Distributed Mining Monitor
Final Report

Final Report for the Distributed Systems Course
Academic Year 2025–2026

Tommaso Remedi
tommaso.remedi@studio.unibo.it

Giacomo Biagioni
giacomo.biagioni@studio.unibo.it

May 10, 2026

1 Introduction

This project presents a *Distributed Mining Monitor*, a controlled experimental platform designed to study Proof-of-Work (PoW) consensus in a distributed setting. The system includes a coordinator service, a configurable set of CPU and GPU miners, and a monitoring dashboard exposing the behaviour of the system in real time. The goal is not to implement a production-ready blockchain, but rather a reproducible laboratory for observing miner competition, fork creation, stale work, branch persistence, and reorganization events.

The final version of the project supports heterogeneous miners, local branch persistence, configurable reorganization thresholds, simulated network delay, and a simplified difficulty-adjustment policy. Instead of re-synchronising to the coordinator before every mining attempt, miners can continue mining on their local branch and switch back to the coordinator only when they fall sufficiently behind. This makes competing branches persist long enough to generate visible reorgs. The coordinator stores the full block DAG, selects the canonical branch according to an explicit policy, and exposes metrics consumed by a dashboard that visualises chain growth, stale branches, accepted blocks by miner, reject rate, difficulty level, and block-DAG evolution.

2 Concept

The developed software is a distributed experimental system composed of three main parts:

- a **coordinator** that validates blocks, stores the global block DAG, selects the canonical chain, and exposes templates and metrics;
- a set of **miners**, implemented in both CPU and GPU variants, competing to solve a Proof-of-Work puzzle;
- a **monitoring dashboard** used to observe the chain, forks, stale blocks, reorgs, and difficulty evolution in real time.

The project is intentionally designed as a **controlled laboratory** rather than as a faithful implementation of a cryptocurrency network. This choice allowed us to focus on distributed-systems concepts such as concurrency, temporary divergence, eventual convergence, observability, and branch selection under competition.
A typical execution proceeds as follows. First, the coordinator is started. Then, a configurable number of CPU and GPU miners are launched. Initially, miners obtain a block template from the coordinator. Subsequently, they continue mining on their *local branch* and only periodically query the coordinator for the current canonical head. If their local branch falls sufficiently behind, they request a new template and realign. This behavior causes forks to persist longer and results in visible reorganization events. During execution, the dashboard periodically queries the coordinator and visualizes the block DAG, the main chain, stale branches, accepted blocks by miner, reject rates, and difficulty-related metrics.
Distribution is needed because the project studies:

- **competition among independent processes**;
- **temporary divergence of local views**;
- **fork creation and persistence**;
- **branch switching and reorganisation**;
- **heterogeneous computational power**, namely CPU versus GPU miners.

The system does not manage user transactions or monetary rewards. It focuses only on the information required to study PoW behaviour: height, previous hash, nonce, miner identity, timestamps, difficulty, and observability metrics.

2.1 Execution Walkthrough

To make the behaviour of the system easier to understand, this subsection summarises a typical execution step by step.

1. The user starts the experiment through the PowerShell launcher script.
2. The script starts the coordinator, the configured miners, and the dashboard.
3. Each miner requests an initial block template from the coordinator. This template tells the miner: "mine the next block at this height, on top of this previous block, with this difficulty". The miner then tries different nonce values locally until it finds a hash that satisfies the required Proof-of-Work difficulty. The candidate block is then submitted to the coordinator for validation.
4. Miners start searching for a valid nonce independently.
5. While mining, each miner periodically asks the coordinator only for the current canonical head.
6. If a miner is still close enough to the canonical chain, it keeps mining on its local branch.
7. If the miner falls behind by at least `SwitchLagBlocks`, it abandons its local branch and requests a new template.
8. When a miner finds a valid nonce, it submits the candidate block to the coordinator.
9. The coordinator validates the block, stores it in the DAG, updates the set of branch tips, and possibly changes the canonical branch.
10. The dashboard continuously queries the coordinator and visualises the current state of the experiment.

This execution model is intentionally simple, but it is sufficient to observe fork creation, stale branches, and reorganisation events.

3 Requirements Elicitation and Analysis

The final requirements are summarized below and organized into functional, non-functional, and implementation requirements.

3.1 Glossary

- **Coordinator**: the central service that validates candidate blocks, stores the DAG, selects the canonical branch, and exposes monitoring data.
- **Miner**: a process searching for a nonce satisfying the PoW difficulty.
- **Canonical branch**: the branch currently selected by the coordinator as main chain.
- **Competing branch**: a valid branch not currently selected as canonical.
- **Stale / orphan block**: a valid block not belonging to the current canonical branch.
- **Fork**: a state in which two or more valid child blocks share the same parent.
- **Reorganisation**: a change in the selected canonical branch caused by a competing branch surpassing the current one according to the branch-selection policy.
- **Switch lag**: the amount of lag tolerated by a miner before abandoning its local branch and requesting a new canonical template.

3.2 Functional Requirements

ID	Requirement	Acceptance criterion
FR1	The system shall provide miners with the current block template, including height, previous hash, and difficulty.	A miner can request a template and start mining with all required parameters.
FR2	The system shall accept block submissions from miners and validate them.	Valid blocks are accepted and stored; invalid submissions are rejected with a reason.
FR3	The coordinator shall keep all accepted blocks in a DAG, including canonical and non-canonical branches.	The full block graph can be reconstructed and exposed through APIs.
FR4	The coordinator shall detect forks and maintain a canonical branch according to an explicit policy.	Competing branches are retained and the main chain can be recomputed after reorgs.
FR5	The system shall support multiple miners executing concurrently.	Several CPU and GPU miners can run at the same time without breaking the coordinator.
FR6	Miners shall be allowed to temporarily continue mining on a local branch.	After an accepted block, a miner can keep extending its local branch until the coordinator's head surpasses it by the configured lag threshold.
FR7	The system shall support configurable reorganisation behaviour.	The coordinator switches canonical branch only when the configured reorg threshold is exceeded.
FR8	The system shall support simulated network delay.	Submission delay can be configured and used to increase branch divergence.
FR9	The system shall expose runtime metrics for monitoring and experimentation.	Height, difficulty, rejects, forks, orphans, and reorgs are visible through the monitoring API.
FR10	The system shall provide a dashboard showing the block DAG and experimental metrics.	A browser-based dashboard updates continuously during experiments.
FR11	The system shall support a simplified difficulty-adjustment mechanism.	Difficulty changes automatically at fixed height intervals and become visible in templates, validation, and dashboard metrics.

3.3 Non-Functional Requirements

ID	Requirement	Acceptance criterion
NFR1	The system shall be reproducible on a single workstation.	All components can run locally on one machine.
NFR2	The system shall be observable.	Relevant metrics and the DAG are visible during the experiment.
NFR3	The system shall remain compact and understandable.	The implementation remains small enough for didactic inspection.
NFR4	The system shall be configurable.	Difficulty, number of miners, polling parameters, reorg threshold, and delay are externally configurable.
NFR5	The system shall tolerate stale work and invalid submissions without stopping the experiment.	Rejects are recorded and do not crash the coordinator.
NFR6	The system shall support heterogeneous mining performance analysis.	CPU and GPU behaviour can be compared under the same execution conditions.

3.4 Implementation Requirements

ID	Requirement	Acceptance criterion
IR1	The project shall be implemented in Python.	Coordinator, miners, and dashboard are implemented in Python.
IR2	The coordinator shall expose HTTP APIs.	Endpoints are available through FastAPI.
IR3	The system shall support both CPU and GPU miners.	Two separate miner implementations are available.
IR4	The system shall support script-based execution and configuration.	A launcher script can start the full environment and pass parameters to all components.
IR5	The project shall be version-controlled and reproducible.	The source code and report are managed in a Git repository.

3.5 Relevant Distributed-System Features

The most relevant distributed-systems features addressed by the project are the following.

Temporary divergence and eventual convergence. The system explicitly allows local views to diverge. Miners can continue mining on a local branch even after the coordinator has selected another canonical branch. Convergence is achieved later, when miners observe that their branch is sufficiently behind and request a new canonical template.

Concurrency and contention. Multiple miners compute PoW in parallel and may submit valid blocks extending the same parent. This directly produces fork creation and branch competition.

Consistency policy. The coordinator is the authoritative component deciding the canonical branch. However, miners are not forced to follow it immediately. This creates a controlled form of eventual consistency in which local progress coexists with delayed global alignment.

Reorganization. Reorganizations are explicit and measurable events. They occur when a competing branch becomes sufficiently ahead of the current canonical branch according to the configured threshold.

Observability. One of the primary goals of the project is observability. The coordinator exposes APIs for templates, metrics, and DAG inspection, while the dashboard provides a visual representation of canonical and non-canonical branches.

Heterogeneity. GPU miners are significantly faster than CPU miners, which makes it possible to study branch dominance, stale work, and the impact of asymmetric computing power.

4 Design

4.1 Architecture

The architecture follows a star-shaped client–server organization. A single coordinator is placed at the centre of the system. Miners and dashboard interact with it through HTTP endpoints.
This architecture was chosen because it:

- simplifies the implementation;
- provides a single authoritative source of state;
- makes experiments reproducible;
- keeps the focus on PoW dynamics rather than on peer discovery and full peer-to-peer propagation.

Although the system is not a decentralised blockchain network, it still preserves the most relevant educational properties: competition among miners, concurrent block proposals, persistence of competing branches, stale work, and reorganisation of the canonical chain. Figure 1 shows the overall architecture of the system. The coordinator acts as the central authoritative component, while CPU and GPU miners compete to generate valid blocks and the dashboard observes the global state through monitoring endpoints.

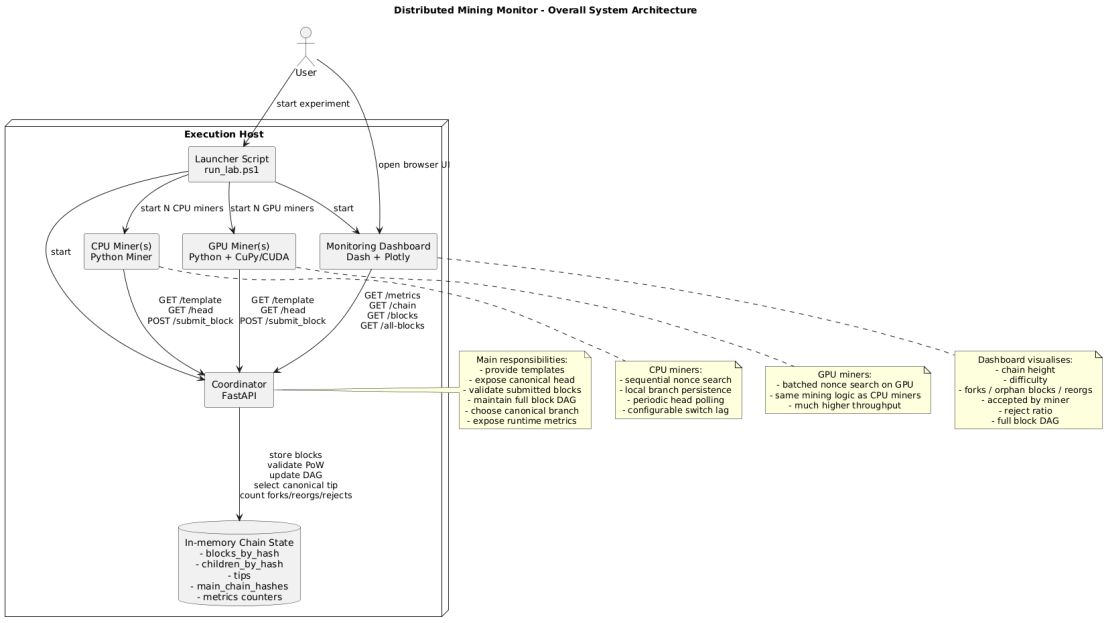

Figure 1: Overall architecture of the Distributed Mining Monitor. The coordinator maintains the block DAG and exposes APIs used by CPU miners, GPU miners, and the monitoring dashboard.

4.2 Infrastructure

The full environment is composed of:

- one **Coordinator** process implemented with FastAPI;
- zero or more **CPU miner** processes;
- zero or more **GPU miner** processes;
- one **Dashboard** process implemented with Dash and Plotly.

All components can run on the same machine. This is acceptable because the goal is to observe distributed behaviour at the process level rather than to build a wide-area deployment.
Configuration is externally controlled through launcher parameters and environment variables. Relevant settings include the number of miners, difficulty base, difficulty-adjustment interval, reorg threshold, switch lag, polling intervals, GPU batch size, and network delay bounds.

4.3 Modelling

The core domain entities are:

- **Block template**: the information miners need to start searching for a valid nonce;
- **Block submission**: a candidate block sent by a miner;
- **Block**: an accepted block stored in the coordinator DAG;
- **Metrics**: runtime counters and derived measurements used for monitoring.

The coordinator state includes:

- all accepted blocks indexed by hash;
- parent–child relations among blocks;
- the set of current tips;
- the set of hashes belonging to the canonical chain;
- reject counters and reject reasons;
- per-miner accepted block counts;
- fork, orphan, and reorganisation counters.

The project also models two explicit policies:

- **Reorganisation threshold**: the amount of advantage a competing branch needs before the coordinator switches canonical branch;

- **Switch lag**: the amount of lag tolerated by a miner before abandoning its local branch and requesting a new template.

Finally, the project introduces a simplified difficulty-adjustment function in which the difficulty of a block is determined only by its height. This avoids ambiguity in the presence of forks and reorgs, because every block at the same height is validated against the same expected difficulty.

4.4 Interaction

The main interaction pattern is request/response over HTTP.
The coordinator exposes the following endpoints:

- `/template`, returning the next block template (height, previous hash, difficulty);
- `/head`, returning the canonical head information used by miners for local branch comparison;
- `/submit_block`, used by miners to submit candidate blocks;
- `/metrics`, returning runtime metrics;
- `/chain`, `/blocks`, and `/all-blocks`, used for DAG inspection and dashboard visualisation.

The interaction logic is the following:

1. a miner requests an initial template from the coordinator;
2. it starts searching for a valid nonce on that local branch;
3. if it finds a valid block and the block is accepted, it continues locally on top of that newly accepted block;
4. while mining, it periodically requests only the canonical head, not a full template;
5. if the canonical chain becomes too far ahead, the miner abandons the local branch and requests a new full template;
6. the dashboard periodically requests metrics and DAG data for visualisation.

This interaction model is one of the most important design choices of the final version. It makes forks persistent enough to generate visible stale branches and measurable reorganisation events.
Figure 2 summarizes the interaction cycle between a miner and the coordinator. After receiving an initial template, the miner keeps mining locally, periodically polls the coordinator head, and only requests a new template when the configured switch lag is exceeded.

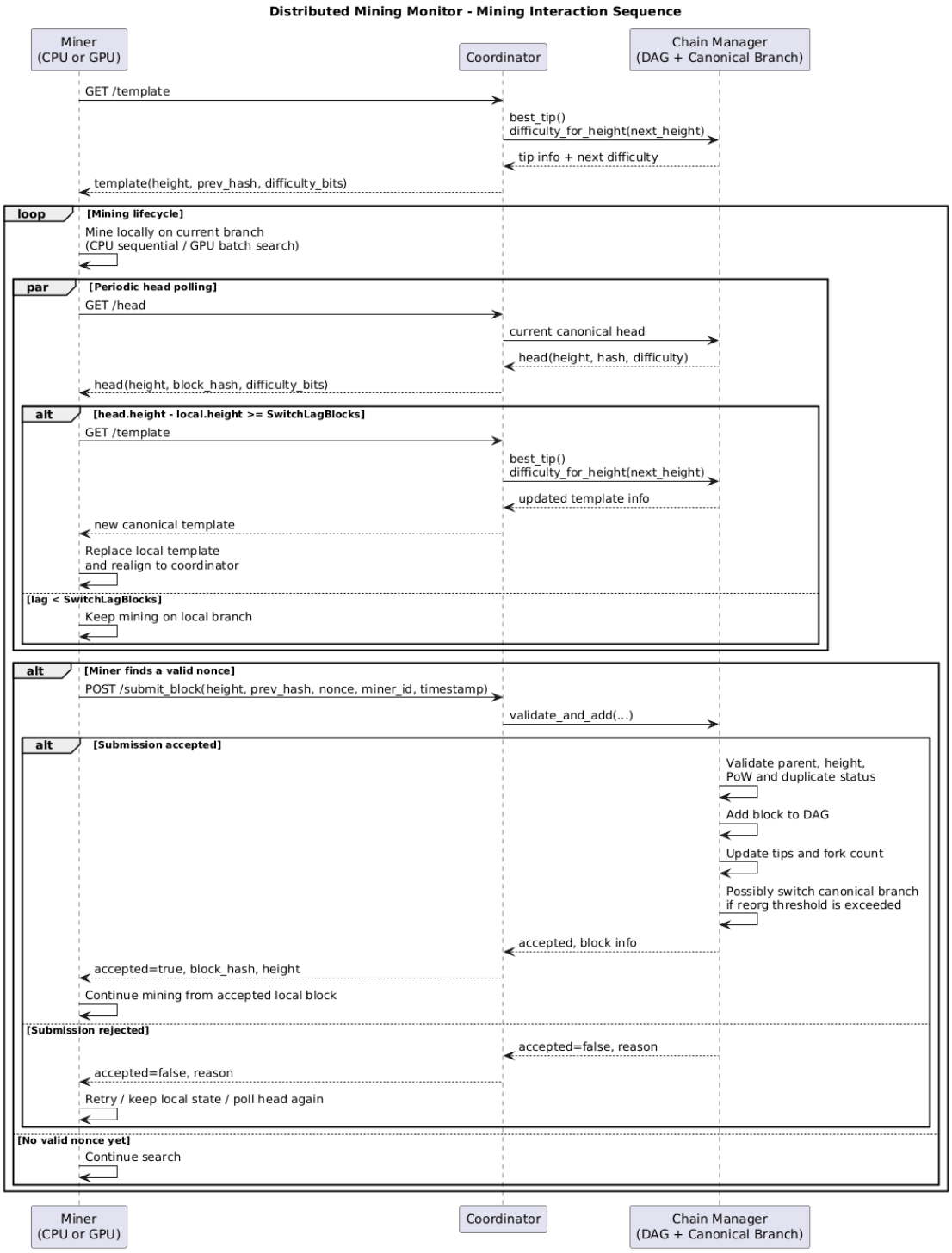

Figure 2: Mining interaction sequence between a miner and the coordinator. Miners obtain an initial template, mine locally, poll the canonical head, and submit blocks when a valid nonce is found.

4.5 Behaviour

Coordinator behaviour. The coordinator maintains the global DAG, validates PoW submissions, counts rejects, updates branch tips, and selects the canonical branch. A reorganisation is counted whenever the coordinator switches from one canonical tip to another that is not a descendant of the previous one.

CPU miner behaviour. A CPU miner performs sequential nonce exploration in software. It starts from an initial template, keeps mining on the local branch, polls the coordinator head at a configurable interval, and requests a new template only when the canonical branch is sufficiently ahead.

GPU miner behaviour. A GPU miner follows the same high-level logic but performs nonce exploration in batches using a CUDA kernel. This makes it possible to compare branch dynamics under heterogeneous computational power.

Dashboard behaviour. The dashboard is a monitoring client. It does not influence consensus logic. It queries metrics and block snapshots and renders the current state of the block DAG, clearly distinguishing canonical and stale branches.

4.6 Branch-selection policy

The final project uses an explicitly controlled branch-selection policy.

Coordinator side. The coordinator follows the current canonical branch when it is extended by a new block. If a new block belongs to a competing branch, the canonical branch is switched only if the competing branch is ahead by at least the configured *reorg threshold*. This introduces a controlled hysteresis that makes branch changes less frequent and easier to observe experimentally.

Miner side. Miners do not immediately discard their local branch when the coordinator advances. Instead, they keep mining locally until the coordinator head is ahead by at least the configured *switch lag*. This behaviour makes it possible for competing branches to survive long enough to create real reorganisation events.
Together, these two mechanisms reproduce the temporary coexistence of competing branches in a simplified but meaningful way.
Figure 3 illustrates the decision logic behind local branch persistence and canonical-branch reorganisation. The miner-side logic determines when a miner abandons its local branch, while the coordinator-side logic determines when a competing branch becomes the new canonical branch.

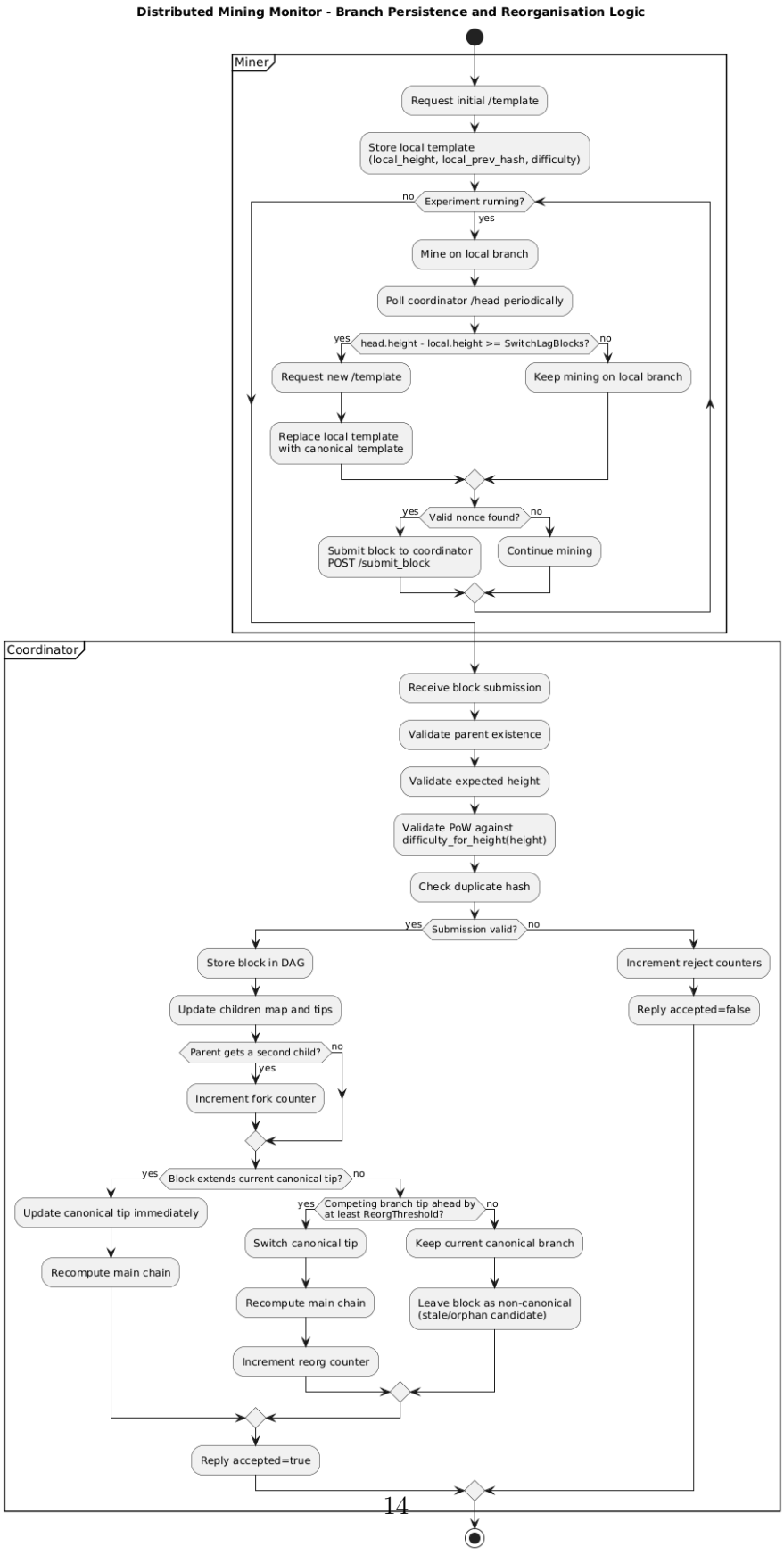

Figure 3: Branch persistence and reorganisation logic. Miners continue mining on local branches until the switch-lag threshold is reached, while the coordinator changes the canonical branch only when the reorganisation threshold is exceeded.

4.7 Difficulty-adjustment policy

The final version introduces a simplified difficulty-adjustment mechanism.
A base difficulty is configured at startup. Then, every fixed number of blocks, the difficulty is increased by one leading-zero bit. Since the probability of finding a valid PoW solution is approximately halved when one bit is added, this means that the expected mining effort is approximately doubled at each adjustment step.
The difficulty of a block is computed as a deterministic function of its height. This is important because it ensures that:

- all blocks at the same height are validated against the same difficulty;
- forks and reorgs do not create ambiguity about which difficulty should be used;
- miners receive the correct difficulty directly from the coordinator template.

4.8 Data and consistency issues

Coordinator state is kept in memory. This is acceptable for a short-lived laboratory system. All state-changing operations are performed centrally, so the coordinator can maintain a consistent canonical view.
At the same time, local miner views are allowed to lag behind, which creates a controlled form of temporary inconsistency. This is the source of persistent forks and reorgs. The project therefore separates:

- **global canonical state** maintained by the coordinator;
- **local branch state** temporarily maintained by each miner.

4.9 Fault tolerance

The system includes basic resilience features:

- dashboard requests fail safely and render placeholders when the coordinator is unreachable;
- miners tolerate temporary request failures and continue with the available local state where possible;
- invalid submissions are rejected and counted rather than causing coordinator failure.

However, the system is not fault tolerant in the strong sense:

- the coordinator remains a single point of failure;
- state is not persisted across restarts;
- no replication or failover mechanism is implemented.

These limitations are consistent with the scope of the project.

4.10 Availability

The coordinator remains continuously available as long as the local execution environment is running. Miners do not require immediate re-synchronisation and therefore can continue to make local progress even while temporarily diverging from the canonical branch.
This behaviour improves experimental realism: local work can proceed even when the global canonical branch is already ahead, and stale work becomes directly observable.

4.11 Security

No authentication or authorization mechanism is implemented. The project operates in a trusted laboratory setting. Cryptographic hashing is used only for the Proof-of-Work puzzle; it is not used for economic security, digital signatures, or adversarial protection.

5 Implementation

5.1 Coordinator implementation

The coordinator is implemented with FastAPI and is responsible for:

- generating templates for the next block;
- validating submissions;
- storing all accepted blocks in a DAG;
- tracking canonical and non-canonical branches;
- exposing metrics and inspection endpoints.

The coordinator stores blocks by hash and keeps an adjacency structure mapping each parent to its children. This is sufficient to reconstruct the full block graph and to count forks, orphan blocks, and reorgs.
The coordinator also implements the simplified difficulty-adjustment policy. The difficulty included in the template for a given height is calculated by a function of the block height. Validation uses the same function, ensuring consistency between mining and acceptance.

5.2 CPU miner implementation

The CPU miner is implemented as a Python process performing sequential hash attempts. It uses the block template returned by the coordinator and repeatedly computes SHA-256 on a fixed 40-byte binary header composed of height, previous hash, and nonce.
An important change compared to the initial version is that the CPU miner no longer requests a full template before every attempt. Instead, it stores the current template locally, continues mining on that branch, and only polls the canonical head to decide whether the branch should be abandoned.
The CPU miner supports the following configurable parameters:

- head polling interval;
- switch lag in blocks;
- minimum and maximum simulated network delay.

5.3 GPU miner implementation

The GPU miner is implemented with CuPy and a custom CUDA raw kernel performing PoW search over a large batch of nonces in parallel. Once a valid nonce is found, the candidate block is submitted to the coordinator.
The GPU miner follows the same local-branch logic as the CPU miner: it maintains a local template, periodically checks the coordinator head, and only requests a new template when sufficiently behind. This makes GPU-generated competing branches survive longer and significantly improves the observability of reorg dynamics.

5.4 Monitoring dashboard

The dashboard is implemented with Dash and Plotly. It periodically queries the coordinator and renders:

- current height;
- current difficulty;
- blocks remaining to the next difficulty increase;
- total rejects and reject ratio;
- coordinator uptime;
- fork count, orphan count, and reorg count;
- accepted canonical versus stale blocks by miner;
- reject rate over time;
- a full DAG view of blocks.

One relevant visual improvement introduced in the final version is the colouring of DAG edges: edges belonging to the canonical chain are shown in green, while edges belonging to stale or orphan branches are shown in red. This makes reorganisation events much easier to interpret visually.

5.5 Launcher script and configuration

The environment is started through a PowerShell launcher script. The script can configure:

- number of CPU and GPU miners;
- base difficulty;
- difficulty-adjustment interval;
- reorganisation threshold;
- miner switch lag;
- head polling intervals;
- GPU batch size;
- minimum and maximum simulated network delay;
- coordinator and dashboard ports.

This makes it easy to run multiple experimental scenarios without changing the source code.

5.6 Technological details

The main technologies used in the project are:

- **Python** for all main components;
- **FastAPI** for the coordinator API;
- **Requests** for miner–coordinator communication;
- **Dash** and **Plotly** for the monitoring dashboard;
- **CuPy / CUDA** for GPU mining;
- **GitHub** for source-code management.

Communication uses HTTP and JSON payloads. This keeps the system transparent and easy to debug.

6 Validation

6.1 Automatic Testing

The project was primarily validated through controlled end-to-end executions rather than through a large dedicated unit-test suite. This choice is consistent with the nature of the system, whose most relevant properties emerge only when multiple components interact concurrently.
Still, some behaviours were checked systematically during development:

- correctness of Proof-of-Work validation for accepted and rejected blocks;
- correctness of block-parent consistency checks;
- correctness of branch switching logic at miner and coordinator level;
- correctness of difficulty computation as a deterministic function of block height.

For the final report, the main validation evidence is therefore provided by acceptance testing and experimental evaluation, since they better capture the distributed behaviour of the system.

6.2 Acceptance test

Acceptance testing was performed by running the complete system under different controlled configurations and observing the resulting behaviour through the dashboard and the coordinator metrics.
The goal of acceptance testing was to verify that the final version of the system satisfies the main intended properties:

- linear chain growth in the absence of competition;
- higher throughput with GPU miners than with CPU miners;
- fork creation under concurrent mining;
- persistence of competing branches due to local miner branch retention;
- actual reorganisation events when a competing branch surpasses the canonical one according to the configured thresholds;
- visible difficulty increase during longer runs.

The following experimental scenarios were selected:

- **T1 – Single CPU miner baseline:** linear execution with one CPU miner;
- **T2 – Single GPU miner baseline:** linear execution with one GPU miner;
- **T3 – Mixed CPU/GPU competition:** concurrent execution with three CPU miners and three GPU miners;

- **T4 – Switch-lag sensitivity:** same mixed setup as T3, varying only the `SwitchLagBlocks` parameter;
- **T5 – Long run with difficulty adjustment:** longer execution aimed at observing the simplified difficulty-increase policy.

6.3 Experimental Evaluation

Each experiment was executed for approximately two minutes, except for the long-run difficulty-adjustment scenario, which was extended in order to allow one or more difficulty changes to become visible.
For each run, we collected the final values shown by the dashboard, including:

- final chain height;
- final difficulty;
- number of forks detected;
- number of orphan blocks;
- number of reorganisation events;
- total rejected submissions;
- reject ratio.

In addition, a screenshot of the final dashboard state was saved for each experiment, in order to visually document the final block-DAG configuration and the evolution of the canonical branch.

6.3.1 Experimental summary

Test	CPU	GPU	Base Diff.	Diff. Int.	Reorg Thr.	Switch Lag	Delay (ms)	Final Height	Final Diff.	Forks	Orphans	Reorgs	Rejects
T1	1	0	21	100	3	2	0–200	10	21	0	0	0	0
T2	0	1	21	100	3	2	0–200	422	25	0	0	0	0
T3	3	3	22	100	3	2	0–600	261	24	42	452	7	0
T4	3	3	22	100	3	1	0–600	271	24	40	453	15	0
T5	0	4	23	100	3	2	0–600	226	25	34	536	12	0

Table 4: Summary of the experimental configurations and final metrics collected from the dashboard.

6.3.2 T1 – Single CPU miner baseline

This experiment was designed to observe the simplest possible execution of the system. With a single CPU miner, the expected behaviour is a linear chain with no meaningful fork creation and no reorganization events. This scenario provides the slowest baseline and is useful for comparing throughput against the GPU case.

Command used

```
.\run_lab.ps1 -CpuMiners 1 -GpuMiners 0 -DifficultyBits 21 `
-DifficultyAdjustmentInterval 100 -ReorgThreshold 3 `
-SwitchLagBlocks 2 -CpuHeadPollMs 200 `
-NetworkDelayMinMs 0 -NetworkDelayMaxMs 200
```

Figure 4: Final dashboard state for T1: single CPU miner baseline.

6.3.3 T2 – Single GPU miner baseline

This experiment reproduces the baseline scenario with a single GPU miner. The chain is still expected to remain linear, but the higher computational throughput of the GPU miner should result in a larger final height than in T1.

Command used

```
.\run_lab.ps1 -CpuMiners 0 -GpuMiners 1 -DifficultyBits 21 `
-DifficultyAdjustmentInterval 100 -ReorgThreshold 3 `
-SwitchLagBlocks 2 -GpuHeadPollMs 200 -GpuBatch 20000 `
-NetworkDelayMinMs 0 -NetworkDelayMaxMs 200
```

Figure 5: Final dashboard state for T2: single GPU miner baseline.

6.3.4 T3 – Mixed CPU/GPU competition

This scenario represents the first genuinely competitive execution. With three CPU miners and three GPU miners, the system is expected to produce fork creation, stale branches, orphan blocks, and possibly reorganization events. This experiment is particularly important because it shows the combined effect of concurrency, heterogeneous computational power, and local branch persistence.

Command used

```
.\run_lab.ps1 -CpuMiners 3 -GpuMiners 3 -DifficultyBits 22 ‘
-DifficultyAdjustmentInterval 100 -ReorgThreshold 3 ‘
-SwitchLagBlocks 2 -CpuHeadPollMs 200 -GpuHeadPollMs 200 ‘
-GpuBatch 20000 -NetworkDelayMinMs 0 -NetworkDelayMaxMs 600
```

Figure 6: Final dashboard state for T3: mixed CPU/GPU competitive scenario.

6.3.5 T4 – Sensitivity to `SwitchLagBlocks`

This experiment reuses the same configuration as T3 and changes only the `SwitchLagBlocks` parameter. The objective is to isolate the impact of local branch persistence on fork duration and reorganization frequency. Lower values should cause miners to realign faster, whereas higher persistence tends to keep competing branches alive for longer.

Command used

```
.\run_lab.ps1 -CpuMiners 3 -GpuMiners 3 -DifficultyBits 22 `
-DifficultyAdjustmentInterval 100 -ReorgThreshold 3 `
-SwitchLagBlocks 1 -CpuHeadPollMs 200 -GpuHeadPollMs 200 `
-GpuBatch 20000 -NetworkDelayMinMs 0 -NetworkDelayMaxMs 600
```

Figure 7: Final dashboard state for T4: mixed CPU/GPU run with modified `SwitchLagBlocks`.

6.3.6 T5 – Long run with difficulty adjustment

The final experiment was executed for longer than the other scenarios in order to observe the effect of the simplified difficulty-adjustment mechanism. In this configuration, the experiment was designed to verify that difficulty changes are propagated correctly through templates, validation, metrics, and dashboard visualization.

Command used

```
.\run_lab.ps1 -CpuMiners 0 -GpuMiners 4 -DifficultyBits 23 `
-DifficultyAdjustmentInterval 100 -ReorgThreshold 3 `
-SwitchLagBlocks 2 -GpuHeadPollMs 200 -GpuBatch 20000 `
-NetworkDelayMinMs 0 -NetworkDelayMaxMs 600
```

Figure 8: Final dashboard state for T5: long run with visible difficulty adjustment.

6.3.7 Discussion

The experimental results confirm that the final version of the system behaves as intended. The two baseline scenarios show that, in the absence of competition, the chain grows linearly and no meaningful reorganization takes place. The mixed CPU/GPU scenario demonstrates that concurrent mining produces fork creation and stale branches in a clear and observable way.

The sensitivity analysis on `SwitchLagBlocks` confirms that local branch persistence plays a crucial role in producing reorganization events. Unlike the initial version of the project, where miners re-synchronized too aggressively with the coordinator, the final implementation allows competing branches to survive long enough for actual canonical-branch changes to occur.

Finally, the long-run experiment confirms that the simplified difficulty-adjustment policy works consistently: difficulty increases become visible in the dashboard and are correctly propagated through the coordinator templates, block validation, and runtime metrics without breaking fork handling or reorganization logic.

7 Deployment

The project can be executed on a developer workstation with Python installed. GPU experiments additionally require a CUDA-compatible environment.
The repository includes a PowerShell launcher script that automatically installs the required dependencies and starts all components of the system. As a consequence, deployment mainly consists of cloning the repository, opening a terminal in the project root, and executing the launcher script with the desired parameters.
GPU-based experiments require a working CUDA environment compatible with the installed CuPy package.

8 User Guide

The project can be executed through the provided PowerShell launcher script. The script automatically installs the required dependencies, starts the coordinator, launches the configured set of CPU and GPU miners, and finally starts the monitoring dashboard. The user only needs to open a terminal in the project root and execute the launcher script with the desired parameters.
A minimal example is the following:

```
.\run_lab.ps1 -CpuMiners 1 -GpuMiners 0 -DifficultyBits 21 ‘
-DifficultyAdjustmentInterval 100 -ReorgThreshold 3 ‘
-SwitchLagBlocks 2 -CpuHeadPollMs 200 ‘
-NetworkDelayMinMs 0 -NetworkDelayMaxMs 200
```

After launch:

- the coordinator becomes available on the configured port;
- the miners start reporting accepted and rejected blocks in their terminals;
- the dashboard becomes available in the browser and shows the real-time state of the experiment.

The user can then repeat the execution with different parameter combinations in order to observe:

- linear chain growth;
- heterogeneous CPU/GPU performance;
- fork creation;
- stale and orphan branches;
- canonical-branch reorganisation;
- simplified difficulty adjustment over time.

9 Self-evaluation

9.1 Tommaso Remedi

Tommaso mainly contributed to the overall architecture, coordinator logic, block-DAG management, and dashboard integration. A major strength of this contribution is the effort devoted to turning the project into an observable distributed-systems laboratory rather than just a mining demo. A limitation is that the architecture remains intentionally centralised and in-memory, which reduces fault tolerance but keeps the project focused and manageable.

9.2 Giacomo Biagioni

Giacomo mainly contributed to miner implementation, GPU support, experimental configuration, and performance-oriented execution aspects. A major strength of this contribution is the introduction of heterogeneous mining behaviour, which significantly improved the experimental value of the project. A limitation is that GPU support adds environmental complexity and reduces portability compared to a CPU-only setup.

Group work split. The project was designed collaboratively. The branch-selection logic, experimental methodology, monitoring strategy, and report structure were discussed jointly. Implementation tasks were split by focusing on complementary areas and then integrated together.

10 Future works

Possible future extensions include:

- implementing a more realistic peer-to-peer propagation model instead of a central coordinator;
- introducing persistence for chain state and experiment logs;
- extending difficulty retargeting from a fixed schedule to a block-time-based policy;
- adding energy-consumption and efficiency metrics for CPU versus GPU miners;
- experimenting with alternative branch-selection rules, such as cumulative work or GHOST-like policies;
- adding WebSocket-based notifications from the coordinator to miners and dashboard clients;
- implementing more advanced network-simulation features such as jitter, packet loss, or asymmetric delays.

References

[1] FastAPI documentation. https://fastapi.tiangolo.com/

[2] Plotly Dash documentation. https://dash.plotly.com/

[3] CuPy documentation. https://cupy.dev/