Polkadot Native Storage

Table of contents

1. Project

1.1 Overview

Our goal is to implement a Filecoin-like Polkadot-native system parachain for data storage.

One that uses DOT as the native token and where anyone in the ecosystem through XCM would be able to store and retrieve data. This is mainly inspired by the vision Gavin presented at Decoded 23 about building the Ubiquitous Supercomputer. When thinking about the app-centricity design and the Space and Cores model it will be critical for us to have native storage and not depend on other networks.

This basically extends the Polkadot value offering to include storage, similar to Filecoin, and brings us closer to the supercomputer vision.

We believe it's very important that this work is done as a public good with DOT used as the fee token, otherwise, any dependence on other networks or other tokens would undermine the goals.

1.2 Details

The work proposed here is based on the extensive research completed and available at https://github.com/eigerco/polkadot-native-storage/blob/main/doc/report/polkadot-native-storage-v1.0.0.pdf. Please read the linked document as it complements this application.

A significant part of the research was the implementation of a proof-of-concept, illustrating several aspects of our planned architecture: Polkadot Native Storage - PoC.
The PoC demonstrates:

- parachain registration to a local relay chain (test environment)
- parachain block production
- XCM: sending messages between parachains using sudo pallet and ping pallet (in a non-test environment, the sudo pallet would be a system vulnerability and a proper governance mechanism should be used.)
- visibility of the Filecoin actors as pallets in the local parachain
- ability to run parts of the FVM code in the Substrate runtime (WASM)
- the ability to interact with the actors (e.g. miner actor - add miner, change worker address) using extrinsics
- RPC endpoint interaction
- ability to control collator selection set

Our work has been introduced to Polkadot in the ecosystem forum:
[Polkadot Native Storage - Ecosystem]

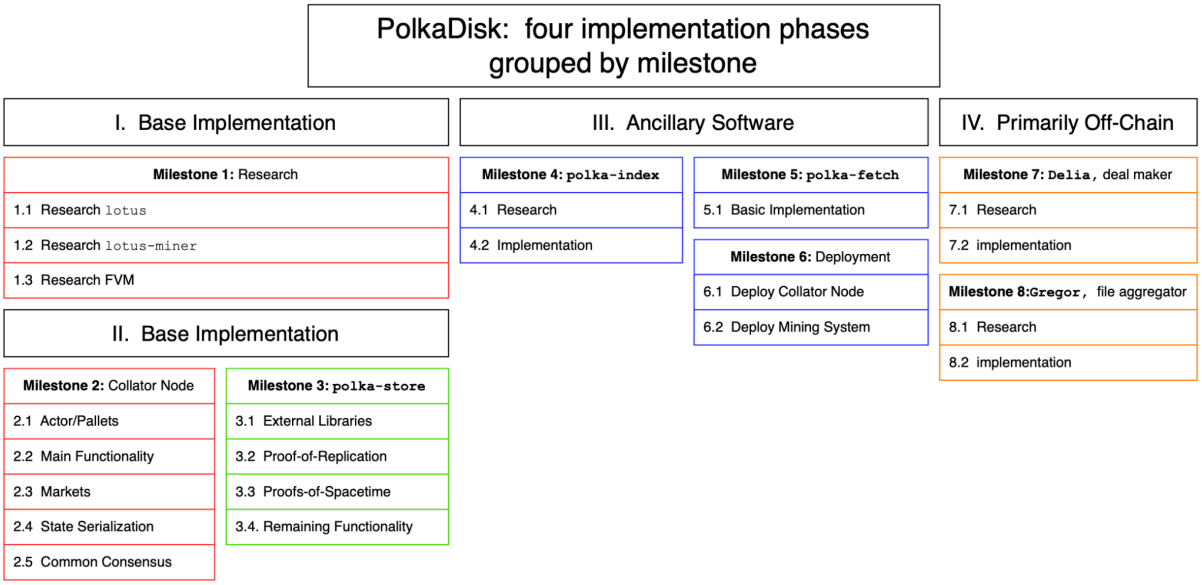
This proposal represents our current thinking, which has continued to evolve from – and improve over – the original solution.
We are moving forward:

- learning from further research,
- engaging in internal discussions,
- debating design choices.

The implementation of Polkadot Native Storage is divided into eight milestones and their constituent tasks.

1. Research
    1.1. Research `lotus`
    1.2. Research `lotus-miner`
    1.3. Research FVM
2. Collator node
    2.1. Port Actors to pallets
    2.2. Main Functionality
    2.3. Markets
    2.4. Serialize blockchain state to disk
    2.5. Common consensus
3. `polka-store`: Rust executable corresponding to a Filecoin storage provider.
    3.1. External Libraries
    3.2. Proof-of-Replication
    3.3. Proofs-of-Spacetime
    3.4. Remaining Functionality
4. `polka-index`
    4.1. Research
    4.2. Implementation
5. `polka-fetch`
    5.1. Basic Functionality
6. Deployment
    6.1. Collator Node
    6.2. Storage System
7. Delia: a web-based demo for creating and executing storage deals.
    7.1. Research
    7.2. Implementation
8. Gregor: a file aggregator, demo web app targeted for smaller file content
    8.1. Research phase
    8.2. Implementation

Here is a diagram of the 21 tasks, organized by milestone and order of implementation.



Milestone and Task Descriptions

1. Research

Tasks:

- Research `lotus`: this is code for the collator
- Research `lotus-miner`: this is code for `polka-store`, the storage system
- Research FVM: specify what will be in our pallets

Research is required due to the scope of the project. This work will decide what parts of the API to keep, what to cut, and what needs to be added. Additionally, some parts of the Filecoin codebase will be examined in greater detail in order to weigh their pros and cons for use in Polkadot Native Storage.

At the completion of the **tasks** in this milestone, all questions regarding the implementation of the collator node and `polka-store` should be answered. The research completed here will also provide a clearer view of the order of implementation and yield optimal implementation strategies.

1.1 Research `lotus` (task 1)

There are three subtasks associated with this task.

1. Analysis of `lotus`

Sufficient time will be spent analyzing both the Filecoin `lotus` application, which runs a full node, along with `lotus-miner`, which performs the storage provider duties. The functionality in these applications will be split between the collator node and `polka-store`.

Care will be taken here to acquire a comprehensive listing of required functionality. The completeness here will enable a smooth implementation of the collator node and `polka-store`.

Our goal is to create a report which covers all the aspects of the Filecoin `lotus` application with ideas and recommendations on how to implement and connect it to Substrate solutions.

2. Define JSON-RPC API

In this subtask, we decide what parts of the API to keep, what to cut, and what needs to be added. The original API is documented here.

There are 21 modules (`Auth`, `Chain`, `Client`, etc). We need to learn what subset of API calls to keep and which calls to eliminate. It will be essential to port only the demanded part to our solution to keep it clean and efficient.

Additionally, if applicable, identify any new calls specific to Polkadot at this time.

An important artifact to come from this research is a list of crates to be implemented, which are ports of Go libraries in lotus (and other Filecoin repositories)

https://github.com/filecoin-project/lotus/blob/master/api/api_test.go

3. Research markets and boost

In Filecoin, the storage and retrieval markets comprise functionality to handle deal negotiation, whether for creating a storage deal or initiating a file retrieval. Each module employs a client and provider finite state machine (FSM), operating via the following protocols over libp2p.

- `/fil/storage/mk/`
- `/fil/storage/ask/`
- `/fil/storage/status/`
- `/fil/retrieval/qry/`

However, Filecoin is phasing these libraries out (now marked as EOL), in favor of boost, a tool for storage providers to manage data onboarding and retrieval. Specifically, there are newer protocols

- `/fil/storage/mk/1.2.0`
- `/fil/storage/status/1.2.0`
- `/fil/storage/transfer/1.0.0`

Filecoin's boost application and libraries should be thoroughly examined as part of this research — other development short cuts may be discovered. Boost is somewhat large and complex, and includes a web front-end with GraphQL. One thing that boost has done for Filecoin is creating a light client node that is not a lotus full node, thus eliminating the need to sync from the chain.

The goal here is to determine:

- a strategy to port the 4 FSMs to Rust (e.g, using the rust-fsm crate, which includes DSL macros)
- how to integrate the protocols with the collator nodes libp2p
- how to port/fork the required data store libraries (e.g., filestore and piecestore); they each bring in several other Go libraries.
- what parts of the node APIs need to be ported.
- discover other useful parts of boost.

1.2 Research `lotus-miner` (task 2)

In the same vein as task **1.1**, but this time for the storage system.
There are three subtasks here:

1. Analysis of `lotus-miner`

Run this and learn the architecture and APIs. There are similar goals for this task as they were for the `lotus` one.

2. Define JSON-RPC API

Corresponding to the work with the collator and `lotus`, here, we decide what parts of the API to keep and what to cut. There is a full listing here, with 15 modules (`Actor`, `Auth`, `Deals`, `Market`, etc). Care will be taken to ensure all market-related calls are accounted for. These will be used by off-chain clients.

In addition, there are two modules in the Worker API which are relevant here, namely `Misc` and `Storage`.
https://filecoin-shipyard.github.io/js-lotus-client/api/worker-api/index.html

3. Analyze the `lotus-worker`

Time should also be spent here considering using the `lotus-worker` executable for doing the CPU & GPU intensive storage tasks (e.g., add piece, seal commit, seal pre-commit, pre-commit, unseal piece, read piece). Strategically, it could save development time by using this application as it is in the `lotus` repository, i.e., do not port that functionality.
This executable does not interact with the blockchain directly.

The result of this subtask would be a decision to fork the lotus worker or not.

Deliverables

- report on `lotus-miner` analysis
- full API specification:
    - modules
    - methods
    - structs
- Report on use or non-use of `lotus-worker`
    - do we gain anything by its use (e.g., development, deployment)
    - enumerate disadvantages

1.3 Research FVM (task 3)

There are two subtasks here

1. Research compiling the FVM for WASM

Our first idea was to bring the actors into Polkadot, bundling each actor into its own pallet. The Rust libraries for IPLD and blockstore are already linked into the individual actor crates.

In our previous grant submission, we sketched an alternative strategy described in our article. Roughly, instead of creating a pallet for each actor, mirroring the structure of the FVM, we would compile the entire FVM into a pallet. This would have mirrored how the Polkadot Frontier project implemented the EVM for Polkadot.

The result of this subtask is a decision regarding which way to proceed.

2. Specification of Actor messages

Here we identify:

- which actors to be ported to pallets
- which messages (extrinsics) to be ported

Calls to the proving system must be identified, as they will be executed by `polka-store`. Multi-threaded code and GPU support do not exist in WebAssembly.

Deliverables

- for each actor/pallet
  - full API specification
  - extrinsics
- design documentation (schemas, diagrams, concepts)

2. Collator node

These are the tasks that make up the collator implementation.

Tasks:

- Port Actors to pallets
- Main Functionality
- Markets
- Serialize blockchain state to disk
- Common consensus

General information about how we plan to port parts of the Filecoin code base are to be found in our research article.

The collator node — when tethered to a storage system (`polka-store`) — runs the Filecoin equivalent of a **Storage Miner Node:**

- writes to the blockchain state
- mines and produces blocks
- participates in the storage market,
- can sync and validate the chain.
- can make storage seals
- seals stored data into sectors. (`polka-store`)
- acquires storage consensus power.

As part of further work, it could be advantageous to enable (perhaps just by feature gating) a chain verifier node, which only would do the following:

- writes to the blockchain state
- can sync and validate the chain.

A verifier node could be used for creating snapshots.

2.1 Port Actors to pallets (task 4)

The specification of this task is the result of research done in task 1.3. That research describes the actors and corresponding messages (extrinsics, for us) to be implemented. Here, we create the pallets, checking their functionality in synthetic

form. Test suites will also be implemented, eventually becoming part of the documentation

The research done in task 1.3 will also drive the decision to compile the FVM as a whole into a pallet, which is an alternative implementation strategy.

Actors are a set of participants that play defined roles within the Filecoin protocol. In principle, they are similar to smart contracts in the Ethereum Virtual Machine: they alone may change the blockchain state.

Each actor is responsible for specific tasks, maintaining its own state tree as one component of the blockchain state. In addition, they interact with other actors to ensure the proper functioning of the overall system.

At this moment, the relevant ones for us to port to pallets are

- system singleton actors: storage market, storage power, reward, cron
- user actors: storage provider, account, payment channel

Deliverables

- pallets codebase
- tests (unit tests, integration tests)
- documentation (with demos)

2.2 Main Functionality (task 5)

The specification of this task is the result of research done in task **1.1. Research lotus**.

Overall, it is a set of coding tasks that cover the expected functionality and tests for the main collator implementation, which includes but is not limited to:

- Pallets and their extrinsics implementation, along with a JSON-RPC API, for use by client applications (CLI/Delia/Gregor, etc.).
- XCMP connectivity implementation, which includes providing the ability to call our chain operations from the different parachains, which is one of the main goals for the project, as the system parachain. This includes:
    - Defining use cases required for implementation based on the business logic agreed upon with stakeholders/community (e.g., requesting remote operations and their types, transferring assets from other chains to pay for requested operations or fees, etc.).
    - Based on the defined use cases and requirements, and utilizing the "staging-xcm" crate in general, as well as the "XCM pallet" with additional documentation available here, XCMP connectivity should be implemented.

In the scope of this task, concrete details should be defined regarding how tokens (DOTs) are managed within a parachain. This involves the integration of Substrate pallets such as "pallet-balances", which is utilized for managing balances of fungible assets (e.g. DOT tokens). The outcome of the research task should clarify whether the use of "pallet-assets" is necessary. The responsibilities of pallet-assets include managing custom fungible assets or tokens. This pallet might be required, for instance, if there is a need to facilitate cross-parachain communication involving custom fungible assets. Additionally, it may be essential if some use case involves custom fungible tokens beyond DOTs, allowing users to create and manage their own assets. The decision to integrate pallet-assets may also depend on whether there is a need for additional features provided by this pallet.

Collaterals

In the scope of this task, logic and implementation details regarding collaterals should be defined.

Keeping with the Filecoin specification regarding miner collaterals, we will ensure network security by requiring participants to invest in resources, namely hardware and upfront token collaterals. Collateral serves as a commitment to appropriate behavior, rewarding value creation and penalizing malicious actions through slashing.

Storage resilience is gained through three mechanisms:

1. **Initial Pledge Collateral:** Miners commit an initial amount of DOT with each storage sector.
2. **Block Reward Collateral:** Minimizes upfront collateral by using block rewards, penalizing miners for failing storage commitments and ensuring incentives for data storage.
3. **Storage Deal Collateral:** Provided by storage providers to secure deals, with detailed information available in the Storage Market Actor.

Based on the above-mentioned specification, stakeholders/community input, and the requirements of the ported version, a concrete mechanism should be defined and implemented.

`forest`

One aid in implementing this relatively large task is a Filecoin node implementation in Rust: https://github.com/ChainSafe/forest. This is an instance of a chain verifier node: no storage is handled and no markets, but block validation and blockchain state transitions are performed.

Storage provider slashing

Proofs-of-Spacetime must be timely submitted to the chain, respecting deadlines within a proving period. The slashing mechanism must be tested here, in conjunction with `polka-store`. It involved several different actors and APIs.

The `cron` pallet processes storage providers every 60 epochs (30 minutes) to validate that miners have correctly proven storage of the portion of their sectors due to be checked. It is this processing that validates the storage, thus ensuring the resiliency of the network. It detects (and penalizes) any lost or corrupted data.

Deliverables

- all API calls implemented
- XCMP connectivity
- tests
- JS/Node.js test client
- Documentation (API definition)

2.3 Markets (task 6)

The deliverable for this task is a Rust library, consisting of the following components and properties.

- finite states machines for storage and retrieval
- API functions support (nodes)
- deal-making protocols executed over `libp2p`
- data persistence.
- form the core of a storage client.

A precise enumeration of API endpoints to be implemented and other implementation details will be the result of research done in subtask 3 of task **1.1.**

1. Finite State Machine

There are 4 state machines:

- storage client
- storage provider
- retrieval client
- retrieval provider

The storage FSM states are of type `StorageDealStatus` and are enumerated here.

The retrieval FSM states are of type `DealStatus` and are enumerated here.

The storage events (both provider and client) are listed here.

Retrieval events are here.

Finally, what is learned about boost will be applied here. In particular, boost comes with a client that can be used to make storage deals – the client does not require a Filecoin node. That should also be **our** goal.

2. Nodes

There are four node types, similar to the FSM types:

- StorageProviderNode
- StorageClientNode
- RetrievalProviderNode
- RetrievalClientNode

Each node is an interface; the API definitions may be found here and here. The research task will determine how much of the API will be ported.

3. Protocols

These are enumerated in task 1.1, subtask 3.

4. Data store

Persistent file storage must be implemented, encompassing the functionality of these libraries:

- `filestore`: a wrapper around Go's `os.File` for use by storage and retrieval markets.
- `piecestore`: a database for storing deal-related PieceInfo and CIDInfo.

Rust libraries for IPFS and IPLD exist, and may prove useful. This task is primarily integrating open source code.

Deliverables

A Rust `polka-markets` library having these features:

- Finite state machines: states + events
- Node interfaces
- Deal protocols
- Data storage

2.4 Serialize blockchain state to disk (task 7)

This task is coupled with task 2.1, as the pallets will mutate the blockchain state. The outcome of this task delivers a code capable of storing data structures either for on-chain workers or off-chain workers (using off-chain storage).

Deliverables

- tests that confirm proper state serialization
    - optional: compare with Forest implementation

2.5 Common consensus (task 8)

This task consists of two subtasks: common consensus mechanisms and collator selection.

The common consensus mechanism assumes creating a code to handle differences between the Substrate and Filecoin consensus. It's deeply connected with the FVM research task as the ability to use FVM inside the Substrate will be critical for agreeing on the Filecoin status. If we choose to handle things inside the FVM, then this task will focus on getting the internal state and mirroring it with Substrate to record all the FVM changes on-chain. In any other way, this task will include creating adapters to translate the state and prepare data to be placed inside the parachain blocks.

The collator selection problem assumes creating a pallet with the collator selection mechanisms based on the storage power. That pallet should promote nodes with more power while still letting some other, smaller nodes take part in the collation process. That's needed to distribute rewards between the block producers and make the whole process beneficial for each party.

Deliverables

- Block generation and collator selection
- Tests
- Documentation (Given that these mechanisms define a fundamental aspect of the entire solution, it is essential to craft comprehensive documentation for both subtasks. This documentation should describe in detail the implementation approach, supplemented by visual representation using UML).

3. `polka-store`

`polka-store` is a Rust executable, our storage provider. It is without blockchain functionality, tethered to a collator node. It is responsible for performing all duties of a storage provider (a.k.a. miner):

- storage min
- maintain file storage, sectors
- storage proving system
- paired with collator node, communicating over JSON-RPC

This is where the API is implemented, semi-porting Go code to Rust. We will be linking here to several external crates and libraries.

Tasks:

- 3.1 External Libraries
- 3.2 Proof-of-Replication
- 3.3 Proofs-of-Spacetime

- 3.4 Remaining Functionality

Additional details may be found in our research article.

3.1 External Libraries (task 9)

This task consists of three subtasks: one major and two relatively minor.

1. Port `dagstore` to Rust

This is one of the libraries that we selected for porting from Go to Rust. It is a major piece of functionality in the Filecoin ecosystem, being ”a sharded store to hold large IPLD graphs efficiently, packaged as location-transparent attachable CAR files”.

The Go code consists of 42 files and 4634 lines of code. This subtask will cover:

- analyze the API
- create the Rust lib
- test suite
- benchmarking

2. Fork `rust-fil-proofs`

The **Filecoin Proving Subsystem** provides the storage proofs required by the Filecoin protocol. It is implemented entirely in Rust as a series of partially inter-dependent crates, some of which export C bindings to the supported API.

We will fork the library for use in `polka-store`.

3. Fork `bellperson`

`bellperson` is a fork of the bellman library. `bellman` is a crate for building zk-SNARK circuits. It provides circuit traits and primitive structures, as well as basic gadget implementations such as booleans and number abstractions.

We will fork the library for use in `polka-store`.

Deliverables

- 3rdparty code directory
- dagstore
    - working library
    - tests
    - documentation (API readme)
- references to FIL changed to DOT
- working tests forked libraries

3.2 Proof-of-Replication (task 10)

In the Filecoin storage lifecycle process, a Proof-of-Replication (PoRep) is generated when a storage provider agrees to store data on behalf of a client.

NB: a storage provider who closes a storage deal with a client for a particular piece of data distributes replicas only within his own storage. A client may avail themselves of multiple storage providers for redundancy and/or security (to be handled efficiently in the data on-ramp clients). There is no connection with the IPFS network, although both IPFS and Filecoin use CIDs and *may* use the same fetch client (`lassie`).

During this process, the storage provider

- receives a piece of client data
- data is placed into a sector
- the sector is sealed by the storage provider
- an encoding is generated, which serves as proof that the SP has replicated a copy of the data they agreed to store. The encoding uses:
    - data which is sealed
    - storage provider
    - the time
- a proof is compressed.
- this result is submitted to the network as certification of storage.

Here it is only required to generate the proof.

This should be callable via a command line API similar to `lotus`.

Deliverables

- API code implemented to support PoRep
- tests

3.3 Proofs-of-Spacetime (task 11)

This task has two subtasks.

1. Generate a Window Proof-of-Storage

This proof is generated at intervals based on `deadlines` and `partitions`, as part of an auditing process.

2. Generate a Winning Proof-of-Storage

This is issued to a miner upon winning a block auction.

In Filecoin, each miner actor is allocated a 24-hr proving period at random upon creation. This proving period is divided into 48 non-overlapping half-hour deadlines

These both should be callable via a command line API similar to `lotus`.

Deliverables

- API code implemented to support Window PoSt
- API code implemented to support Winning PoSt
- tests

3.4 Remaining Functionality (task 12)

Here we implement the remaining API calls: Go code that needs to be either ported or implemented from scratch. The specification of this task is defined in the results of research task **1.2.**

Here are two examples.

1. From the `Sectors` module, method `SectorsStatus`: A sector number is passed in, and a sector info structure is returned.

```
SectorsStatus(ctx context.Context, sid abi.SectorNumber,
showOnChainInfo bool) (SectorInfo, error)
```

2. From the `Market` module, method `MarketGetAsk`: A structure is returned including information about price, minimum and maximum piece sizes.

```
MarketGetAsk(ctx context.Context)
(*storagemarket.SignedStorageAsk, error)
```

Deliverables

- implementation of remaining API calls
- tests
- documentation (API specification)

4. `polka-index`

This is the indexer process. It handles indexes, which are a key-value mapping of CID's to storage providers, including other metadata such as supported protocols.

Tasks:

- Research
- Implementation

`polka-index` has three purposes:

- store indices
- ingest indices
- respond to query requests.

Efficient indexing is of prime importance in order for clients to perform fast retrieval. To improve content discoverability, indexer nodes are instantiated to store mappings of CIDs to content providers for content lookup upon retrieval request.

In IPFS, this is handled through a Kademlia DHT (Distributed Hash Table). However, this method does not scale well and is not terribly performant.

Filecoin has a network indexer instance running at `cid.contact`. When one inserts a CID into the query field, an array of storage providers is returned. If we use the following CID, `bafybeic56z3yccnla3cutmvqsn5zy3g24muupcsjtoyp3pu5pm5amurjx4`, a returned provider will look something like this:

```
Peer Id:   12D3KooWSQ1Qg74oMQ7uAHh8gtME2HdENJMiaoyLnjDQn3drvagg
Multiaddress: /dns/ipfs.va.gg/tcp/3747/http
Protocol:  2336
```

`cid.contact` is an instance of their indexer at https://github.com/ipni/storetheindex.

We recommend implementing `polka-index` starting from the earlier Rust project, https://github.com/vmx/storethehash, which Filecoin ported to Go in the above library.

Deliverables

4.1 Research (task 13)

- a specification of how the application would work
- documentation
- implementation plan
    - software suite: applications & tools
    - resource allocation

4.2 Implementation (task 14)

- working application
- tests
- documentation (API)

5. `polka-fetch`

Only one task here:

- Basic Functionality

The goal of this milestone is to provide file retrieval.

Note regarding file **caching**: This topic was discussed in our previous work here. Caching is a hard problem – Filecoin has several teams working on it. While a comprehensive solution is beyond the scope of this application, it is something we would be keen to address after the completion of this project.

5.1 Basic Functionality (task 15)

There are two subtasks in this task:

1. fork `rs-graphsync`

This library is a Rust implementation of the GraphSync protocol. This protocol is used to transfer content between systems for retrieval.

This is the Go version used by Filecoin.

Another option is to consider a simpler HTTP protocol. We would like to spend some time during this task to research such alternatives.

2. Return a stored file via CID

Implement file retrieval, and get a file back.

Deliverables

- working code able to retrieve a file
- tests (unit and integration)
- documentation (instructions readme)

6. Deployment

Tasks:

- Collator Node
- Storage System

6.1 Collator Node (task 16)

Create necessary scripts and instructions to bootstrap a new collator node. Develop instructions to maintain the node with the steps the user should follow e.g. update the node, test connectivity with other parts of the system.

Deliverables

- scripts
- tests
- instructions and user guides

6.2 Storage System (task 17)

Create necessary scripts and instructions to bootstrap a new mining system. Use Docker to containerize applications to provide a smooth user experience for setting up new storage.

Important is configuration, especially for large systems (e.g., 256GB + GPU, many TBs of storage).

Deliverables

- scripts
- tests
- documentation
- system constraints

7. Delia

Delia is the first of two web apps we are proposing. It is an MVP which implements a service for connecting storage clients with storage providers. Delia hides the low-level details for creating a storage deal and processing the data to be stored. In contrast, Gregor, another web app we propose, is a file **aggregator**.

- Delia is for making **deals**. Large `pieces` are uploaded (a specially created data file) as a whole, and directly stored by a storage provider
- Gregor is a demo on-ramp for storing smaller amounts of data, even a single file.

Tasks:

- 7.1 Research
- 7.2 Implementation
    - Economics
    - Storage client and provider markets
    - Delia implementation

The concept of Delia:

- instantiating a storage client
- finding a miner
- proposing a deal
- executing a deal

A protocol is executed over JSON-RPC between a running instance of Delia and a storage provider.

This web app will be an MVP, a demo of how a working solution could be implemented. A definitive solution is beyond the scope of this proposal, as it involves file depots and cloud resources.

In the research phase, time will be spent studying the various solutions implemented by Filecoin, specifically the boost library: https://github.com/filecoin-project/boost.

7.1 Research (task 18)

In the research phase, time will be spent studying the various solutions implemented by Filecoin, specifically the boost library: https://github.com/filecoin-project/boost.

There will be integration with a storage client: part of the Markets library, capable of interacting with a storage provider to create a deal. Much of that will involve interaction between market entities storage provider and storage client.

Deliverables

- specification of how the application would work
- detailed plan of implementation
    - software suite: applications & tools
    - task breakdown

7.2 Implementation (task 19)

There are two subtasks here.

1. Economics

To maintain an equitable and mutually beneficial relationship between the customer and the storage provider, a financial model should be defined. In this model, the service provider continues to receive benefits for the provided services, and the customer does not incur additional costs despite potential fluctuations in the token's value as the DOT is not a stablecoin and its value may vary based on market conditions.

Since decisions on this task should be based on the involvement of third parties (community, stakeholders), organizational work should be done within the scope of this task. This includes preparing and internally agreeing on financial models, proposing them to stakeholders/community for discussion, and then implementing the decision based on the consensus reached.

As a result, there should be defined a service-providing financial model (or models) outlining the terms for service payments. Here are some potential options to consider:

- **All at once.** Specify whether the price should be determined and billed as a one-time payment for the entire storage time, when the deal is made on the market. Additionally, determine if the size of each prolongation fee should be defined in the future when the storage deal period ends, in a separate market deal based on the new market prices.
- **Subscription.** Alternatively, there could be subscription-based relationships where services are provided by the storage provider until separate payments are provided by the client on an agreed interval basis in the market layer.

- **Combined.** It might be possible that the combination of previously described strategies could be proposed to the client by the storage provider for flexibility purposes.

An important point to consider while defining a service-providing financial model is related to the **"Subscription"** based strategy. This involves directly anchoring the pricing to a stable currency such as USD, calculating the service price based on the cost of DOT equivalent to USD at the moment of fee charging, ensuring consistent compensation value for each transaction and for each deal side.

2. Delia implementation

Build the complete web application per specification.

Research done in task 1.1 will be applied here.

Deliverables

- working versions of specified software applications & tools (MVP)
- tests
- documentation (instruction readme's)

8. Gregor

This is the storage **aggregator** app, a demo web service specifically targeting smaller amounts of data.

Tasks:

- 8.1 Research
- 8.2 Implementation

This server collects files and payments until a given threshold is reached. When that happens, a piece is created. Thereafter, the data flow is similar to Delia.

It is still to be determined exactly how the confirmation of storage per client will be handled.

At a minimum:

- The client has a wallet
- The client receives a confirmation when the file content is in storage
- The client receives a CID upon upload

Just as with Delia, this web app will be an MVP, a demo of one possible working solution.

Deliverables

8.1 Research (task 20)

Deliverables

- specification of how the application would work
- detailed plan of implementation
    - software suite: applications & tools
    - task breakdown

8.2 Implementation (task 21)

Deliverables

- working versions of specified software applications & tools
- tests
- documentation (instruction readme's)

1.3 Ecosystem Fit

Help us locate your project in the Polkadot/Substrate/Kusama landscape and what problems it tries to solve by answering each of these questions:

Where and how does your project fit into the ecosystem?

- Polkadot Native Storage will be a system parachain. We are open to testing the solution on Kusama too.

Who is your target audience (parachain/dapp/wallet/UI developers, designers, your own user base, some dapp's users, yourself)?

- We hope to attract storage providers who aim to offer their storage on Polkadot.
- We hope that all developers in the ecosystem will utilize this storage network through XCM.

What need(s) does your project meet?

- Native file and data storage in the Polkadot ecosystem.

Are there any other projects similar to yours in the Substrate / Polkadot / Kusama ecosystem? If so, how is your project different?

- Crust Network is a storage network built on the Substrate framework. Crust aims to provide a decentralized and incentivized storage protocol that allows users to contribute their storage space and earn rewards in return.

  Here are the main differences compared to Crust:
- Trusted Execution Environments are not used
- We want the fee token to be native DOT throughout (not CRU)

- Crust uses IPFS for actual storage, the parachain bridges to and offloads the storage to IPFS; on the other hand, ours would be native storage provided by Polkadot storage providers.
- While we plan to employ IPFS technology (e.g., CIDs, IPLD, etc), storage providers are completely orthogonal to IPFS per se.

1.4 Future Plans

How you plan to finance the project in the future

- This storage network is intended to be a public good, a system parachain, we do not want to monetize the network. We aim to get funding from the treasury for future maintenance work.

How you intend to use, enhance, promote, and support your project in the short term and the team's long-term plans and intentions in relation to it.

- Eiger is active in the web3 space due to our involvement in multiple ecosystems. We plan to use our connections and socials to promote this work and let developers know that Polkadot may soon have its own storage offering. Our long term plan is to build this to production and maintain it in the future.

What are your plans on auditing and maintaining the work post implementation.

- This current proposal focuses on implementation of the storage system. After completion of the current milestones we plan on doing external audits before deployment to Kusama and Polkadot.
- As the network evolves, we plan to maintain and improve the solution. This will most likely require additional audits to make sure major ongoing protocol-level changes do not break anything.

1.5 Risks

1. Technical constraints: Implementation details affect architecture and design decisions. While we've made a PoC which models several design choices, new technical constraints may emerge. We should be flexible and agile in order to adapt to emergent situations in a timely manner.
2. Time estimates: The project is complex and requires much work and not an insignificant amount of research. We will need to carefully manage time estimates and ensure that we can deliver the project on time.
3. External dependencies: Risks associated with changes to the Polkadot ecosystem and other third-party dependencies, such as potential discontinuation of support or development, breaking changes in various APIs, compatibility challenges, etc. We must proactively plan and organize the necessary steps to identify and quickly resolve such issues.

4. Easy onboarding: We must provide an intuitive process for onboarding storage providers and collators. It may not be easy to achieve and will most likely be an iterative process.
5. Rewards: Storage providers and collators may not be interested in joining the network for non-technical reasons. We should provide a good business model in order to make it attractive.
6. Storage and retrieval complexity: We will need to provide a simple and intuitive interface to make it attractive and easy to use for the majority of users. The complexity must not be too high.
7. Security vulnerabilities: Like any software development, parachain code may potentially have security vulnerabilities that could be exploited by malicious actors. We must outline strategies for responding to and mitigating security incidents promptly. We should discuss plans for routine security audits and assessments to identify and rectify vulnerabilities
8. Integration: We must clearly define and test interfaces between the different system parts. Each part in itself is a complex piece of software: they must all work smoothly together.

2. Team

2.1 Team members

1. Kyle Granger (team leader)
2. Michael Eberhardt
3. Piotr Olszewski
4. Serhii Temchenko
5. Tomek Piotrowski
6. Karlo Mardesic
7. Eloy Peñamaría
8. Pierre Larger

2.2 Contact

- **Contact Name:** Daren Tuzi
- **Contact Email:** daren@eiger.co + hello@eiger.co
- **Website:** Eiger.co

2.3 Legal Structure

- **Registered Address:** Meritullinkatu 1B, 00170 Helsinki, Finland
- **Registered Legal Entity:** Eiger Oy
- **Structure**: Part of the Equilibrium group

2.4 Experience

Eiger and the Equilibrium group as a whole (including Equilibrium Labs) have been working on the decentralized web3 stack since 2018.

- We integrated Move to Substrate. Allowing builders to integrate the Move VM and execute Move smart contracts. Anyone can now create a Move parachain on Polkadot. https://x.com/Polkadot/status/1816109501394637034
- We are building Strawberry, our client implementation of JAM and are very close to the first milestone. More updates on that soon.
- We have been working together with Forte since 2019, in building large scale infrastructure for game to chain integrations.
  (Money Transmitter License and BitLicense from the New York State Department of Financial Services, we can operate with regulatory compliance)
- We implemented the rust implementation of IPFS. We also developed and maintained the main implementation of Internledger in Rust.
- Pathfinder. The most advanced full node implementation of Starknet, the L2 on top of ETH using STARK proofs. We work closely with their core team and ecosystem partners who use it.
- Aleo core, the snarkOS and other developer tools under NDA. The privacy focused smart contract L1 platform going live this year.
- Ziggurat. Network protocol testing framework for ZCash, XRP and Algorand. We have found critical network vulnerabilities and reported to the core teams.
- We work very closely with Polygon Avail. Now known as Avail and released OpEVM.
- Hooks IDE. A browser based IDE to write, compile and execute contracts. Basically a tool to onboard developers and simplify their workflow.
- Starsign. A Starknet multisig implementation written in Cairo
- Eclipse. a prototype system for storing zk-proofs of Solana votes on the Aleo blockchain for the purpose of creating bridges.
- We help Membrane build all their backends and have done the Fireblocks integration. (First EU regulated stablecoin EuroE)
- We helped Elusiv build some of their core privacy tech.
- Zcash UNIFFI. Exposing Rust bindings in other languages.
- We help Celestia build some of their core components in Rust and are building their light node to run in the browser .
- We are building a smart contract library for Stellar Soroban.
- We are working to bridge Solana to Axelar.
- We are building integrations for Zksync.
- We are doing move spec testing for Aptos Move lang, similar to Gambit.
- We put an Ethereum light client on the Internet Computer to facilitate bridging.
- We helped Sovereign integrate Celestia into their modular SDK.

2.4.1 Team Code Repos

https://github.com/eigerco

Please also provide the GitHub accounts of all team members. If they contain no activity, references to projects hosted elsewhere or live are also fine.

1. Kyle Granger https://github.com/kylegranger
2. Piotr Olszewski https://github.com/asmie
3. Serhii Temchenko https://github.com/serg-temchenko
4. Tomek Piotrowski https://github.com/tomekpiotrowski
5. Karlo Mardesic https://github.com/Rqnsom
6. Eloy Peñamaría https://github.com/eloylp
7. Pierre Larger https://github.com/pierre-l

2.4.2 Team LinkedIn Profiles

1. Kyle Granger https://www.linkedin.com/kyle-granger
2. Piotr Olszewski https://www.linkedin.com/piotr-olszewski
3. Serhii Temchenko https://www.linkedin.com/in/temchenko
4. Tomek Piotrowski https://www.linkedin.com/in/tomasz-piotrowski-17466b4/
5. Karlo Mardesic https://www.linkedin.com/in/karlo-mardesic/
6. Eloy Peñamaría https://www.linkedin.com/in/eloylp/
7. Michael Eberhardt https://www.linkedin.com/in/michael-eberhardt-bb15b0100/
8. Pierre Larger https://www.linkedin.com/in/pierre-larger-61729948/

2.5 Communication and reporting

All our work will be done open source so we welcome everyone to follow the progress in the github issues and pull requests.

In addition to this we will push bi-weekly updates of our progress to a Polkadot forum post similar to this one, so the community can easily follow the progress and ask questions in the forum.

3 Milestones/Cost Breakdown

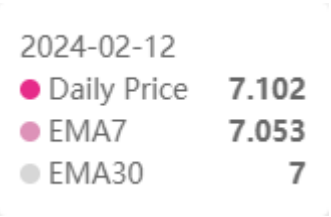
This section should break the development roadmap down into milestones and deliverables.

3.1 Overview

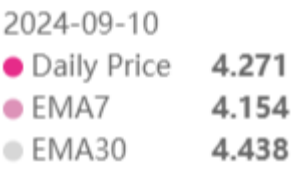
- **Total Estimated Duration:** ~14 months
- **Full-Time Equivalent (FTE):** 112 FTE
- **Total Costs:** 2,800,000 USD
- **Eiger address:** 131MpMXeuKG6L27Ye23uzWr739KFbbrCBdiv39XZtnTCPwQB
- **Contact:** hello@eiger.co
- **DOT pricing:**
  $7 as of Phase 1 referendum 12 Feb 2024, EMA30 as reported by Subscan.
  $4.438 as of Phase 2 referendum 10 Sep 2024, EMA30 as reported by Subscan.
  Phase 3 referendum will be done in stablecoin.

This will be split into 3 phases:

1. Research, documentation and start of collator
   Total amount in USD: $456,250
   Total amount in DOT: 65,178 DOT
   Resources: 18.25 FTE
   Status: Work Completed



2. Collator, polka-store storage subsystem and delia research
   Total amount: $443,750
   Total amount in DOT: 99,988 DOT
   Resources: 17.75 FTE
   Status: Work Completed



3. polka-index + polka-fetch + Delia + Gregor
   Total amount: $1,900,000
   Total amount in USDT: 1,900,000
   Resources: 76 FTE
   Original estimate: 7 months from phase 2 completion
   April 2025 note: 90% already done, improvements continue

Follow our weekly updates here.
Check out the technical book at https://eigerco.github.io/polka-storage-book/

3.2 Milestones

Milestones	Tasks	Deliverables	Time	Rates	Costs	Status
1. Research	1.1 Research `lotus`	- report on `lotus` analysis - full API specification:	4.5 FTE	$25k per FTE FTE here means 1 engineer month	$112.5K	done
	1.2 Research `lotus-miner`	- report on `lotus-miner` analysis - full API specification: - report on use or non-use of lotus-worker	4.5 FTE		$112.5K	done

	1.3 Research FVM	- for each actor/pallet
		  - full API specification
		  - extrinsics
		- documentation
			2.0 FTE		$50.0K	done

2. Collator Node	2.1 Port Actors to pallets	- pallets - tests - documentation	3.5 FTE		$87.5K	done
	2.2 Main Functionality	- all API calls implemented - tests - JS/Node.js test client - documentation	9.0 FTE		$225.0K	done
	2.3 Markets	- `polka-markets` library: FSM, nodes, protocols, data storage	5.5 FTE		$137.5K	done
	2.4 Serialize blockchain state to disk	- tests that confirm proper state serialization	2.0 FTE		$50.0K	done
	2.5 Common consensus	- block generation - tests	3.5 FTE		$87.5K	done
3. polka-store	3.1 External Libraries	- 3rdparty code directory - dagstore	3.5 FTE		$87.5K	done

		- FIL references to changed to DOT - working tests of forked libraries				
	3.2 Proof-of-Replication	- implement PoRep - tests	3.5 FTE		$87.5K	done
	3.3 Proofs-of-Spacetime	- implement Window PoSt - implement Winning PoSt - tests	4.5 FTE		$112.5K	done
	3.4 Remaining Functionality	- implementation of remaining API calls - tests - documentation	16.0 FTE		$400.0K	done
4. polka-index	4.1 Research	- specification - documentation - implementation plan	2.5 FTE		$62.5K	done
	4.2 Implementation	- working application - tests - documentation	9.5 FTE		$237.5K	done
5. polka-fetch	5.1 Basic Functionality	- working code - tests - documentation	4.5 FTE		$112.5K	done
6. Deployment	6.1 Collator Node	- scripts - tests - documentation	2.5 FTE		$62.5K	Private and public testnet first
	6.2 Storage System	- scripts - tests - documentation - system constraints	2.5 FTE		$62.5K	Private and public testnet first
7. Delia	7.1 Research	- specification - implementation plan	6.0 FTE		$150.0K	done but will be improved
	7.2 Implementation	- software & tools - tests - documentation	12.0 FTE		$300.0K	done but will be improved

8. Gregor	8.1 Research	- specification - implementation plan	2.5 FTE		$62.5K	done
	8.2 Implementation	- software & tools - tests - documentation	8.0 FTE		$200.0K	done
		TOTAL	112 FTE		$2,800K	

Milestone task FTE and duration breakdown

#	Tasks	FTE	Num engineers	Months	Quarter mo.
1	1.1 Research `lotus` & JSON-RPC API	4.5	3	1.500	6
2	1.2 Research `lotus-miner` & JSON-RPC API	4.5	3	1.500	6
3	1.3 Research FVM:	2	3	0.667	3
4	2.1 Collator: Port Actors to pallets	3.5	2	1.750	7
5	2.2 Collator: JSON-RPC API	9	4	2.250	9
6	2.3 Markets	5.5	3	1.833	7
7	2.4 Collator: Serialize blockchain state to disk	2	3	0.667	3
8	2.5 Collator: Common consensus (block prod.)	3.5	2	1.750	7
9	3.1 polka-store: External Libraries	3.5	2	1.750	7
10	3.2 polka-store: JSON-RPC I: PoRep	3.5	2	1.750	7
11	3.3 polka-store: JSON-RPC II: PoSt	4.5	3	1.500	6
12	3.4 polka-store: JSON-RPC III: remaining	16	4	4.000	16
13	4.1 `polka-index`: Research	2.5	3	0.833	3
14	4.2 `polka-index`Implementation	9.5	2	4.750	19
15	5.1 `polka-fetch`: Basic Functionality	4.5	3	1.500	6
16	6.1 Collator Node Deployment (parach. integ.)	2.5	3	0.833	3
17	6.2 Mining System Deployment	2.5	3	0.833	3
18	7.1 `Delia`: Research phase (PoC)	6	2	3.000	12
19	7.2 `Delia`: Implementation	12	3	4.000	16
20	8.1 `Gregor`: Research phase	2.5	3	0.833	3
21	8.2 `Gregor`: Implementation	8	2	4.000	16
	total	112.00	112.0	41.50	

Months	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21		
0.00	3	3																				6	
0.25	3	3																				6	
0.50	3	3																				6	
0.75	3	3																				6	
1.00	3	3																				6	
1.25	3	3																				6	
1.50			3						2													5	
1.75			3						2													5	
2.00			3						2													5	
2.25				2					2									2				6	
2.50				2					2									2				6	
2.75				2					2									2				6	
3.00				2					2									2				6	
3.25				2						2								2				6	
3.50				2						2								2				6	
3.75				2						2								2				6	
4.00					4					2								2				8	
4.25					4					2								2				8	
4.50					4					2								2				8	
4.75					4					2								2				8	
5.00					4						3							2				9	
5.25					4						3								3			10	
5.50					4						3								3			10	
5.75					4						3								3			10	
6.00					4						3								3			10	
6.25						3					3								3			9	
6.50						3					3								3			9	
6.75						3						4							3			10	
7.00						3						4							3			10	
7.25						3						4							3			10	
7.50						3						4							3			10	
7.75						3						4							3			10	
8.00							3					4							3			10	
8.25							3					4						-	3			10	
8.50							3					4	3						3			13	
8.75								2				4	3						3			12	
9.00								2				4	3						3			12	
9.25								2				4		2						3		11	
9.50								2				4		2						3		11	
9.75								2				4		2						3		11	
10.00								2				4		2							2	10	
10.25								2				4		2							2	10	
10.50												4		2							2	8	
10.75														2							2	4	
11.00														2		3					2	7	
11.25														2		3					2	7	
11.50														2		3					2	7	
11.75														2			3				2	7	
12.00														2			3				2	7	
12.25														2			3				2	7	
12.50														2	3						2	7	
12.75														2	3						2	7	
13.00														2	3						2	7	
13.25														2	3						2	7	
13.50														2	3						2	7	
13.75														2	3						2	7	
14.00	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-		112

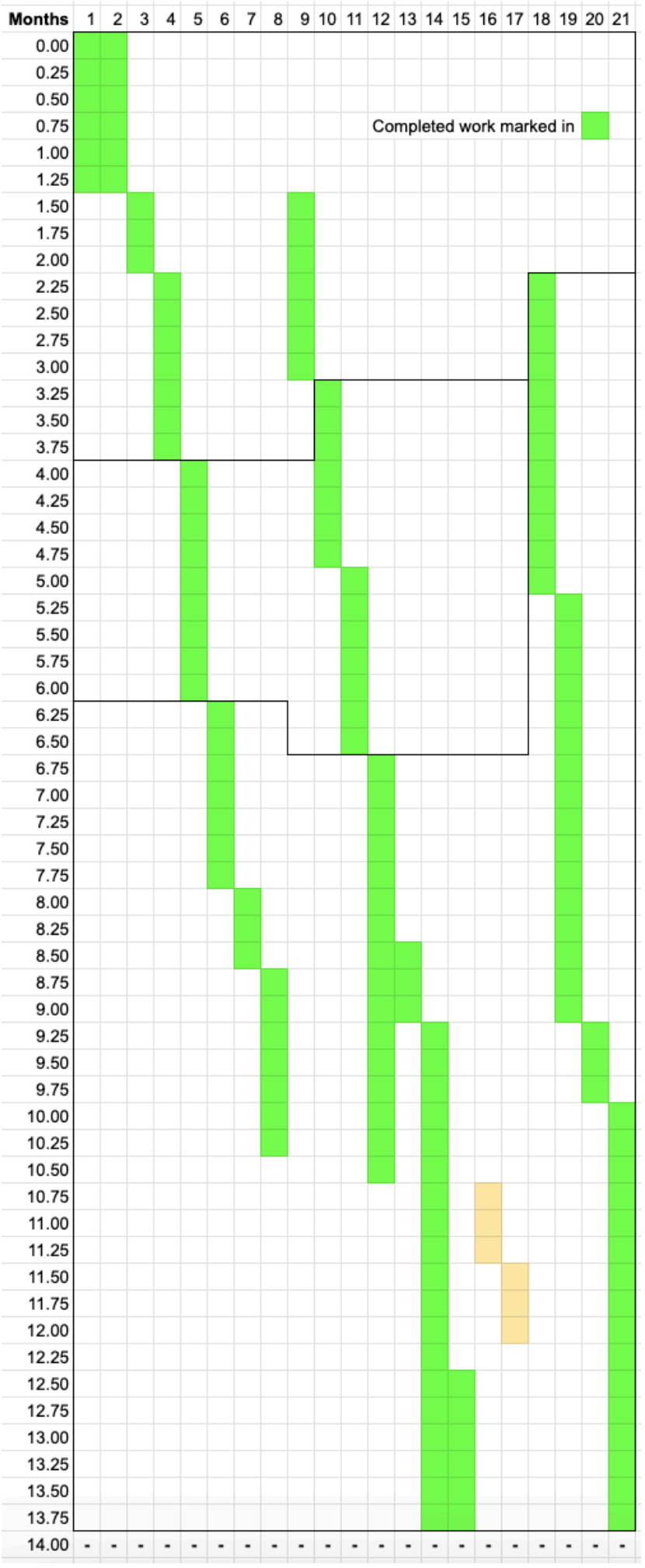
Milestone Key
1. Research: lotus/FVM
1. Research: lotus-miner
2. Collator
3. polka-store
4. polka-index
5. polka-fetch
6. Deployment
7. Delia
8. Gregor

The number inside the cell represents # of engineers working on that task

Months	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21		
0.00	3	3																				6	
0.25	3	3																				6	
0.50	3	3																				6	
0.75	3	3																				6	
1.00	3	3																				6	
1.25	3	3																				6	
1.50			3						2													5	
1.75			3						2													5	
2.00			3						2													5	
2.25				2					2									2				6	
2.50				2					2									2				6	
2.75				2					2									2				6	
3.00				2					2									2				6	
3.25				2						2								2				6	
3.50				2						2								2				6	
3.75				2						2								2				6	
4.00					4					2								2				8	
4.25					4					2								2				8	
4.50					4					2								2				8	
4.75					4					2								2				8	
5.00					4						3							2				9	
5.25					4						3								3			10	
5.50					4						3								3			10	
5.75					4						3								3			10	
6.00					4						3								3			10	
6.25						3					3								3			9	
6.50						3					3								3			9	
6.75						3						4							3			10	
7.00						3						4							3			10	
7.25						3						4							3			10	
7.50						3						4							3			10	
7.75						3						4							3			10	
8.00							3					4							3			10	
8.25							3					4						-	3			10	
8.50							3					4	3						3			13	
8.75								2				4	3						3			12	
9.00								2				4	3						3			12	
9.25								2				4		2						3		11	
9.50								2				4		2						3		11	
9.75								2				4		2						3		11	
10.00								2				4		2							2	10	
10.25								2				4		2							2	10	
10.50												4		2							2	8	
10.75														2		3					2	7	
11.00														2		3					2	7	
11.25														2		3					2	7	
11.50														2			3				2	7	
11.75														2			3				2	7	
12.00														2			3				2	7	
12.25														2							2	4	
12.50														2	3						2	7	
12.75														2	3						2	7	
13.00														2	3						2	7	
13.25														2	3						2	7	
13.50														2	3						2	7	
13.75														2	3						2	7	
14.00	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-		112

Milestone groups
1. Reseach, documentaiton and start of collator
2. Collator + polka-store storage subsystem
3. polka-index + polka-fetch + Delia + Gregor

Tasks
1 Research `lotus` & JSON-RPC API
2 Research `lotus-miner` & JSON-RPC API
3 Research FVM:
4 Collator: Port Actors to pallets
5 Collator: JSON-RPC API
6 Markets
7 Collator: Serialize blockchain state to disk
8 Collator: Common consensus (block prod.)
9 polka-store: External Libraries
10 polka-store: JSON-RPC I: PoRep
11 polka-store: JSON-RPC II: PoSt
12 polka-store: JSON-RPC III: remaining
13 `polka-index`: Research
14 `polka-index`Implementation
15 `polka-fetch`: Basic Functionality
16 Collator Node Deployment (parach. integ.)
17 Mining System Deployment
18 `Delia`: Research phase (PoC)
19 `Delia`: Implementation
20 `Gregor`: Research phase
21 `Gregor`: Implementation


Months
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
0.00 0.25 0.50 0.75 1.00 1.25 1.50 1.75 2.00 2.25 2.50 2.75 3.00 3.25 3.50 3.75 4.00 4.25 4.50 4.75 5.00 5.25 5.50 5.75 6.00 6.25 6.50 6.75 7.00 7.25 7.50 7.75 8.00 8.25 8.50 8.75 9.00 9.25 9.50 9.75 10.00 10.25 10.50 10.75 11.00 11.25 11.50 11.75 12.00 12.25 12.50 12.75 13.00 13.25 13.50 13.75 14.00
Completed work marked in