Decentralized Confidential Machine Learning

Illia Polosukhin, Alex Skidanov
NEAR AI
illia@near.ai, alex@near.ai

Introduction

Large language models are becoming more important in the everyday lives of millions of people and for an ever-increasing number of use cases. Many of these use cases require users to share their personal data with the service that runs the model. For proprietary models such as ChatGPT, that means sharing personal data with the model creator. For open-source models, this means sharing the data with the service that hosts the model. For smaller models,users may opt to host models locally, but for frontier models, this is not viable for the majority of users.

At the same time, for a model developer, there are two ways to publish a model: to keep the weights private and publish it as an API endpoint, or to publish the weights. In the latter case, monetizing the model becomes problematic because users can choose to run the model locally, or can use it using third-party providers for hosting and avoid paying the developer.

In this paper, we present a decentralized machine learning cloud that allows model developers to publish their models in such a way that anyone can get a usable deployment of the model without permission from the developer, but no one can use the model without paying the developer, and all the deployments of the model are such that neither model hosts nor the model developer can see the data of the users using the model, which includes inputs, outputs, and any intermediate values.

The paper is structured by iteratively building the components of this decentralized machine learning cloud, including:

- Private inference. The ability for a user to use a remotely hosted machine learning model in a way that makes the user confident that neither the host nor the model supplier can see the user's data.
- Verifiable open-source models. A pair of source code and model weights such that the user of the weights can be certain that the weights are, in fact, produced by the code provided.
- Decentralized monetizable models. An approach to distributing models such that anyone can get a usable deployment of the model from any

other participant of the decentralized network, and yet no one can use the model without paying the model supplier.

- Community-owned models. An approach to training models in which a group of participants that do not necessarily trust each other pools resources together, and then train a model in such a way that it can be deployed to the decentralized network that does not allow usage without payment, and the payments are distributed back to the participants that funded the model training.

1 Trusted Execution Environments

Modern processors support a concept of Trusted Execution Environments (TEEs), such as Intel TDX, AMD SEV and ARM CCA. TEEs allow one entity (Alice) to run some code on the machine of another entity (Bob) in such a way that (a) Alice is certain that the computation carried out is in fact the computation she expected to be carried out, and (b) Bob cannot see any data passed by Alice to the computation, nor the results of the computation.

Starting with H100s, NVIDIA GPUs also support Trusted Execution Environment, as long as the CPU on the machine supports TEEs (such as Gen5 Intel processors). These also allow machine learning workloads that move data in and out of GPUs to be executed in such a way that the owner of the hardware cannot eavesdrop on the user data. Independent benchmarks Zhu et al., 2024 show that the overhead of running inside the TEEs is less than 7%. It is expected to be even lower on the Blackwell GPUs.

In this document, we utilize TEEs via the following abstraction: we craft a special open source confidential virtual machine (CVM) that the hardware owner (Bob) deploys on their TEE-enabled hardware, which allows users (Alice) to establish a TLS connection to the CVM in a way that Alice can in fact be certain that the other end of the connection is served by CVM that runs inside a TEE, and that there's no man in the middle that can eavesdrop on the connection. In other words, Alice is certain that only the code of the CVM and no other code participates in handling her requests, and that neither Bob, nor the creators of the CVM, nor any other entity can see the data being sent or received by Alice, nor, for that matter, any intermediate results of the computation.

The CVM then allows Alice to spin up an arbitrary Docker container inside the CVM, and run it.

It further allows a third party (Charlie) to prepare and publish a Docker container with some computation spin up a CVM on Bob's hardware, deploy her Docker container there, and then have Alice establish a TLS connection with the CVM and be certain that the service she is interfacing with is in fact operated by the Docker container published by Charlie.

It is worth mentioning that TEEs rely on the hardware manufacturers not to be malicious. Code running inside an Intel TDX enclave could be compromised if Intel is compromised, and similarly, the code running inside H100 TEEs could be compromised if NVIDIA is compromised.

2 Private Inference

Inference can be considered as the most direct application of TEE technology to machine learning use cases. While quantization and other techniques made it possible to run many models locally, it is still the case that specialized, expensive hardware is necessary to serve the best models at fast speeds, and that users of such models need to use models hosted outside of their local environments.

When users interface with systems deployed by centralized entities, such as OpenAI or Anthropic, they voluntarily give away all their data to such centralized entities. However, after decades of conditioning, users of the modern web are generally content to share their data with big centralized players, hoping that such centralized players have good security measures in place and assuming that the probability of the data being misused, leaked, or otherwise used against the user is low.

Now consider a decentralized inference solution. This would naturally consist of hardware provided by various entities, and any computation carried out on such hardware would be seen by the hardware providers. In contrast to the situation above with large centralized players, users have no reason to believe that hardware providers will not intentionally or unintentionally store and misuse their data.

There are three general approaches to getting around this problem:

1. Multi-party computation (MPC). The general idea is to serve the model in such a way that the participation of multiple parties is necessary to carry out the computation, and no user data can be recovered unless some percentage of the parties collude. For the purposes of private computation we do not consider MPC alone to be a sufficient measure, since the collusion of the parties is undetectable, and the parties have no incentive not to collude. MPC also comes with expensive computational overhead, e.g. 5 minutes per token for a 7B parameter model as described in (Dong et al., 2023).

2. Homomorphic encryption. A homomorphic encryption scheme is an encryption scheme that allows carrying out computation on top of encrypted data in such a way that the result is also encrypted. Homomorphic Encryption would be the most desirable and principled approach to private inference, but unfortunately modern schemes are still prohibitively expensive in terms of their computational overhead.

3. Running inside Trusted Execution Environments.

Returning to the abstraction outlined in section 1, a Docker container is prepared that runs an inference server[1] and exposes an OpenAI-compatible

[1] An implementation of such a Docker container using VLLM can be found here `https://github.com/nearai/private-ml-sdk`

API. Then a network of participants running TEE-enabled hardware spins up CVMs with this Docker container. Users of the network can establish a TLS connection with the CVM hosted on a machine of any such participant, and use it to carry out inference workload, fully utilizing the GPUs on the machine.

According to the TEE abstraction we use described in section 1, after establishing the TLS connection to the CVM, the user is certain that (a) the other end of the connection is in fact running inside the TEE, (b) the code that handles the connection is in fact the VLLM docker container, and nothing else. Since the TLS connection is encrypted, as long as the Docker container itself does not contain any logic that leaks the prompts and completions, the user can be certain that all their data is visible only to them, and neither the owner of the machine nor the developer of the Docker container can see their data.

3 Verifiable Inference

In addition to the ability to hide the data coming in and out of the inference server, it is also desirable to be certain that the completions are in fact coming from a machine learning model, and from the model the user expects to be used. For example, if the inference provider claims that the completions are done with DeepSeek R1 (DeepSeek-AI et al., 2025), users want to be certain that they were not done instead with a smaller model.

A way to achieve this is to extend the API from section 2 with the following:

- `load_model(path)`. Loads a model from a local path for inference, and computes its hash *modelH*. The completion API uses the model loaded with `load_model`.
- The completion API is extended including *modelH* with each completion.

In this way, the user can compare the *modelH* returned with each completion to the hash of the model they expected to be used. Since the user is using a TLS connection for which they are certain that the other side of the connection is handled by this particular Docker container running inside the TEE, they can be certain that the owner of the hardware cannot tamper with the *modelH* returned with the completion.

4 Decentralized Encrypted Models

Say that a particular entity, Charlie, spent considerable resources to build a new model. The process comprised preparing the dataset, writing software for the training loop, and actually running the training run. The output is the set of weights that Charlie has locally. She now wants the model to be available on a decentralized network—in other words, she wants to ensure that anyone can use the model without depending on Charlie as the single point of failure. In section 7 we will also discuss how to remove Charlie as the single point of failure during training and data preparation.

Naturally, Charlie can just publish the weights. Although this fully solves the problem of Charlie being the single point of failure, it also prevents Charlie from being able to monetize the model. Instead, we want such a construction in which no one but Charlie ever sees the weights, but anyone can get a usable deployment of the model without involving Charlie, and in such a way that Charlie cannot prevent it from happening.

With TEEs, Charlie can achieve this by encrypting the model, and creating a special Docker container that will run inside the CVMs such that the model can only be decrypted inside that Docker container.

Specifically, Charlie first creates a key pair (sk, pk) and encrypts the model with pk. She then prepares a Docker container that is a fork of the container described in section 3, with the following modifications to the API:

- `set_sk(sk)`. The container stores sk in-memory, but sk is not baked into the Docker container itself, lest it could be extracted by anyone inspecting it. Instead, the sk needs to be supplied to the virtual machine after it is launched. `set_sk` is the way to supply it. Initially, only Charlie knows the sk, and she seeds the network by calling `set_sk` on one of the instances.

- `request_sk(host, port)`. An instance of the virtual machine that does not yet have sk can request it from another instance that does. For that, the owner of the former establishes the TLS connection with the latter and calls this method. In other words, the `request_sk` is called on the machine that already has sk, and the arguments are the host and port of the machine that wants to obtain the sk. The handler of the `request_sk` establishes the TLS connection with the address provided, and calls `set_sk` with the sk. Note that due to guarantees of the abstraction presented in section 1, the machine establishing the connection can be certain that the code that handles the `set_sk` method is running the very same Docker container as the caller, inside the TEE.

- `load_model(path)`. Loads an encrypted model from a local file, and decrypts it with the sk. The decrypted weights can now be used for inference, but never escape the TEE.

The construction above ensures that (a) any participant of the network can fetch sk from any other participant of the network that already has it, and (b) neither sk nor the model weights are ever seen outside the TEE by anyone but Charlie.

This in itself does not solve the problem of Charlie monetizing the model. We cover one approach that can introduce monetization on top of such a closed-weight decentralized distribution in section 6.

5 Verifiable Encrypted-Weights Models

Models such as LLaMa (Touvron et al., 2023) are often referred to as "open-source," even though no source code used to train them, nor to prepare their datasets, is actually open. The only "open" part of these models is the model weights. In some circles such models are referred to as "open-weight" models, to emphasize the fact that only the weights, and not the source code, are open. "Open-source" then would be a model for which all the code used to train it, and to prepare the dataset on which it is trained, are open. OLMo, by the Allen Institute for Artificial Intelligence (Groeneveld et al., 2024), is an example of a truly open-source model. However, even in the case of OLMo, there is no way to verify that the code published is what was used to generate the weights provided, without incurring a cost equal to the cost of the original OLMo training run.

After LLaMa published their first model, many open-source efforts emerged to fine-tune, or do other forms of post-training, on top of the weights. However, any effort that wanted to make progress at the pre-training step, such as Mistral, had to fully re-implement the training loop, since the training loop of LLaMa is not open source. If it were open source, the progress on pre-training naturally would have been much faster.

Consider an open-weight model with open source code. If one were certain that the source code published was in fact what was used to generate the model, then they would be able to verify whether there were any attempts to introduce biases or back doors into the model.

Verifying that the published weights are the result of a particular computation is possible if the computation itself was carried out inside of the TEEs. We will expand on the idea from section 4.

In this section, we consider a simplified scenario in which the entire training pipeline (data preparation, pre-training, and post-training) are executed in one go on a single machine. But this construction generalizes to practical approaches in which those stages are executed separately, and on bigger clusters.

For the simplified scenario, the Docker container from section 4 is extended in the following way. Charlie first chooses the raw data on which she plans to train her model, for example a subset of Common Crawl (Rana, 2010), and/or a dump of public GitHub repositories. She then computes the hash of the raw data $rawH$, which she bakes into the Docker container. The API of the Docker is then extended with:

- `load_data(path)`. Loads data from a local file, and verifies its hash is $rawH$.
- `train_and_encrypt(pk)`. Trains the model from the raw data. Fails if the data was not previously loaded with `load_data`. Computes the hash of the resulting model $modelH$. Encrypts the model with pk and saves the encrypted weights on disk. The encrypted model is available outside of the TEE.
- `load_encrypted_model(path, seed_host, seed_port)`. Loads a previously trained encrypted model from a local file, decrypts it with sk. Then establishes a TLS connection to the `seed_host:seed_port`, and calls to `get_model_h`. Ensures that the loaded model hash matches provided. Lo-

cally sets *modelH* to be able to seed it via `get_model_h` to other participants.

- `get_model_h()`. Returns *modelH*.

For the original training run, Charlie (the entity training the model) prepares all the data, computes its hash, and prepares the Docker container with the data hash baked in. She then calls to `load_data`. The `load_data` would not succeed if the loaded data does not hash to *rawH*.

Charlie then calls to `train_and_encrypt`. Once the training commences, the encrypted model is on Charlie's disk. She can decrypt and inspect it, but no one else can. Charlie publishes the encrypted model and the host and port of her machine. Now other participants can fetch *sk* and *modelH* from Charlie's machine or other participants who have already fetched it.

Now, say Alice wants to use the model, but she wants to be certain that the model she uses is in fact the model trained by the logic in the Docker container published by Charlie, on the data that Charlie claims it was trained on. Alice downloads the raw data, computes its hash, and ensures it is equal to *rawH*. She launches her Docker container inside the TEE. Her instance does not yet have either *sk* or *modelH* in it. She connects to some other participant, Bob, who already has those on their instance. Via `request_sk` and `load_encrypted_model`, she fetches both from Bob's instance of the virtual machine into hers.

After a successful call to `load_encrypted_model`, Alice can be certain that the model loaded is in fact trained via the code and using the data published by Charlie. This follows by induction. For *modelH* there are only two ways Bob could have acquired them: either they fetched them from yet another participant, David, or Bob ran `train_and_encrypt` (i.e. Bob is Charlie). Since there is no other way to obtain *modelH*, it must have been obtained initially via a call to `train_and_encrypt`. This guarantees that the model was, in fact, trained using the code provided. In turn, `train_and_encrypt` would have failed if it had not been preceded by a successful call to `load_data`, which in turn would fail if the data loaded did not hash to *rawH*. This guarantees that the model was trained on the raw data provided by Charlie.

This allows the creation of open-source, encrypted-weights decentralized models, such that anyone can run the model locally and be certain that the model is created in the way the creators claim, while preventing anyone from seeing the weights.

This approach naturally extends to open-source open-weights models with the same guarantees. For that, it is sufficient to remove the logic that encrypts the model but retain the logic that computes and exposes h.

6 Monetizable Deployments

In section 5 we discussed how someone could publish an open-source encrypted-weights model. The motivation was that if the model is open-weights, there is no way to track and monetize the usage of the model. With the distribution mechanism explained in section 4, we can now address this problem.

Specifically, we want the following construction. One entity (Charlie) has created a model they want to host in a decentralized fashion, but they want to be paid whenever the model is used. Another entity (Bob) wants to provide the hardware to host the model, and they also want some payment per invocation of the model in order to recuperate the cost of the hardware. Finally, a third entity (Alice) wants to use the model.

Any two of those entities can coincide. The creator of the model can also be its host (i.e. Charlie and Bob can be the same entity); while the user of the model must also be able to choose to host the model on their own (i.e. Bob and Alice can be the same entity). Moreover, it is required that the user of the model is able to host it in an environment completely isolated from the Internet.

We propose the following model: the code in the Docker container that is used to distribute and host the model (and possibly was used to train it) has the following logic to perform inference:

1. Charlie deploys a smart contract for the model that allows Bob to register themselves as a model provider. Charlie configures the cost of using the model, and Bob configures the extra cost of using it via their deployment.

2. Before an inference session can begin, a random number (*challenge*) is generated inside the TEE and is given to Alice.

3. Alice uses the *challenge* to open a uni-directional payment channel with Bob and Charlie.

4. Alice provides a light client proof of the transaction that opened the payment channel to the code running inside the TEE.

5. Now whenever Alice wants to do inference, she sends a payment via the payment channel that covers the cost specified by Charlie and Bob per some number of tokens.

For the model above, the Docker container from section 5 is extended with

- `get_challenge()`. Generates a random challenge locally, remembers it, and returns.

- `establish_payment_channel(tx, light_client_proof)`.
  The arguments are a transaction on NEAR that establishes the payment channel with the challenge provided, and a light client proof of the transaction inclusion. The challenge ensures that the same payment channel is not reused with multiple instances of the virtual machine.

- `request_channel_closure()`. Returns a signed approval to close the payment channel that can be submitted to NEAR. Locally removes all the information about the payment channel, i.e. no completions can be done until another payment channel is established.

The contract with which the payment channel must be established is baked into the Docker container by Charlie.

Furthermore, the completion endpoint only works if the payment channel was previously opened and if it is not saturated yet. Each call to the completion API charges the payment channel.

The actual payment to Bob and Charlie happens when Alice closes the payment channel. The exact protocol of closing it needs to consider several scenarios.

Bob's hardware crashes after the channel opening. The first scenario to consider is the following: Alice opens the payment channel, and locks some amount of NEAR in it. Immediately after the channel opening, Bob's hardware intentionally or unintentionally goes down. Thus, there is no way to obtain a message with channel closure approval from the other side of the payment channel. Thus, it must either be the case that Alice carries the risk of losing the money locked in case of hardware failure, or it must be possible to close the channel by Alice without the approval from the service running inside the TEE.

Bob and Alice are the same entity. The second scenario is Alice hosting the model herself. She deploys the model inside of her TEE-enabled machine, opens up the state channel, then disconnects the machine from the internet. She then carries out multiple completions from the model, saturating the payment channel, and then, assuming she can close it without the approval of the service running inside of the TEE, closes the payment channel at the initial state, i.e. as if she did not spend any money inside of the TEE[2]. This way Alice can use the model without paying Charlie, which completely defeats the purpose of the construction.

The two scenarios presented above present a challenge. Without further engineering, if Alice can close the channel without the approval of the service running inside of the TEE, even with a challenge period, she can use the model without paying Charlie. If she cannot, Bob can cause Alice to lose her money without providing the service to her.

This problem can be solved in the following way: when a payment channel is open, both Alice and Bob lock the same amount of money. In the happy case Alice uses the model, pays for it over the payment channel, then requests the approval to close the channel, obtains it, and closes without involving Bob. Bob's locked money gets returned to him on closure. In the unhappy case she initiates the channel closure without the approval, which gives Bob time to challenge it. To challenge the closure, Bob uses an endpoint in the service running inside the TEE that receives the light client proof of the Alice's closure request, and then returns a transaction to be submitted to NEAR with the closure approval. The channel is then closed, Alice gets reimbursed the remainder of her payment channel, and Bob gets fully reimbursed. If Bob fails to respond to the closure request, Alice gets fully reimbursed, and Bob's locked money are fully sent to

[2]We consider the ability to use the model offline after opening the state channel a requirement. However, even if this requirement is lifted, Alice can still execute the attack presented, though it becomes more involved.

Charlie. This way even if Alice used up the whole payment channel, Charlie gets paid for each completion.

6.1 Cloning VMs to reuse payment channels

The solution makes an assumption that there is no way to force the service running inside the TEE to provide an approval to close the channel at a non-final state. While it is generally impossible to tamper with the computation that happens inside of the TEE, for obtaining an approval for an earlier state of the state channel one does not need to tamper with the execution, it is sufficient to be able to clone or snapshot the virtual machine running inside of the TEE. Consider the following scenario: Alice opens up the state channel with Bob's machine; then Bob snapshots the state of the memory. It is encrypted, but Bob does not need to know what is in the memory, he only needs to be able to recover in to exact same state. Alice then proceeds to fully saturate the payment channel by using the model; Bob then resets the memory to the state that was snapshotted. The state of the TEE is now the same as it was right after state channel opening. Alice can either use up the payment channel again, or close it and get her money back.

At least Intel TDX has a solution for this problem: TDX has a concept of non-resettable counters, meaning that the code running inside of the TDX has access to a counter that the owner of the hardware cannot rewind. With such a counter the payment channel can be engineered to detect attempts to clone or roll-back the state of the virtual machine.

6.2 Faster payment channel closures

There is a problem with the above approach to enabling Alice to close the payment channel without obtaining approval from the service running inside the TEE. Both Alice and Bob can grieve each other by forcing the challenge (Alice by initiating it; Bob by shutting down his machine).

With the mechanism described in section 8, there is a third possible option for closing the channel, besides having an explicit approval, or initiating a long challenge period. Specifically, if Bob is part of a Proof of Response network (see section 8), the payment channel can be closed if Alice can prove that Bob was disconnected. This way Alice can repeatedly request the approval to close the payment channel via the Proof of Response network, and eventually either get the response, or end up disconnecting Bob from the network, which enables her to immediately close the payment channel (and use Bob's locked funds to pay Charlie).

7 Community-Owned Models

Previous sections explored decentralized distribution and monetization of a model. In those sections, the model was trained by a single entity, which in the modern world is usually the case. However, this requires a single entity to be able to gather sufficient resources to pull off training a model that will be competitive enough for people to want to pay for it.

For state-of-the-art models, this often means tens of millions of dollars per run. In this section, we explore a modification to the mechanism that allows a set of participants to pool funds together, train the model, and share the profits in a decentralized manner.

Suppose that Charlie and David want to pool resources together to train a model and then to share the profits from its deployment. As section 6 shows, once they have the model, they will be able to deploy it in such a way that it can be monetized.

The challenge here is that if Charlie and David do not trust each other, then if either of them has access to the weights, that entity would be able to encrypt and deploy those weights so that only they get paid for the inference. Therefore, ideally the model must be trained from the beginning in such a way that even David and Charlie never see the weights, and the only way to use the weights is to deploy it as presented in section 6, with both Charlie and David as beneficiaries.

We will build upon the construction from section 5. The Docker container is prepared to have logic to train the model, and then to deploy it in a decentralized fashion. The difference with this construction is that the entity training the model no longer supplies the key to encrypt the model. Instead, the key pair (pk, sk) is generated inside of the TEE, before the training. Charlie and David deploy a special contract that tracks the ownership structure. When training starts, either Charlie or David provides the light client proof of the contract, and then training commences. During training, the weights inside the TEE are not encrypted, but once training is completed, or during the intermediate checkpoints, the model is always encrypted before hitting the disk. At no point during or *after* training are the unencrypted weights seen by any entity, including David and Charlie.

Once training commences, the model can be distributed in a decentralized fashion via the protocol explained in section 4 to other machines running exactly the same Docker container. When the model decryption key sk is shared between two machines, they also share the hash of the model, as well as the smart contract tracking the ownership structure.

For monetization in this case, whenever a payment channel is closed and payment is made to the model owner, it is now made to the smart contract that tracks the ownership structure, and the smart contract distributes the funds to all the contributors.

7.1 Training Process

Training a model requires a very large set of machines. We envision the full solution to work in the following way:

1. David and Charlie deploy a smart contract and deposit money to cover

the cost of training. They further configure the smart contract to indicate how much they are willing to pay per epoch per machine and the minimum configuration of machines that can participate.

2. A participant who has hardware, Bob, deploys the Docker container prepared by David and Charlie into a TEE on their machine. The logic inside the TEE verifies the machine configuration and prepares a quote to submit to the smart contract. Bob registers on the smart contract with their host and port, and joins the training run.

3. Once per epoch, the training logic pings the smart contract with a proof of being run inside the TEE via the Docker container provided by Charlie and David, and Bob gets paid for one epoch of work.

4. The actual training loop implements DiLoCo or a similar mechanism that enables training a large model efficiently on a geographically distributed set of machines.

Note that it is important that the various participants in the network can communicate with one another. Training requires accumulating gradients, and if Bob is offline after computing their epoch of work, they are not useful to the overall effort. We propose to use Proof of Response (see section 8) to accumulate gradients, thus ensuring that if one or more participants is offline, that can be proven on the blockchain, and the participant(s) can be removed from the training run.

8 Proof of Response

This section briefly explains a mechanism called Proof of Response, which comprises a network of machines such that if one participant, Alice, wants to request something from another participant, Bob, then either Alice will receive the response from Bob, or at least one edge on the path from Alice to Bob will be severed. Since Alice can repeatedly try to request the information, ultimately Alice will either receive the response or will disconnect Bob from the network, and have a proof that Bob is no longer part of the network.

The full explanation of the mechanism can be found in "Proof of Response" (Polosukhin and Skidanov, 2025); we will include only a high-level discussion of the idea in this document.

For each edge in the network, the two nodes that are incident to the edge have a state channel open. Alice chooses a path via the network between herself and Bob, and sends the message to the first node on the path. Then for any two consecutive nodes P and Q on the path, as the message passes between them, the following must happen:

1. Within some predefined time δ, Q returns the response to P;

2. Within δ, Q sends to P a proof that one of the edges on the path is no longer present;

3. Neither has happened, and Q streamed to P a penalty payment proportional to time passed beyond δ.

This property is maintained by induction. If Q then routes a message to R, then of R has responded to Q in time with either of the two responses (actual response, or a proof of broken edge), Q routes it back to P. If R streams payment to Q for delay, Q streams it back to P. If R does neither, Q can choose to wait for some time and pay to P out of pocket, but ultimately they break the edge with R and send the proof to P.

9 Decentralized Confidential Machine Learning Cloud

With all the building blocks described in the previous sections, the full system comprises a network of machines, each with a TEE-enabled processor and GPUs. The nodes are connected to a network with a topology tracked on the blockchain through Proof of Response (see Section 8).

There is a smart contract that tracks commitments of such nodes to run particular machine learning workloads on their hardware. The smart contract exposes the following endpoint:

`commit(docker_hash, end_time, host, port)`

The end-point is payable, and locks the attached NEAR in the contract until the `endtime`.

That states that the operator of the node at `host:port` commits to running the docker container with `docker_hash` on their TEE-enabled machine.

Generally, it is unreasonable to bake in a model into a docker container, but the docker container can be designed in such a way that if for a particular hash h model weights that hash to h are not provided when it starts, it immediately shuts down, thus ensuring that if a user can interface with an instance of such docker container, it has access to the weights.

All the API end-points of the docker containers are designed in such a way that each response is accompanied with a proof that the response was generated inside the TEE, and that the TEE was in fact running the docker container with `docker_hash`.

While Proof of Response itself only ensures that *some* response signed by the service provider was received, it does no validation of the nature of the response. Thus, the contract further has two extra end-points:

`show_offline(host, port)`

That in turn pings the proof of response contract to confirm that `host:port` is offline, and if so, starts gradually slashing their stake, and

`show_invalid_response(host, port, request, response)`

That verifies that the response is signed by the service provider, but does not have a valid proof of being executed inside the TEE by code running in the correct docker. If the verification fails, a percentage of the stake is slashed.

Thus, if the service provider has commits to running a particular docker image for a particular duration, they either in fact run it, and are online, or they get slashed.

Such a decentralized machine learning cloud can be used for all the use cases described above. In particular,

- A node can commit to serving a VLLM docker with a particular model in it for a particular duration. Users can expect that they can either get reliable *confidential* inference at the end-point specified, or that the node will get slashed. Note that the node can at best violate, and be slashed for the violation of, liveness, it cannot violate confidentiality.

- A user can enter an agreement with a particular node to host their own docker container. The node operator ensures that the docker container is properly designed (that all the API end-points in fact annotate their responses with the correct proofs), and commits to running it.

Generally, any scenario in which one entity needs an AI workload to run in a reliable and confidential manner, and another entity has compute to run it on, is supported by the construction outlined above.

10 Conclusion

With the anticipated dominance of artificial intelligence, the ability to use the models and tools built upon them in a verifiable and confidential manner will be crucially important. In this document, we explored applications of Trusted Execution Environments to solving such problems. We further present a construction of a machine learning cloud that enables running arbitrary workloads in a reliable and confidential manner.

References

DeepSeek-AI, Daya Guo, Dejian Yang, et al. (2025). *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning.* arXiv: `2501.12948 [cs.CL]`. URL: `https://arxiv.org/abs/2501.12948`.

Dong, Ye, Wen-jie Lu, Yancheng Zheng, et al. (2023). *PUMA: Secure Inference of LLaMA-7B in Five Minutes.* arXiv: `2307.12533 [cs.CR]`. URL: `https://arxiv.org/abs/2307.12533`.

Groeneveld, Dirk, Iz Beltagy, Pete Walsh, et al. (2024). "OLMo: Accelerating the Science of Language Models". In: *Preprint*.

Polosukhin, Illia and Alex Skidanov (2025). *Proof of Response.* URL: `https://arxiv.org/abs/2502.10637`.

Rana, Ahad (2010). *Common Crawl – Building an open web-scale crawl using Hadoop.* URL: `https://www.slideshare.net/hadoopusergroup/common-crawlpresentation`.

Touvron, Hugo, Thibaut Lavril, Gautier Izacard, et al. (2023). *LLaMA: Open and Efficient Foundation Language Models*. arXiv: `2302.13971 [cs.CL]`. URL: `https://arxiv.org/abs/2302.13971`.

Zhu, Jianwei, Hang Yin, Peng Deng, et al. (2024). *Confidential Computing on NVIDIA Hopper GPUs: A Performance Benchmark Study*. arXiv: `2409.03992 [cs.DC]`. URL: `https://arxiv.org/abs/2409.03992`.