting the target $\mathcal{W}_{p,i}$ with challenge-dependent data.

3. The pod returns an erasure certificate $E_{p,i}$ within a bounded response window. The certificate binds the pod identity, the erasure boundary t_i, the challenge $\chi_{p,i}$, the target manifest $\mathcal{W}_{p,i}$, the response time, the claimed coverage, and the bound $R_{p,i}$.

4. The gateway records the certificate, or a content hash of the certificate together with any public metadata needed for verification, as part of the epoch data used by R_{pod}.

The overwrite procedure must force the pod to use the targeted memory during the challenge window. For example, the challenge server may stream unpredictable data of total length at least $|\mathcal{W}_{p,i}|$ and later sample locations from that stream. If a short seed or pseudorandom generator is used instead, the construction must be memory-hard or otherwise prevent a pod from answering challenges by recomputation while preserving pre-erasure state elsewhere. This condition is part of the erasure assumption, not a consequence of the gateway digests alone.

Validity. An erasure certificate is valid for R_{pod} only if all of the following conditions hold.

1. *Freshness and binding.* The challenge $\chi_{p,i}$ was generated after the ordinary-traffic cut-off for pod p at t_i, and the certificate is bound to $(p, t_i, \mathcal{W}_{p,i})$.

2. *Coverage.* The certificate demonstrates that the agreed target $\mathcal{W}_{p,i}$ was overwritten, invalidated, or otherwise made unavailable to later workload computation, except for regions already included in $R_{p,i}$.

3. *Timing.* The response was produced within the allowed response window. The window should be short enough, relative to the network controls and the challenge size, that the pod cannot fetch missing state from an unobserved restore channel or reconstruct it after seeing the challenge.

4. *No unexplained restore.* Gateway records during a guard interval around t_i must not contain unexplained traffic that would let the pod export task-specific state before the erasure and restore it after the erasure without being charged as gateway-observed ingress. Any post-erasure restoration through a gateway is allowed, but its contents are treated as ingress in the next interval and are charged by $\mathrm{new}_{I_0}(\cdot)$ unless already committed in I_0.

5. *Residual accounting.* Any uncommitted state from before t_i that can affect computation after t_i and is not gateway-observed ingress must be covered by $R_{p,i}$. If this cannot be established, the relevant epoch rejects or the deployment must use a larger residual-state bound.

Bound contributed. A valid erasure does not bound the amount of computation performed inside an interval, and it does not provide a lower bound on utilization. The compute bound still comes from the public capacity bound and erasure schedule, e.g. $C_{\mathrm{pod}} T_{\mathrm{erase}} \leq C$. What the erasure check contributes is a bound on hidden state crossing an erasure boundary. Conditional on the gateway and erasure assumptions, the computation by pod p in (t_i, t_{i+1}) can depend only on I_0, public parameters, gateway-observed ingress during that interval, at most B bits of unrecorded communication over the monitoring period, and at most $R_{p,i}$ bits of surviving state. This is the physical premise behind the Pod-PoComp soundness statement that acceptance gives uncommitted input at most $X + B + R$ and output at most $Y + B + R$.

C.2. Example implementation on GPU clusters

This subsection sketches one possible implementation for an NVL72-style pod. The details are deployment-specific; a real instantiation should derive $\mathcal{W}_{p,i}$, $R_{p,i}$, and the response-time bounds from an audited hardware inventory.

Representative pod. Consider a pod consisting of one or more NVL72-style systems with 36 CPU hosts and 72 Blackwell GPUs, where each host is attached to two GPUs, two CX-7 NICs, and a BF-3 DPU. The relevant state includes host DRAM, GPU HBM, DPU and NIC memory, SRAM caches and registers, device queues, persistent firmware, and any local storage included in the monitored pod. The auditor knows the system configuration before setup, including the size and location of host-connected erasable memory.

Some state may be inaccessible to the host CPU or inappropriate to overwrite during normal operation, such as certain firmware regions, boot ROMs, device microcontroller state, or small recovery images. Such state is handled in one of three ways: it is attested as fixed and included in I_0; it is included in the erasure target $\mathcal{W}_{p,i}$ and checked by the erasure certificate; or it is charged to $R_{p,i}$. As an illustrative order-of-magnitude figure, if approximately $10\,\mathrm{GB}$ per NVL72 cannot be overwritten or verified, then the mutable portion of that state should be included in the residual-state bound.

Phase 1: quiesce and overwrite. The gateway first stops ordinary traffic for the pod. Each host then stops workload

processes, tears down accelerator contexts, drains device queues, disables or fences DMA paths not needed for erasure, and frees unneeded memory. The challenge server supplies a challenge-dependent fill stream large enough to occupy the host-connected erasable memory. The host writes the stream into host DRAM and orchestrates corresponding writes into GPU HBM, DPU memory, NIC memory, and any local storage included in $\mathcal{W}_{p,i}$.

To overwrite memory occupied by the operating system, the host can hand off to a small erasure kernel, for example using a kexec-style handoff. The erasure kernel then overwrites the memory vacated by the normal operating system. Any memory occupied by the erasure kernel, drivers, recovery image, or bootstrapping state is either fixed by I_0 or charged to $R_{p,i}$.

Phase 2: challenge. After the overwrite, the challenge server samples random locations from the fill stream and asks the pod to return small openings at those locations, e.g. $w = 64$ bytes per sampled location. The samples should cover each class of erasable device memory in $\mathcal{W}_{p,i}$. The response deadline is calibrated so that a pod that did not actually store the challenged data cannot retrieve it from another machine or regenerate it after seeing the sampled indices. This is why erasure events across hosts in the same pod should either be coordinated closely in time or run while gateway policy prevents unaccounted pod-to-pod restore traffic.

Phase 3: reset and reload. Once the challenge server verifies the sampled openings, the host resets into a clean boot path and resumes service. The post-erasure software image can be obtained through PXE boot, from an attested recovery image, or from another agreed recovery mechanism. Any code, model weights, data, or configuration loaded after the reset is either committed in I_0 or appears as gateway-observed ingress and is charged by $\mathrm{new}_{I_0}(\cdot)$ in the next interval.

Coordination and side channels. The main implementation risk is restore traffic. A pod should not be able to send state to another pod or external machine before the erasure and retrieve it during or after the challenge except through gateway-observed messages. During the erasure window, gateways should either block non-erasure traffic or record it as ordinary ingress and egress. Any channel not observed by gateways is covered only by the side-channel bound B; if the deployment cannot justify such a bound, the corresponding erasure assumption fails.

Caches, registers, device queues, and memory-coherency effects should be flushed or invalidated where possible. If the hardware cannot expose or clear a region, the conservative treatment is to include its mutable capacity in $R_{p,i}$. Similarly, if local SSDs or other persistent storage remain inside the monitored pod, they must either be erased, fixed by commitment and attestation, or counted as residual state. Because persistent storage can dominate erasure time, inference-oriented deployments may prefer to exclude large local storage from the monitored pod and load necessary data through gateways.

Timing. For an NVL72-style system, the volatile erasable memory is plausibly dominated by GPU HBM and host DRAM. A back-of-the-envelope implementation with roughly $13\,\mathrm{TB}$ of GPU HBM and $17\,\mathrm{TB}$ of host DRAM could complete the fill and challenge phases on the order of tens of minutes if the challenge server can drive the relevant links near line rate and devices erase in parallel. Adding reboot, authentication, and software reload time may put a volatile-memory erasure event on the order of 30 minutes. By contrast, erasing hundreds of terabytes to a petabyte of local persistent storage could take hours. The erasure schedule should therefore be chosen jointly with the hardware configuration and the Pod-PoComp compute constraint $C_{\mathrm{pod}} T_{\mathrm{erase}} \leq C$.

Security interpretation. If the erasure event for every host in a pod passes, then, under the hardware and timing assumptions above, the pod cannot carry more than $R_{p,i}$ bits of uncommitted local state across t_i without using a gateway-observed restore path or an unrecorded channel counted in B. This is precisely the condition needed for Pod-PoComp to treat the computations on either side of t_i as separate compartments. It is not a claim that the pod performed little computation before the erasure; that role is played separately by the public compute-capacity bound and the chosen erasure schedule.

D. Survey of Deterministic ML Processes

This appendix surveys evidence for the assumption made in Section 4.3 that the developer's tasks can be made deterministic. The public record shows that bitwise determinism has been demonstrated in serious distributed ML systems, including some frontier-scale training runs when the hardware/software stack, data order, random state, and kernels are carefully controlled.

End-to-end evidence at scale. The clearest public frontier-scale example is PaLM. The PaLM paper reports a 540B dense Transformer trained on 6144 TPU v4 chips, with the largest model completing training at 255k steps, and explicitly states that the model is "fully bitwise reproducible from any checkpoint." The authors give a concrete restart guarantee: restarting from checkpoint 15,000 must reproduce identical results through step 17,000. They attribute this to a bitwise-deterministic JAX+XLA+T5X modeling

stack and a deterministic dataset pipeline in which the contents of a training batch are a function of the step number (Chowdhery et al., 2023). This is strong evidence that long-running, frontier-scale training can be made deterministic on a fixed stack.

T5X and SeqIO provide supporting stack-level evidence. The T5X/SeqIO paper states that the libraries have been used to train models with hundreds of billions of parameters on multi-terabyte datasets, and presents SeqIO as providing reproducible input and evaluation pipelines (Roberts et al., 2023). This is relevant because PaLM's bitwise checkpoint-restart guarantee depends partly on the same ecosystem.

A newer large-scale example is LongCat-Flash, a 560B-parameter MoE model trained on more than 20T tokens. The LongCat-Flash technical report states that training was completed within 30 days on tens of thousands of accelerators, and that the training infrastructure enforced determinism across computation and communication operators, enabling bitwise-aligned loss values across multiple re-runs of a training step (Meituan LongCat Team, 2025). Because this is a recent technical report rather than a mature systems paper, we treat it as strong but less independently validated evidence than PaLM.

A smaller but methodologically relevant example is Kim et al. (2025), who report that most of their pre-training jobs ran on TPU v4-64 or v4-128 with bitwise determinism for handling preemption and promoting reproducibility. This is not frontier-scale in parameter count, but it is evidence that deterministic checkpointing and restartability are useful in nontrivial long-running pre-training experiments.

Kernel-level sources of overhead. The main implementation difficulty is not randomness in the high-level algorithm; it is floating-point accumulation order. GPU kernels often use atomics, split reductions, dynamic scheduling, or communication overlap that changes the order in which partial sums are accumulated. Because floating-point addition is not associative, these choices can change low-order bits. Enforcing determinism therefore often requires fixing reduction order, avoiding some split-reduction strategies, or introducing deterministic post-reduction kernels.

The resulting overhead can be large in unoptimized implementations. For example, LongCat-Flash reports that a default deterministic ScatterAdd implementation can cause up to a $50\times$ slowdown before being replaced with a hierarchical deterministic reduction, and DASH reports that deterministic FlashAttention-3 backward can reduce throughput by up to 37.9% relative to the non-deterministic version (Meituan LongCat Team, 2025; Qiang et al., 2026). This is consistent with our prototype: the deterministic inference path measured in Appendix **??** reduces output-token throughput by 38–69% relative to an otherwise identical vLLM-default baseline.

However, these large overheads should not be interpreted as lower bounds. They are properties of particular deterministic implementations. LongCat-Flash reports a deterministic FlashAttention-gradient kernel reaching $0.95\times$ the performance of the non-deterministic version, and a deterministic ScatterAdd implementation reaching performance parity after kernel redesign (Meituan LongCat Team, 2025). DeepSeek-V4 similarly reports custom batch-invariant attention and GEMM kernels designed to preserve bitwise behavior with minimal performance overhead, including a dual-kernel attention strategy that confines batch-invariant decoding overhead to be negligible (DeepSeek-AI, 2026). These reports suggest that deterministic execution is often a kernel-engineering constraint rather than a model-level asymptotic cost.

Cross-platform reproducibility. Most large-scale training claims are fixed-stack claims: the same hardware type, same software versions, same compiler behavior, same data order, and same checkpoints. Cross-platform bitwise reproducibility is a stronger requirement. Recent systems such as RepDL and Verde/RepOps target this stronger problem by enforcing fixed floating-point operation order or reproducible operators across heterogeneous hardware (Xie et al., 2025; Arun et al., 2025). These systems are useful evidence that stronger notions of reproducibility are technically possible, but they should be treated as enabling research rather than as evidence that frontier-scale training is already routinely cross-platform bitwise reproducible.

Implications for PoComps. Our survey supports three conclusions. First, deterministic long-running ML computation is feasible on fixed stacks, including at frontier scale. Second, determinism is not the default behavior of arbitrary ML workloads: it requires disciplined control of data order, random state, kernel selection, communication order, checkpointing, and numerical lowering. Third, the performance cost is not a single constant. Naive deterministic implementations may be prohibitively slow, while optimized deterministic kernels can make overhead small on the hot paths they target.

System or source	Scale	Determinism claim	Overhead evidence
PaLM (Chowdhery et al., 2023)	540B dense model; 6144 TPU v4 chips; largest run completed at 255k steps.	Fully bitwise reproducible from any checkpoint; deterministic modeling stack and deterministic dataset pipeline.	No direct determinism-off vs determinism-on overhead is reported.
T5X/SeqIO (Roberts et al., 2023)	Used for models with hundreds of billions of parameters on multi-terabyte datasets.	Reproducible input and evaluation pipelines; supports deterministic data access useful for restartability.	No direct overhead number is reported.
LongCat-Flash (Meituan LongCat Team, 2025)	560B MoE model; more than 20T tokens; tens of thousands of accelerators; training completed within 30 days.	Determinism enforced across computation and communication operators; bitwise-aligned losses across multiple re-runs of training steps.	Reports optimized deterministic FlashAttention-gradient kernel at $0.95\times$ non-deterministic performance, deterministic ScatterAdd at performance parity, and default sequential ScatterAdd paths causing up to $50\times$ slowdown before optimization.
DeepSeek-V4 (DeepSeek-AI, 2026)	1.6T-total-parameter MoE model family with pre-training on more than 32T tokens.	Reports high-performance batch-invariant and deterministic kernel libraries for bitwise alignment across pre-training, post-training, and inference.	Reports batch-invariant attention overhead as negligible after custom dual-kernel design; notes that avoiding split-K GEMM can degrade performance, but that custom DeepGEMM optimizations match or exceed split-K performance in most major scenarios.
DASH (Qiang et al., 2026)	Attention backward kernels evaluated on NVIDIA H800 GPUs.	Deterministic attention backward pass gives bitwise-identical gradients across repeated runs.	Reports that deterministic FlashAttention-3 backward can incur up to a 37.9% throughput reduction relative to the non-deterministic counterpart; DASH improves the deterministic baseline by up to $1.28\times$.
UniEP (Zheng et al., 2026)	Expert-parallel MoE MegaKernel evaluated on Hopper GPU clusters.	Deterministic token mapping preserves accumulation order and yields bitwise-identical forward and backward results relative to a serial reference.	Reports that maintaining bitwise equivalence imposes a measurable backward-pass cost, while the full optimized system still achieves $1.03\times$–$1.38\times$ speedups over prior systems.
Framework APIs (TensorFlow Developers, 2024; PyTorch Contributors, 2025)	General-purpose ML frameworks.	TensorFlow exposes deterministic op execution under fixed inputs and hardware; PyTorch exposes deterministic algorithms but warns that complete reproducibility is not guaranteed across releases, commits, or platforms.	TensorFlow documentation explicitly notes that deterministic ops may run slower; PyTorch documentation frames determinism as scoped to a specific platform, device, and release.

Table 2. Public evidence on deterministic ML processes and deterministic kernel overhead. The strongest end-to-end evidence is checkpoint-restart determinism in PaLM. The strongest recent large-MoE evidence is LongCat-Flash, though it is a technical report. Kernel-level papers show why overhead varies: naive deterministic reductions can be expensive, but specialized scheduling and deterministic accumulation schemes can recover much of the performance.