Ledgera: The Modular Distributed Ledger

Scope of this White Paper.
This white paper describes Ledgera's architectural vision and intended execution model. It is not a release-specific feature list: the exact set of implemented capabilities, limitations, and deployment assumptions is documented in the corresponding release notes.

Authorship and intellectual property notice.
Ledgera is a technology developed by the CEA LICIA team. This white paper was produced within the CEA LICIA team, with Antonella Del Pozzo and Sara Tucci as main authors. The formal specification of Ledgera is provided in the accompanying Yellow Paper, whose main authors are Erwan Mahe and Antonella Del Pozzo.

Abstract

Ledgera is a distributed ledger designed for organizations and devices that need to collaborate without relying on a central trusted authority. To support industrial applications with compute-intensive workloads and large datasets, Ledgera adopts a modular architecture that relies on a flexible pool of software nodes called Ledgents, that can play distinct roles—Executors, Provers, Storers, Loggers, and Validators—depending on application needs, able to certify computations through cryptographic proofs. In traditional blockchain architectures, these responsibilities are usually collapsed into the same validator set and executed under a single replicated state machine. Ledgera separates them, allowing applications to rely only on the roles and coordination guarantees they actually need. In its minimal form, Ledgera supports the certified execution of stateless functions: Executors perform the computation, Provers produce the corresponding proofs, and Loggers anchor those proofs in a secure log for auditability purposes. Storers and Validators are introduced only when applications require durable result persistence, proof-carrying composition across executions, or shared-state coordination. By decoupling execution, proof production, storage, anchoring, and validation into separable roles that can be deployed in different ways, Ledgera provides a high degree of modularity without systematically resorting to state machine replication, which remains a major bottleneck in current blockchain architectures. This white paper presents Ledgera's motivation, core design choices, and usage patterns, and illustrates the approach through representative distributed applications including federated learning, multi-party computation, and asset transfer.

1. Introduction

As digital systems become increasingly distributed, organizations and devices must collaborate across administrative and trust boundaries. From industrial IoT and supply chains to cross-organizational analytics, federated learning, and multi-party workflows, these settings require guarantees on data integrity and computation correctness, without relying on a central intermediary.

Blockchains pioneered a compelling approach by providing tamper-evident logs backed by consensus mechanisms. However, blockchains are designed as monolithic state machine replication systems, imposing a single "world-state" model on all applications. Specifically, blockchains provide smart contracts as stateful programs executed under consensus-enforced

total ordering to update a globally replicated world state. This world state is shared across and visible to all applications that interact with these smart contracts. This execution abstraction is often unnecessarily rigid for industrial data and compute workflows. In particular, the requirement to totally order and replicate all state transitions introduces latency and consensus costs, which can become the dominant bottleneck in data and compute-intensive settings and hinder scalability.

More recently, so called “Layer-2” approaches have been explored to off-load blockchains (where the blockchain represent the “Layer-1”) from application computation and storage needs. Layer-2 solutions can be seen as replicating the application logic in a parallel blockchain, typically composed of a smaller number of nodes dedicated to the application, which validate its computation and are equipped with a sequencer responsible for establishing a total order on state updates to be committed to Layer-1. In this way, Layer-2 solutions pursue a separation between the computation of a new state update and its ordering. Yet, in general, it is still total order that is pursued, regardless of whether the application actually requires it.

On the other hand, it is well known that total order is not always needed to ensure application correctness. This is true for several important applications, such as asset transfer [AT16, FS24] or federated learning [FL21], where it is possible to guarantee correctness without relying on state machine replication and its costly overhead. For other applications, such as confidential multi-party computation based on secret sharing, agreement is needed to avoid cases where output shares are produced from input shares that were not produced from the same set of private inputs [MPC93]. Another interesting example is the management of access lists and deny lists for access control, where consensus may be needed if deny lists are required, but not otherwise [AC23]. Ledgera proposes an approach that flexibly adapts to different application needs by isolating two core functionalities: guaranteeing the integrity of a one-shot computation, and ensuring the availability of its result when persistence is required by a client application. These functionalities can then be extended with validation mechanisms to support the integrity and auditability of composed computations issued by multiple clients.

To this end, Ledgera relies on a set of software nodes called **Ledgents** (for Ledgera Agents), each equipped with its own cryptographic identity. In its minimal form, Ledgera relies on Ledgents playing three roles: Executors, Provers, and Loggers. Upon request from a client application, Executors collectively execute an application-declared function, while Provers generate a cryptographic threshold proof certifying the correctness of the result. This proof, called a Ledgera proof of computation, is obtained through a Byzantine quorum protocol [1] and returned to the client application. Loggers then anchor the resulting proof in a secure log, making it immutable, auditable, and reusable across parties. When applications require durable persistence of results, Ledgera augments this minimal form with Storers. In that case, Ledgents cooperate with a persistence layer to ensure durable storage of the result and return a corresponding Ledgera Proof of Storage to the client application.

Ledgera also supports multi-party input verifiable functions in this minimal setting. In this case, Ledgents wait for inputs coming from multiple client applications, run an agreement protocol on the input set, and execute the function, producing the corresponding Ledgera proof. Multi-party

[1] Total-order consensus is a strictly stronger abstraction than Byzantine quorum certification and typically incurs higher latency and communication costs. In asynchronous networks, total-order consensus cannot deterministically guarantee termination in the presence of faults without additional assumptions.

verifiable functions are particularly useful for tasks such as aggregation in federated learning or agreement on how input shares are distributed for confidential MPC.

When needed, Ledgera can further augment Ledgents with the Validator role, enabling client applications to orchestrate computations across multiple executions and to enforce constraints over the application shared state. For example for applications such as cryptocurrency or asset transfers, Validators may keep track of previously validated transfers so as to prevent double spending.

Note that Ledgera can also accommodate applications that require state machine replication, but this remains an option rather than an obligation. In that case, Ledgents can coordinate among themselves using the secure log as a total-ordering service, as is commonly done in many blockchain infrastructures.

What Ledgera challenges, therefore, is not state machine replication itself, but the assumption that every application must be forced into that model. The central novelty of Ledgera lies in its modularity of roles, allowing each application to rely only on the certification and coordination it actually requires. In this way, coordination becomes configurable rather than hard-wiring applications into a single, state-machine-replicated world-state model.

In the following, we present Ledgera's core functionalities, its main usage patterns, and a set of illustrative distributed applications that can benefit from Ledgera's modular approach, including federated learning, multi-party computation, and asset transfer.

***Historical note**. The design of Ledgera was inspired by our experience combining Federated Learning with Blockchain infrastructure [Fantastyc24]. In federated learning, multiple clients collaboratively train an AI model on their local data and periodically aggregate their local models into a global one. Traditionally, this aggregation is offloaded to a trusted server that collects client updates and combines them iteratively.*

To secure aggregation, we explored offloading this computation to a blockchain. However, we quickly realized that the full power of smart contracts was not required and that treating aggregation as a smart contract was performance overkill (notably in latency and consensus overhead). This led us to an architecture in which a committee of nodes, not trusted individually, executes the aggregation and produces a proof that clients (and third parties) can verify. The proofs can be anchored on a blockchain for immutability, while the (potentially large) models are stored in persistent distributed storage.

With Ledgera, we generalise this approach from verifiable aggregation to arbitrary verifiable computations, accommodating a broader range of computational needs and application workflows.

2. Ledgera's core functionalities

Ledgera envisions decentralized ecosystems where independent actors can collaborate with strong guarantees like long-term auditability, accountability and compliance without relying on a central authority. The key idea is to treat cryptographic proofs as a first-class output: computations and data exchanges are made verifiable, by yielding proofs that can be verified and reused across organizations, devices, and time. To this aim, Ledgera provides three main core functionalities to client applications:

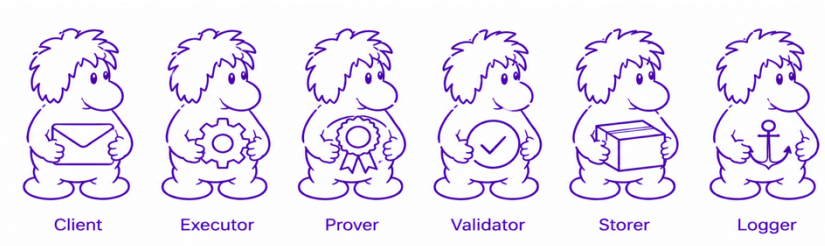
- **Verifiable Compute**. Clients can ask Ledgera to execute Verifiable Functions. In this case, the client application delegates the execution of a specific piece of code to Ledgera. Ledgera then returns a verifiable proof of computation, hereafter simply called a Ledgera Proof of Computation. This proof certifies the correctness, or integrity, of the result.
- **Verifiable Store.** Clients may optionally request that function inputs and/or results be persistently stored in Ledgera, so that they can later be retrieved or made available to other client applications. In that case, Ledgera durably stores inputs/results and produces a Ledgera Proof of Storage in addition to the Ledgera Proof of Computation. The proof contains a storage reference that enables retrieval of the stored value.
- **Anchoring.** For each proof generated in response to a client request, Ledgera anchors proofs as Proof-Anchoring Transactions in a Secure Log, making computation and storage proofs immutable, auditable, and reusable across parties.

On the nature of Ledgera Proofs. Throughout this paper, the term *Ledgera Proof* denotes a quorum-based cryptographic certificate. A Ledgera Proof of Computation certifies that a quorum of Provers attested that the declared Verifiable Function was executed on the agreed input and produced the certified result. Its validity relies both on the unforgeability of the underlying signatures and on the Byzantine fault assumption, namely, that at most f out of n Provers are faulty and that correct Provers sign only results consistent with the declared function and agreed input. This notion should be distinguished from succinct proofs of computation, such as zk-SNARKs or zk-STARKs, where the proof itself establishes the validity of the computation, rather than certifying that a Byzantine quorum attested to it. Ledgera's design is compatible with the future integration of succinct cryptographic proofs as an additional certification mechanism, but the core protocol described here uses quorum-based attestations.

3. Ledgents and Roles

Ledgera relies on software nodes called Ledgents. Each Ledgent has its own cryptographic identity and can play one or more roles depending on the deployment model and application requirements. This identity allows Ledgents to sign requests, protocol messages, and proofs, making their actions accountable and verifiable.

A Ledgent may act as a Client, Executor, Prover, Storer, Logger, or Validator. These roles may be co-located on the same Ledgent or separated across distinct Ledgent pools.

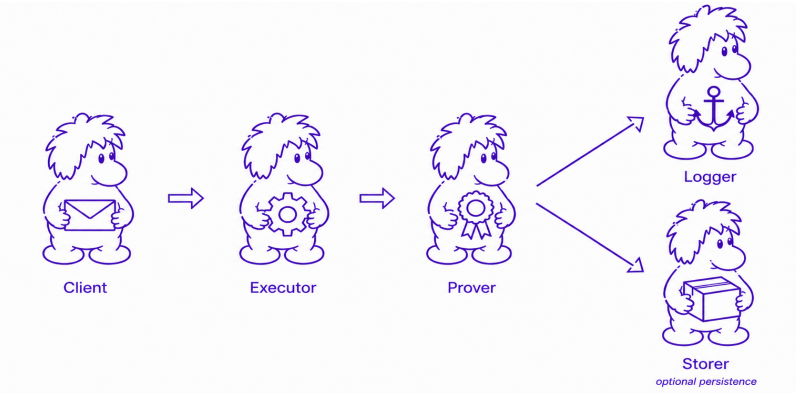


- Clients issue signed requests for the execution and certification of Verifiable Functions. A Client specifies the function to be executed, provides the required inputs, indicates whether persistence is requested, and receives the corresponding result and Ledgera Proofs.
- Executors execute Verifiable Functions.
- Provers certify execution results by producing Ledgera Proofs, typically by collecting a quorum of signatures from Executors on the same result. They then request proof anchoring from Loggers.
- Storers persist inputs and outputs of Verifiable Function executions, when persistence is requested.
- Loggers anchor Ledgera Proofs in a Secure Log, making them immutable, auditable, and reusable across parties.
- Validators verify the validity of Client requests whenever the inputs include Ledgera Proofs generated by previous executions, or whenever the client request aims at updating a shared application state. Validators enable proof-carrying composition across executions and support applications that require shared-state coordination.

4. Client Request Lifecycle

Client Requests

To execute a function, a Client issues an execution request containing the function specification and the corresponding inputs. The request may also include a persistence flag, associated with selected inputs and/or outputs, indicating that they should be persistently stored.



Each request is executed by Executors and triggers the generation of the corresponding Ledgera Proofs by Provers. Once the proofs have been generated, Provers request their anchoring in the Secure Log through Loggers. Depending on the execution semantics chosen by the application, the result may be returned to the Client either before or after anchoring, as follows.

Execution Modes

- Asynchronous execution**:** the result is returned to the Client as soon as the corresponding proofs have been produced by the Provers, while anchoring in the Secure Log may complete later.
- Synchronous execution**:** the result is returned to the Client only after the corresponding proofs have been anchored by Loggers in the Secure Log.

Invocation Modes

Ledgera supports two invocation modes for Verifiable Functions, depending on whether their inputs are provided by a single party or by multiple parties.

- Single-Party Verifiable Function**:** all inputs are provided by a single Client in one execution request. Executors can immediately execute the function and return the result together with a Ledgera Proof of Computation.
- Multi-Party Verifiable Function**:** inputs are contributed by multiple Clients through separate requests. The function is executed only once an explicit input-collection condition is satisfied, such as a predefined participant set, a threshold number of contributions, or a deadline/timeout.

For multi-party Verifiable Functions, Ledgera must first establish which input set will be used for execution. Provers therefore propose candidate « input sets », each justified by the corresponding signed input request by contributing Clients. The Secure Log is used only to select one agreed input set. Once this input set is fixed, Executors run the function and Provers produce the corresponding Ledgera Proof.Thus, total ordering is used only for the limited purpose of resolving input-set agreement, not as the default execution model for the whole application.

Proof-Anchoring Transactions

Proof-anchoring transactions are issued by Provers in response to Client requests. They carry the corresponding Ledgera Proofs and are submitted to Loggers for anchoring in the Secure Log. Depending on the stage of the request lifecycle, different proof-anchoring transactions may be produced. Some of these proofs certify the final outcome of an execution, while others certify intermediate protocol steps, such as the declaration of a function or the agreement on the input set for a multi-party execution. The transaction set includes:

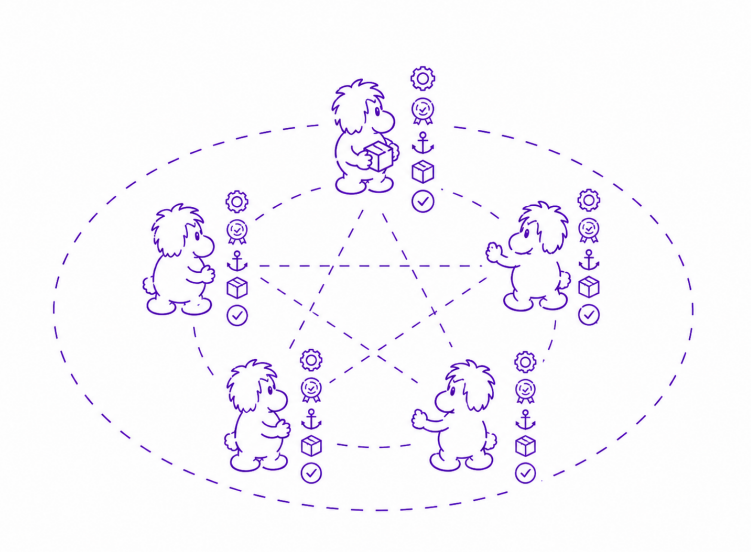
- $\mathbf{T_{fun}}$**:** carries a Proof of Function Declaration, generated when a Client issues a request to execute a given Verifiable Function. It certifies that the function declaration and request metadata have been registered for execution.
- $\mathbf{T_{ins}}$**:** carries a Proof of Valid Input Assignment, used for multi-party requests. It certifies that the input-collection condition has been satisfied and identifies the agreed input set to be used for execution.
- $\mathbf{T_{out}}$**:** carries the Proof of Computation, certifying the result of the Verifiable Function execution.
- $\mathbf{T_{sto}}$**:** carries the Proof of Storage, certifying that the value associated with the computation result has been handed over to the storage layer according to the storage protocol.

5. Deployment Models

Depending on the deployment model, both the number of Ledgents and the roles played by each Ledgent may vary. As illustrative deployment models, we distinguish between a Fully Replicated Model, where each Ledgent plays all roles, and a Role-Specialised Model, where roles are assigned to distinct pools of Ledgents according to their resilience and resource requirements.

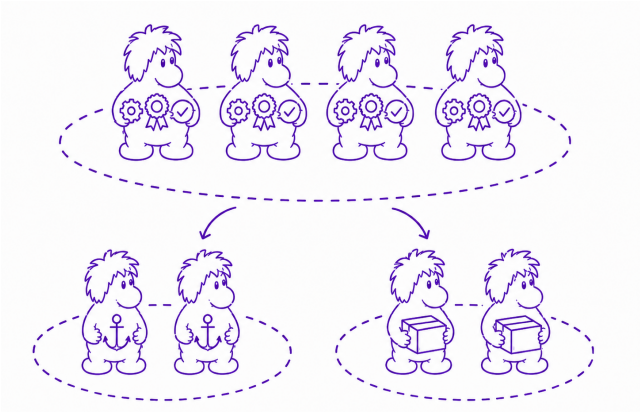
Fully Replicated Model

In a Fully Replicated Model, Ledgents are deployed as a single pool of nodes, where each Ledgent plays all roles. Since every Ledgent plays every role, the system must satisfy the most demanding role-specific resilience requirement. For instance, if Loggers use a BFT Consensus, it implies that the Logger role requires $3f + 1$ Ledgents, where f denotes the maximum number of Byzantine Ledgents to be tolerated. In a Fully Replicated Model, this requirement applies to the whole Ledgent pool, even if other roles would require fewer replicas in isolation.



Role Specialised Models

In Role-Specialised Models, Ledgents are organised into different pools according to the roles they play. This allows each pool to be sized according to the resilience and resource requirements of the roles it supports.



Compute–Persistence Split Model. A concrete instance of this approach is the Compute–Persistence Split Model. In this model, one pool consists of Ledgents that play the roles of Executor, Prover, and Validator, while the Logger and Storer roles are assigned to two additional separate pools of Ledgents. This is the model adopted in Fantastyc [Fantastyc24].

In the following section, we focus on a stateless instance of the Compute–Persistence Split Model, where Validators are not used. This corresponds to Ledgera in its stateless configuration and provides a simple yet representative setting to describe the proof production protocols used by Provers.

6. Proof Production in Ledgera's Stateless Configuration

In the following, we describe how proofs are produced in the Compute–Persistence Split Model, in Ledgera's stateless configuration, i.e., without Validators.

- Ledgera Proof of Computation (LPc)**:** a threshold-signature proof produced by Provers, certifying the correct execution of a Verifiable Function. The proof contains a quorum of signatures from Ledgents acting as Provers on the agreed execution result.
- Ledgera Proof of Storage (LPs)**:** a proof certifying that the value v, identified by H(v) has been correctly shipped toward the Storer pool and is associated with a valid computation proof. The client does not wait for the value to be fully replicated by all Storers before receiving the LPs. This keeps storage replication off the critical path while still guaranteeing that, under the assumed resilience conditions, a valid LPs can eventually be used to retrieve the value.

Byzantine Resilience

In the Compute–Persistence Split Model, under the assumed stateless configuration, Ledgera relies on Byzantine quorum-certification protocols to produce Ledgera Proofs of Computation and Ledgera Proofs of Storage.

For computation, Provers are deployed in a pool of size $n = 2f + 1$, tolerating up to f Byzantine Provers. A quorum of $f + 1$ Prover signatures on the same result is sufficient to produce a Ledgera Proof of Computation. Since at most f Provers may be Byzantine, such a quorum contains at least one correct Prover. Under the assumption that correct Provers sign only the result obtained by executing the declared Verifiable Function on the agreed request input, the proof certifies the correctness of the computation result. This guarantee assumes that Verifiable Functions are deterministic: given the same input, every correct Executor produces the same output. Non-deterministic functions (e.g., those depending on local randomness or wall-clock time) fall outside the scope of this stateless configuration and require additional mechanisms to be handled correctly.

For storage, Storers are deployed in a pool of size $f + 1$, each hosting a replica of the key-value store. Provers ship the value v, addressed by $k = H(v)$, to the Storer pool and produce signed shipment statements. A Ledgera Proof of Storage certifies that the value has been handed over to the storage layer and is associated with a valid computation proof.

Importantly, a Ledgera Proof of Storage does not certify that the value has already been fully replicated by all Storers at the time the proof is returned. Rather, it keeps storage replication off the critical path while ensuring eventual retrievability under the assumed resilience and network-delivery conditions.

The formal specification of these protocols, including their resilience assumptions and correctness guarantees, is provided in the companion Ledgera Yellow Paper.

7. Usage Patterns

Depending on the application and the deployment model, serving a client request may involve a different number of steps and messages among Ledgents. In the following, we describe canonical usage patterns in the Compute–Persistence Split Model.

Verifiable Storage Only

This usage pattern focuses on verifiable storage over a content-addressed replicated KVS.The Verifiable Storage Only pattern, depicted in Figure 1, allows clients to persist data while obtaining verifiable Proof of storage. The resulting proof can later be used to retrieve and verify the stored value.

Closest analogues: content-addressed storage systems (IPFS-like) and replicated KVS backends, but with explicit shipment evidence and guaranteed integrity and availability of the stored value.

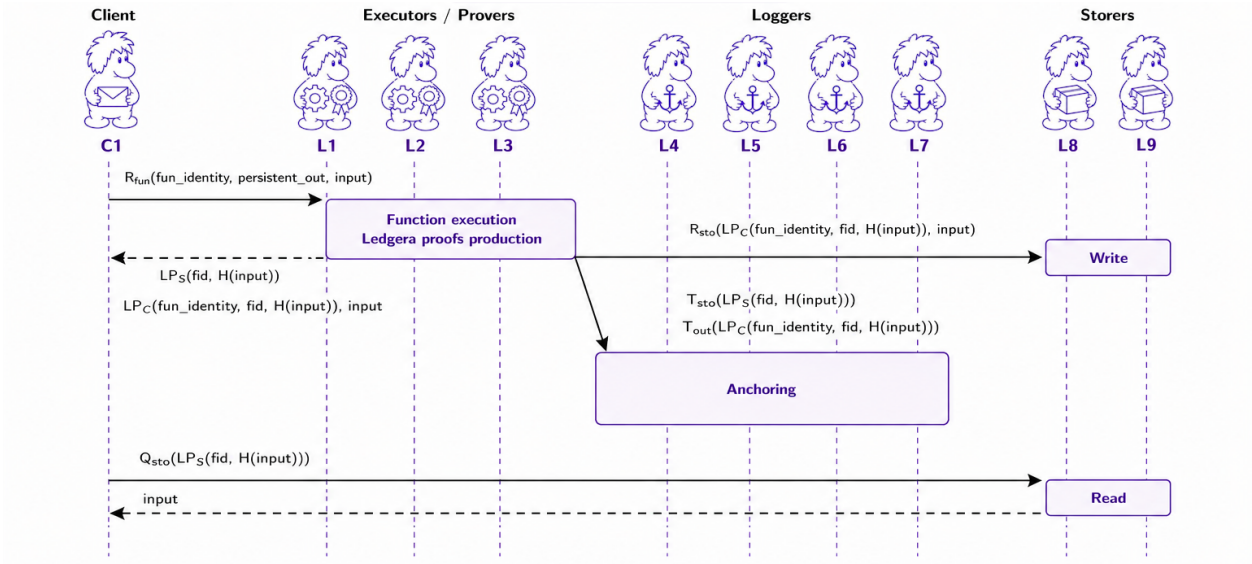


Figure 1 « Verifiable Storage Only » Pattern of Use. A client application requests the execution of an identity verifiable function together with an input value and a persistence policy (R_{fun}). Since the function is the identity, storing the output is equivalent to storing the input value itself. Executors execute the function, while Provers generate a LP_C for the identity computation and a LP_S for the persistent storage request from the client (both proofs contain the unique function identifier fid, associated with this execution). The Executors then relay a storage request (R_{sto}) containing the value and the associated LP_C to the Storers, which store the value durably in the KVS if the associated LP_C is valid. Finally, Provers send the Proof of Storage LP_S to the client application as returning value and to Loggers, along with the LP_C, for anchoring in the Secure Log as respectively T_{sto} and T_{out}. Later, a client can retrieve the value from the KVS by presenting the LP_S to the Storers. If a Storer has not yet received the corresponding value, it waits until the value becomes available rather than rejecting the request. This guarantees eventual retrieval whenever a valid LPS exists.

Verifiable Compute only

This pattern is used to realise stateless deterministic execution producing externally verifiable computation proofs (oracle-like pattern), with no shared function state and no storage persistence.

Figure 2 illustrates the Verifiable Computing Only usage pattern. A client invokes a verifiable function and receives both the computation result and a Proof of Computation attesting its correctness. The generated proof is subsequently anchored in the Secure Log to provide a durable and independently verifiable record of the computation.

Closest analogues: blockchain oracle networks, but generalized from data attestation to computation certification. Ledgera certifies the result of arbitrary Verifiable Functions through threshold proofs, without forcing the computation into a smart contract or a replicated world-state model.

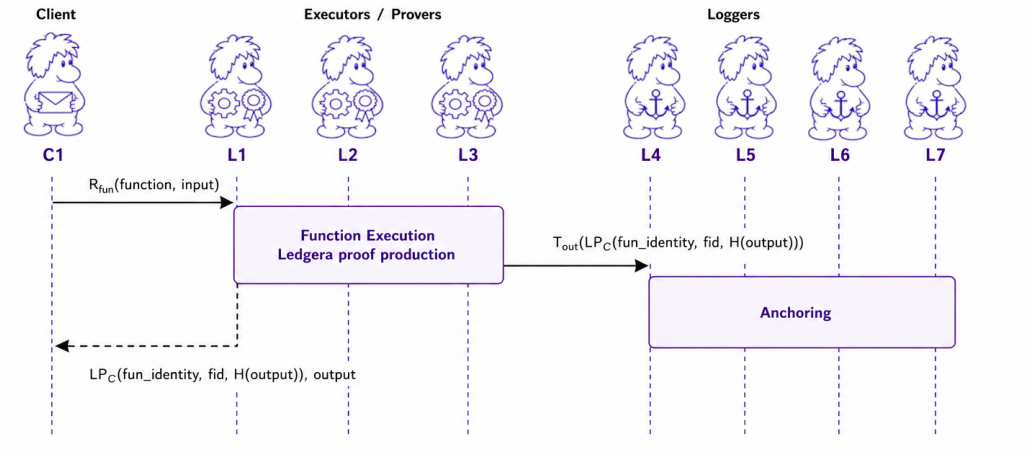


Figure 2 « Verifiable Compute Only » Pattern of Use with single-party non persistent invocation. Messages have the same syntax as detailed for Figure 2.

Simple Data Flow : Compute + Storage

This pattern combines Verifiable Functions with Verifiable Storage to support workflows such as:

- computing a result and storing it under a content-derived key k = H(v),
- storing inputs and outputs in the KVS while anchoring only proofs in the Secure Log,
- building computation pipelines in which stored artifacts become inputs to subsequent verifiable executions.

Figure 3 shows a simple data flow in which the output of one verifiable execution becomes the input of a subsequent one. Rather than transferring the data directly, the second execution references the stored output through its storage certificate, allowing the system to retrieve the required value from storage.

This pattern enables the construction of composable verifiable workflows, where outputs of prior computations can be securely reused as inputs to subsequent executions without requiring direct data transfer between clients.

This approach has been used to develop a fine-tuning application on top of Ledgera [FT26].

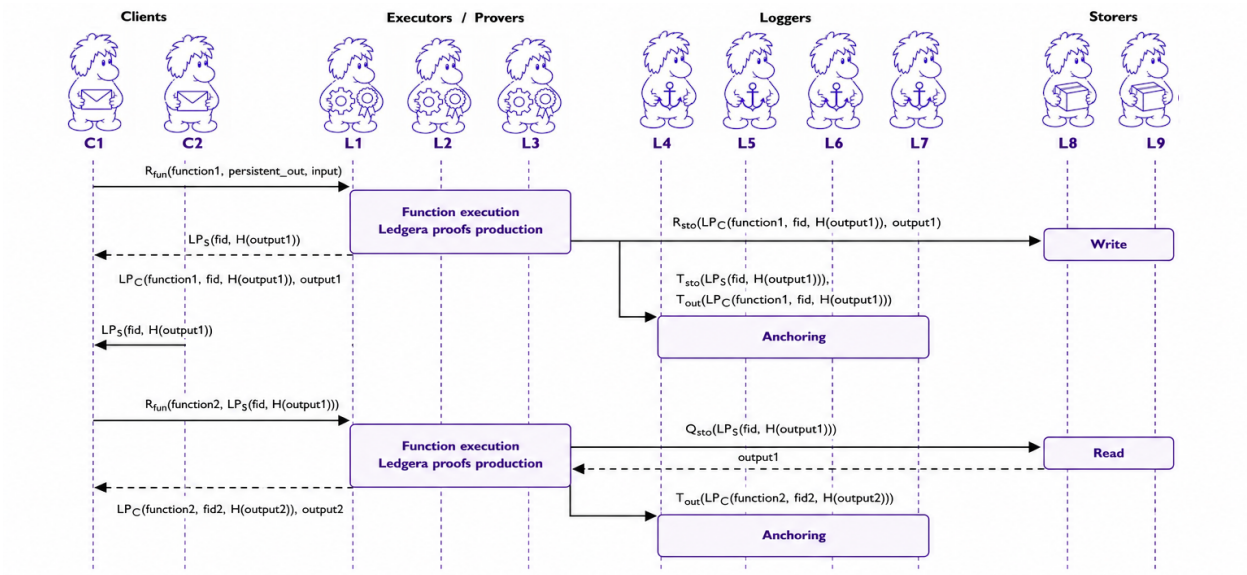


Figure 3 « Simple Data Flow » Pattern. Messages have the same syntax as detailed for Figure 2.

8. Decentralised Applications with Ledgera

The following examples illustrate how Ledgera can support different decentralised applications without forcing all of them into the same state-machine-replication model. Each application uses only the roles and coordination mechanisms it needs: some require only verifiable computation, others require storage, validation across multiple executions, or limited agreement on a set of inputs.

Federated Learning

Federated Learning (FL) allows several clients to train a shared model while keeping their local datasets private. In the classical architecture, a central coordinator distributes the model, collects client updates, aggregates them, and starts the next training round. This coordinator becomes both a single point of failure and a trust bottleneck: clients have limited evidence that the expected participants contributed, that the correct aggregation rule was used, or that the training process followed the declared workflow.

Ledgera can model FL as a sequence of multi-party Verifiable Function executions. A model owner declares an aggregation function, the expected participants, the initial model, and the number of training rounds. At each round, clients train locally and submit their model updates as inputs. Ledgera agrees on the input set for the round, executes the aggregation function, and returns both the new global model and a Ledgera Proof certifying the execution. Ledgera in this case should be configured in stateful mode, with Ledgents playing the role of Validators that keep track of the state. Indeed, the proof produced at one round must be passed as input to the next round. Validators check these proofs and track the current round, ensuring that the workflow progresses according to the declared training plan. At the end, the model owner obtains not only a trained model, but also verifiable evidence that the expected rounds were executed, the declared participants contributed, and the aggregation followed the specified computation.

Confidential Multi-Party Computation

Confidential Multi-Party Computation (MPC) enables several parties to compute a result without revealing their private inputs to one another. Typically, each party splits its secret input into shares and distributes those shares confidentially. Input shares corresponding to different inputs are composed to obtain an output share. The final result of the computation is then obtained by composing the output shares. Only the final output is public while all inputs remain private. In a confidential MPC application built on Ledgera, the main coordination problem is to ensure that the output shares used to compose the output were produced using input shares of the same set of initial inputs.

Ledgera addresses this by modelling the input-submission phase as a multi-party Verifiable Function. The purpose of this first phase is to establish and certify the agreed input set (without revealing it). Once that input set has been fixed, the confidential MPC computation can be executed and Provers can produce a proof of the final result. Ledgera therefore provides a verifiable coordination layer for confidential MPC: it certifies agreement on inputs and makes the execution externally auditable, without requiring parties to reveal their private inputs.

Asset Transfer

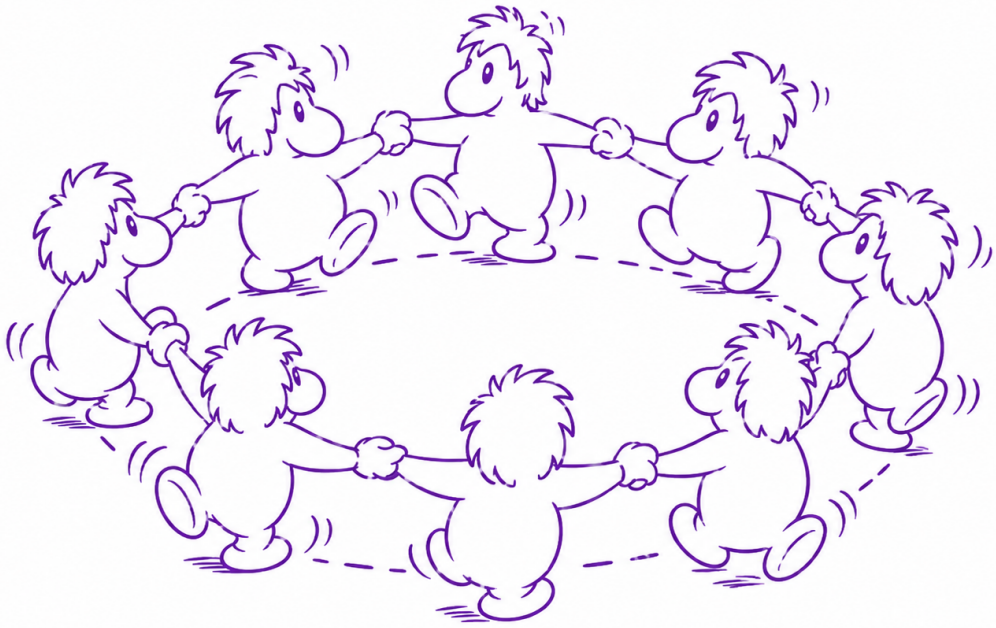
Asset transfer is the canonical ledger application: a client can transfer an asset to another client only if it legitimately owns that asset and has not already spent it. Unlike general smart-contract execution, asset transfer does not require a global total order over all transactions. What matters is that conflicting transfers of the same asset cannot both be validated.

For Asset Transfer, Ledgents must also be Validators that maintain the application state needed to detect conflicts, such as the set of assets or transfer records that have already been validated. Non-conflicting transfers do not need to be ordered in the same way by Validators. The system only needs consistent validation when two requests conflict, for example when the same asset is transferred twice.

For this reason, asset transfer requires stronger validation quorums than a simple stateless Verifiable Function. A deployment for asset transfer should use $3f + 1$ Validators/Executors/Provers, with certification requiring a quorum of $2f + 1$ signatures. The reason is quorum intersection: any two such quorums share at least $f + 1$ nodes, of which at least one is correct. That correct node will refuse to endorse a second, conflicting transfer of the same asset, ensuring that two conflicting transfers cannot both be certified — even though no global total order over all transactions is required.

9. Conclusion

Ledgera is motivated by a simple observation: distributed applications do not all require the same form of coordination. Some need only a certified one-shot computation. Others require durable storage, proof-carrying composition across executions, agreement on a set of inputs, or conflict detection over shared application state. Treating all these cases as instances of state machine replication imposes unnecessary ordering, replication, and latency costs. Ledgera addresses this by separating execution, proof production, storage, anchoring, and validation into distinct, composable roles. Applications can therefore rely only on the guarantees they actually need: computation can be certified without global ordering, results can be persisted when required, and proofs can be anchored for auditability. In this sense, Ledgera reframes the role of a distributed ledger: not as a universal state machine for every application, but as a modular trust substrate for verifiable computation, durable persistence, and auditable coordination.



References:

[AT16] Saurabh Gupta. 2016. A non-consensus based decentralized financial transaction processing model with support for efficient auditing. Master's thesis. Arizona State University

[FS24] Bazzi, Rida, and Sara Tucci-Piergiovanni. "The fractional spending problem: Executing payment transactions in parallel with less than f+ 1 validations." *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing*. 2024.

[FL21] El-Mhamdi, El Mahdi, et al. "Collaborative learning in the jungle (decentralized, byzantine, heterogeneous, asynchronous and nonconvex learning)." *Advances in neural information processing systems* 34 (2021): 25044-25057.

[MPC93] Ben-Or, Michael, Ran Canetti, and Oded Goldreich. "Asynchronous secure computation." *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*. 1993.

[AC23] Frey, Davide, Mathieu Gestin, and Michel Raynal. "The Synchronization Power (Consensus Number) of Access-Control Objects: the Case of AllowList and DenyList." *37th International Symposium on Distributed Computing (DISC 2023)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2023.

[Fantastyc24] Boitier, William, et al. "Fantastyc: Blockchain-based federated learning made secure and practical." *2024 43rd International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2024.

[FT26] Antonella Del Pozzo, Erwan Mahe, Sami Souihi, Sara Tucci-Piergiovanni. "Ledgera and its application to Decentralised Collaborative Fine-Tuning", to IEEE ICDCS 2026 Posters/Demonstration.