Leios technical design and implementation plan

Sebastian Nagel sebastian.nagel@iohk.io
Nicolas Frisby nick.frisby@iohk.io
Thomas Vellekoop thomas.vellekoop@iohk.io
Michael Karg michael.karg@iohk.io
Martin Kourim martin.kourim@iohk.io

Contents

1 Introduction

This technical design document bridges the gap between the protocol-level specification (CIP-164) and its concrete implementation in `cardano-node`. While CIP-164 defines *what* the Leios protocol is and *why* it benefits Cardano, this document addresses *how* to implement it reliably and serve as a practical guide for implementation teams.

This document builds on the impact analysis and early threat modelling conducted. The document outlines the necessary architecture changes, highlights key risks and mitigation strategies, and proposes an implementation roadmap. As the implementation plan itself contains exploratory tasks, this document can be considered a living document and reflects our current understanding of the protocol, as well as design decisions taken during implementation.

Besides collecting node-specific details in this document, we intend to contribute implementation-independent specifications to the cardano-blueprint initiative and also update the CIP-164 spec-

ification through pull requests as needed.

Document history

This document is a living artifact and will be updated as implementation progresses, new risks are identified, and validation results become available.

Version	Date	Author	Changes
0.5	2025-10-29	Sebastian Nagel	Re-structure and start design chapter with impact analysis content
0.4	2025-10-27	Sebastian Nagel	Add overview chapter
0.3	2025-10-25	Sebastian Nagel	Add dependencies and interactions
0.2	2025-10-24	Sebastian Nagel	Add implementation plan
0.1	2025-10-15	Sebastian Nagel	Initial draft

2 Overview

Cardano as a cryptocurrency system fundamentally relies on an implementation of Ouroboros, the consensus protocol (TODO cite praos and genesis papers), to realize a permissionless, globally distributed ledger. The consensus protocol provides two essential properties that underpin Cardanos value proposition: **persistence** ensures immutability of confirmed transactions, while **liveness** guarantees that new valid transactions will be included. These properties enable secure and censorship-resistant transfer of value, as well as the execution of smart contracts in a trustless manner.

Ouroboros Leios introduces **high-throughput** as a third fundamental property, extending the currently deployed Ouroboros Praos variant. By enabling the network to process a significantly higher number of transactions per second, Leios addresses the economic scalability requirements necessary to support a growing user base and application ecosystem. This enhancement transforms Cardano from a system optimized for security and decentralization into one that maintains these properties while achieving higher transaction processing capacity demanded by modern blockchain applications.

2.1 From research to implementation

As was the case for the Praos variant of Ouroboros, the specification embodied in the published and peer-reviewed research paper for Ouroboros Leios was not intended to be directly implementable. Initial research and development studies confirmed this expectation, identifying several unsolved problems with the fully concurrent block production design when considering

the concrete Cardano ledger and what consequences this would have (TODO: cite suitable R&D reports, Tech Report #2, Impact analysis survey); further research is needed before those parts can be implemented.

The design presented in CIP-164, also known as Linear Leios, focuses on the core insight of utilizing the unused network bandwidth and computational resources during the necessary and eponymous calm periods of the Praos protocol. This approach provides an immediately implementable design that can deliver orders of magnitude higher throughput while preserving the security guarantees that make Cardano valuable.

The Linear Leios protocol operates by allowing a second, bigger type of block to be produced in the same block production opportunity. Block producers can produce and announce an endorser block (EB), which endorses additional transactions that would not fit within the Praos block. EBs are distributed through the network and subjected to validation by a committee of stake pools, who vote on their transaction data closures availability and validity. Only EBs that achieve a high threshold of stake-weighted votes become certified and can be included in the ledger through exclusive anchoring of a certificate in the subsequent block - now called a ranking block (RB). This mechanism allows for significantly higher transaction throughput while maintaining the security properties of the underlying Praos consensus. See the CIP for more details on the protocol specification and rationale itself.

2.2 Cardano node as a real-time system

The implementation of Leios must be understood in the context of the Cardano node as a concurrent, reactive system operating under real-time constraints in an adversarial environment. While real-time in this context does not refer to the millisecond-level hard deadlines found in industrial control systems, timely action at the scale of seconds nontheless remains crucial to protocol success and network security.

The currently deployed Praos implementation establishes clear data diffusion targets: blocks must reach 95% of nodes within the 5-second parameter, with target performance at 98% and stretch goals at 99%. While these are comfortably achieved most of the time, blocks are regularly adopted within 1 second across the network, there are some situations even in the current system where the target is not reached. For example, due to reward calculations happening at the epoch boundary.

Despite being hard deadlines, these targets reflect the reality that network vulnerability increases when not being met. The protocols safety and liveness guarantees depend on honest nodes being able to propagate blocks rapidly enough to prevent adversarial forks from gaining traction. Failure to meet these timing constraints can lead to increased rates of short forks, reduced chain quality, and - if persistent - ultimately compromise the integrity of the ledger.

2.3 Concurrency and resource management

The current primary responsibilities of a Cardano node are roughly:

- Block diffusion: receiving chains from upstream, validate and select the best chain, and transmit chains downstream.
- Transaction submission: receiving, validating, and transmitting transactions to be included in blocks.
- Block production: creating new blocks and extending current chain when selected as slot leader.

Despite this apparent simplicity, this already results in a highly concurrent system once cardinalities of upstream and downstream network peers are considered. A `cardano-node` with the default configuration maintains 20 upstream hot peers, 10 upstream warm peers and can have up to a few hundred downstream connections, each of which may be simultaneously requesting or serving data. All of these operations share critical resources, including memory, CPU, and network bandwidth, requiring careful resource management to ensure timing requirements are met even under load.

Concretely, in the current system there are (including protocols for supporting features like peer sharing):

- 2 pipelined + 3 non-pipelined instances per upstream peers => 7 threads per upstream peer;
- 1 pipelined + 4 non-pipelined instances per downstream peer => 6 threads per downstream peer

Leios significantly expands this concurrency model by introducing new responsibilities:

- Endorser block and closure diffusion: receiving, validating, and transmitting EBs and their transaction closures.
- Voting and vote diffusion: receiving, validating, and transmitting own and foreign votes on EBs.

Given the proposed Leios mini-protocols, this would result in:

- 4 pipelined + 3 non-pipelined per upstream peers => 11 threads per upstream peer;
- 1 pipelined + 6 non-pipelined per downstream peer => 8 threads per downstream peer

With these two additional functionalities, each across many peers, the node set of concurrent tasks strictly increases. The implementation must ensure that the increased data flows and processing demands do not interfere with each other, or priorization mechanisms ensure to meet the stringent timing constraints necessary for protocol security.

2.4 Designing for the worst-case

Related to the principle of optimizing for the worst case, the security argument for Leios protocol depends critically on worst-case diffusion characteristics. Endorser blocks and their transaction

closures must be small enough that the difference between optimistic diffusion (leading to successful certification) and worst-case diffusion remains bounded by the protocol parameter L_{diff} according to the protocols security argument.

If the optimistic, average-case performance is improved with suitable algorithms, data structures and optimizations, but the worst-case scenario is not, more conservative parameter choices would be required to maintain security guarantees. This would negate the anticipated benefits of the optimizations in the first place. Therefore, the implementation must prioritize ensuring that even in adverse network conditions or under attack, the diffusion of EBs and their closures remains within acceptable bounds.

> [!WARNING] TODO: the situation is not as dire though, we have some design freedom because strictly less work needs to be done on the worst-case path (e.g. rely on certified validity and cheaply build ledger states instead of validating transactions)

Besides, as with Praos, the enhanced information exchange requirements of Leios must not compromise the systems resilience against denial of service attacks and asymmetric resource consumption attempts. The implementation must maintain defensive properties while supporting the increased data flows and processing requirements that enable higher throughput.

2.5 Implementation imperatives

In summary, the technical design described in subsequent chapters must ensure that nodes continue to operate reactively and meet timing requirements despite increased responsibilities and data volumes. This requires careful bounding of resource usage and sophisticated prioritization mechanisms across concurrent responsibilities.

The complexity of this challenge emphasizes the critical importance of non-functional requirements specification for each component, rigorous performance engineering practices, and continuous benchmark validation throughout the development process. Only through systematic attention to these implementation details can the protocol deliver the security and performance properties that make Leios a valuable enhancement to Cardanos capabilities.

The following chapters detail the specific risks that inform architectural decisions, the concrete technical design that addresses these challenges, and the implementation plan that will deliver a production-ready system.

3 Implementation plan

The implementation of Ouroboros Leios represents a substantial evolution of the Cardano consensus protocol, introducing high throughput as a third key property alongside the existing persistence and liveness guarantees. The path from protocol specification to production deployment requires careful validation of assumptions, progressive refinement through multiple system readiness levels, and continuous demonstration of correctness and performance characteristics.

This chapter outlines the strategy for maturing the Leios protocol design through systematic application of formal methods, simulation, prototyping, and testing techniques.

The result is an implementation plan that not only covers the path to active of CIP-164, but also serves as a rationale for what concrete steps will be taken on our product roadmap of realizing Ouroboros Leios.

> [!WARNING]
>
> TODO: mention on-disk storage and its availability; relevant for prototyping and early testnet (chain volume)
>
> TODO: incorporate or at least mention interactions with Peras
>
> TODO: also mention Genesis (potential to only do this later once testnet available?)

3.1 Approach

Research and development of distributed consensus protocols does not follow a linear waterfall process. Rather, the protocol design must be matured through various stages of validation, each building confidence in different aspects of the system. The peer-reviewed research paper provides strong theoretical guarantees under certain assumptions, but translating these guarantees into a working implementation that operates reliably on real-world infrastructure requires bridging substantial gaps. The implementation strategy must therefore balance multiple concerns: validating that core assumptions hold in practice, ensuring that refinements preserve essential properties, building developer confidence through rigorous testing, and ultimately securing acceptance from the governing bodies that must approve deployment to mainnet.

The challenge is compounded by the nature of the system itself. Cardano as deployed on mainnet is a globally distributed system with hard real-time constraints operating in an adversarial environment. Failures or performance degradation cannot be tolerated, as they directly impact the economic value and security guarantees that users depend upon. This necessitates an implementation approach that validates critical properties early, maintains continuous delivery of working prototypes, and ensures transparency in both progress and limitations throughout the development process.

Three principles guide the implementation strategy: First, **early validation** of critical assumptions and risks enables discovery of fundamental problems as early as possible in the development cycle and reduces the likelihood for wasteful pivots and delays in delivery. Second, the implementation must progress through **continuous delivery** of increasingly capable prototypes rather than attempting to build the complete system in isolation. This allows for empirical validation at each stage and enables course corrections based on observed behavior. Third, **transparency** in both capabilities and limitations must be maintained throughout, ensuring that stakeholders including stake pool operators and delegated representatives can make informed decisions about deployment readiness.

These principles are also reflected in the choice of validation techniques applied at each stage. Formal methods provide the strongest guarantees of correctness but apply to abstracted models. Simulation enables exploration of protocol behavior under controlled conditions including adversarial models. Prototypes running on real infrastructure validate that theoretical performance bounds can be achieved in practice. Public testnets demonstrate end-to-end integration and allow the broader community to evaluate the system under realistic conditions.

3.2 Correctness in two dimensions

Formal specification and verification play a central role in ensuring correctness throughout the implementation process, which happens along two dimensions: - Maturity: Implementations maturing from proof of concept, prototype to production-ready release candidates - Diversity: Multiple emerging implementations of Cardano nodes using different programming langugages and targeting slightly different use cases

A protocol specification captured in a formal langugage like Agda, provides an unambiguous description of the protocol that can be checked for consistency and allows proving equivalence and other properties. A formal specification serves as the authoritative reference against which all implementations must be verified.

The approach to formal verification in Leios follows the trail of evidence methodology successfully applied in previous Ouroboros consensus implementations. Rather than attempting to verify the entire codebase directly, which becomes intractable for systems of this complexity, the methodology establishes correctness through a chain of increasingly refined specifications. For example, the high-level specification defines the protocol abstractly, while further refined specifications would focus on details such as message ordering and timing. Finally, an executable implementation is shown to correspond to the formal specification through a combination of techniques including type safety, property-based testing, and trace verification - often summarized as **conformance testing**.

Trace verification deserves particular attention as it provides a bridge between formal specifications and running code. The approach involves instrumenting both the formal specification and the implementation to produce detailed execution traces. These traces can then be compared to verify that the implementation exhibits the same observable behavior as the specification for given inputs. For consensus protocols, the relevant observable behavior includes the sequence of blocks produced, the certificates generated, and the final ledger state. By systematically exploring the space of possible inputs including adversarial scenarios, high confidence can be achieved that the implementation faithfully realizes the specification.

Multiple implementations provide additional assurance through diversity. The primary Haskell implementation in `cardano-node` continues to serve as the reference, while alternative implementations in other languages are currently in development and will eventually increase the **node diversity** of Cardano. Alternative implementations on the node- or component-level serve multiple purposes: - validate that the specification is sufficiently precise and complete, -

exercise different corner cases that might be missed by a single implementation, and - reduce the risk that a subtle bug in one implementation compromises the entire network.

The formal specification must be maintained as a living artifact throughout implementation. As design decisions are made to address practical concerns, these decisions must be reflected back into the specification to ensure it remains accurate. This bidirectional relationship between specification and other steps on the implementation plan is essential. The specification guides implementation, while implementation experience reveals necessary refinements to the specification. Documentation of these refinements and the rationale behind them provides crucial context for future maintainers and for external review. Consequently, the specification itself and other implementation-independent artifacts will be contributed to the `cardano-blueprint` initiative.

3.3 Simulation and protocol validation

Simulations provide a very controlled environment for exploring protocol behavior before deploying to real infrastructure. Two complementary simulation approaches have been used so far to validate the proposed protocol in CIP-164, each with distinct strengths and even using different implementation languages.

A discrete event simulation implemented in Rust, models Leios message exchanges between nodes, abstracting lower-level details for speedrunning orders of magnitude faster than real time to enable statistical analysis over thousands of runs with complete observability and arbitrary adversarial behavior injection. This validates security arguments by systematically exploring protocol behavior under varying loads, expected data diffusion in small to medium sized network topologies, or adversarial scenarios like data withholding, and exploration of protocol parameters before testnet deployment.

Another Haskell-based simulation using IOSim and the actual network framework used in the `cardano-node`. This reduces model-implementation divergence while enabling studies of the dynamic behavior and resource management in detail. While IOSim is used in the existing network and consensus layers through property-based testing, and extends naturally to Leios components, the simulator built from this was not able to scale to large networks.

Both approaches necessarily abstract real system details and thus provide evidence of correct behavior under idealized conditions and suggest workable parameters, but cannot definitively predict real-world performance. Maintaining simulation synchronization with evolving implementation requires discipline, but enables rapid exploration of alternatives, early feature validation, and serves as executable documentation for new developers.

3.4 Prototyping and adversarial testing

Prototypes on real infrastructure validate performance characteristics that simulation typically cannot guarantee. The line between simulation and prototyping is blurry, but both concepts share the trait of allowing rapid exploration of the most uncertain aspects of the design before

committing to a full implementation. Referring back to the key threats and assumptions to validate early, the primary focus of prototyping is on network diffusion performance under high throughput conditions and adversarial scenarios.

Network diffusion prototype: An early implementation of the actual Leios network protocols and freshest-first delivery mechanisms, that allows running experiments with various network topologies. Ledger validation of Leios concepts is stubbed out and transmitted data is generated synthetically to focus purely on network performance. Deployed to controlled environments like local devnets and private testnets like the the Performance and Testing cluster, this prototype systematically explores how performance scales with network size and block size, tests different topologies, and crucially answers whether the real network stack achieves the diffusion deadlines required by protocol security arguments. Key measurements include endorser block arrival time distributions, freshest-first multiplexing effectiveness, topology impact on diffusion, and behavior under adversarial scenarios including eclipse attempts and targeted withholding. These measurements will answer questions like, how much freshest-first delivery we need, whether the proposed network protocols are practical to implement and what protocol parameter are feasible.

Adversarial testing represents a crucial aspect of prototype validation. In a controlled environment, some nodes can still be configured to exhibit adversarial behaviors such as sending invalid blocks, withholding information, or attempting to exhaust resources of honest nodes. Observing how honest nodes respond provides evidence that the mitigations described in the design are effective. Despite using real network communication, such systems can still be determinstically simulation-tested using tools like Antithesis, which is currently picked up also by node-level tests in the Cardano community via moog. If we can put this technique to use for adversarial testing of Leios prototypes and release candidates, this can greatly enhance our ability to validate the protocol under challenging conditions by exploring a much wider range of adversarial scenarios than would be feasible through manually created rigit test scenarios.

Beyond networking prototypes, additional focused prototypes may be created to address other known unknowns of the implementation:

Ledger validation benchmark: measures the throughput of transaction validation and ledger state updates. This is critical for understanding whether a node can process the contents of large endorser blocks within the available time budget and confirm whether our models for transaction validation are correct. The benchmark explores different transaction types and sizes, measures the impact of caching strategies, and validates the performance improvement from the no-validation application of certified transactions.

Cryptographic primitives prototype: validates the performance and correctness of the BLS signature scheme integration. This includes key generation, signing, verification, and aggregation operations. The prototype must demonstrate that batch verification of large numbers of votes can complete within the voting period deadline. It also serves to validate the proof-of-possession mechanism and explore key rotation techniques.

Focused prototypes provide empirical data that complements the theoretical analysis. They reveal where optimizations are necessary and validate that the required performance is achievable with available hardware. They also serve to build developer confidence in the feasibility of the overall design, as well as directly validate and inform architectural decisions. Discovering a fundamental performance limitation early, through prototyping, is far preferable to discovering it late during testnet deployment or, worse, after mainnet deployment.

3.5 Public testnets and integration

A public testnet serves distinct purposes over simulations and controlled environments: it requires integration of all components into a complete implementation, enables for tests under realistic conditions with diverse node operators and hardware, and allows the community to experience enhanced throughput directly. While some shortcuts can still be made, the testnet-ready implementation must offer complete Leios functionality - endorser block production and diffusion, vote aggregation, certificate formation, ledger integration, enhanced mempool - plus sufficient robustness for continuous operation and operational tooling for deployment and monitoring.

The testnet enables multiple validation categories. Functional testing verifies correct protocol operation: nodes produce endorser blocks when elected, votes aggregate into certificates, certified blocks incorporate into the ledger, and ledger state remains consistent. Performance testing measures achieved throughput against business requirements - sustained transaction rate, mempool-to-ledger latency, and behavior under bursty synthetic workloads. Adversarial testing is limited on a public testnet, but some attempts with deliberately misbehaving nodes can be made on withholding blocks, sending invalid data, attempting network partitioning, or resource exhaustion.

The testnet integrates ecosystem tooling: wallets handling increased throughput, block explorers understanding new structures, monitoring systems tracking Leios metrics, and stake pool operator documentation and deployment guides. Crucially, the testnet further enables empirical parameter selection (size limits, timing parameters), where simulation provides initial guidance but real-world testing with community feedback informs acceptable mainnet values.

Software deployed to the public testnet progressively converges toward mainnet release candidates. Early deployments may use instrumented prototypes lacking production optimizations; later upgrades run increasingly complete and optimized implementations. Eventually, all changes as outlined in this design document must be realized in the `cardano-node` and other node implementations. This progressive refinement maintains community engagement while preserving engineering velocity. Traces from testnet nodes can still be verified against formal specifications using the trace verification approach, ultimately linking the abstraction layers.

3.6 Mainnet deployment readiness

Mainnet deployment requires governance approval and operational readiness beyond technical validation. The Cardano governance process involves delegated representatives and stake pool operators who need clear understanding of proposed changes, benefits, and risks. Technical validation evidence from formal methods, simulation, prototyping, and testnet operation must be communicated accessibly beyond technical documentation.

Operational readiness encompasses stake pool operator testing in their environments, updated procedures and training, clearly documented upgrade procedures, updated monitoring and alerting systems, and prepared support channels. The hard fork combinator enables relatively smooth transitions, but Leios represents substantial consensus changes. Conservative timeline estimates must account for discovering and addressing unexpected issues - a normal part of the hard-fork scheduling process. The months of validation and refinement required before prudent mainnet deployment reflect the critical nature of modifications to a system holding substantial economic value and providing essential services that users depend upon.

> [!WARNING]
>
> TODO: more thoughts:
>
> - why (deltaq) modeling? quick results and continued utility in parameterization
> - parameterization in general as a (communication) tool; see also Peras parameterization dashboard https://github.com/tweag/cardano-peras/issues/54
> - whats left for the hard-fork after all this? more-and-more testing / maturing, governance-related topics (new protocol parameters, hard-fork coordination)

4 Dependencies and interactions

The changes necessary to realizing Leios must integrate carefully with existing infrastructure and emerging features. This section examines the critical dependencies that must be satisfied before Leios deployment, identifies synergies with parallel developments, and analyzes potential conflicts that require careful coordination. The analysis informs both the implementation timeline and architectural decisions throughout the development process.

4.1 On-disk storage of ledger state

> [!WARNING]
>
> TODO: Add some links and references to UTxO-HD and Ledger-HD specification and status

The transition from memory-based to disk-based ledger state storage represents a fundamental prerequisite for Leios deployment. This dependency stems directly from the throughput characteristics that Leios is designed to enable.

At the time of writing, the latest released `cardano-node` implementation supports UTxO state storage on disk through UTxO-HD, while other parts of the ledger state including reward accounts are put on disk within the Ledger-HD initiative. The completion of this transition is essential for Leios viability, as the increased transaction volume would otherwise quickly exhaust available memory resources on realistic hardware configurations.

The shift to disk-based storage fundamentally alters the resource profile of node operation. Memory requirements become more predictable and bounded, while disk I/O bandwidth and storage capacity emerge as primary constraints. Most significantly, ledger state access latency necessarily increases relative to memory-based operations, and this latency must be accounted for in the timing constraints that govern transaction validation.

Early validation through comprehensive benchmarking becomes crucial to identify required optimizations in ledger state access patterns. The (re-)validation of orders of magnitude bigger transaction closures, potentially initiated by multiple concurrent threads, places particular stress on the storage subsystem, as multiple validation threads may contend for access to the same underlying state. The timing requirements for vote production - where nodes must complete endorser block validation within the L_{vote} period - on one hand, and applying (without validation) thousands transactions during block diffusion on the other hand, impose hard constraints on acceptable access latencies.

These performance characteristics must be validated empirically rather than estimated theoretically. The ledger prototyping described in the implementation plan must therefore include realistic disk-based storage configurations that mirror the expected deployment environment.

4.2 Synergies with Peras

The relationship between Ouroboros Leios and Ouroboros Peras presents both opportunities for synergy and challenges requiring careful coordination. As characterized in the Peras design document, the two protocols are orthogonal in their fundamental mechanisms, Leios addressing throughput while Peras improves finality, but their concurrent development and potential deployment creates several interaction points.

Resource contention and prioritization emerges as the most immediate coordination challenge. Both protocols introduce additional network traffic that competes with existing Praos communication. The resource management requirements for Leios - prioritizing Praos traffic above fresh Leios traffic above stale Leios traffic - must be extended to also accommodate Peras network messages. Any prioritization scheme requires careful analysis of the timing constraints for each protocol to ensure that neither compromises the others security guarantees. Current understanding is that Peras traffic should be prioritized above both, stale and fresh Leios traffic, such that Leios protocol burst attacks may not force Peras into a cooldown period.

Vote diffusion protocols present a potential area for code reuse, though this opportunity comes with important caveats. The Leios implementation will initially evaluate the vote diffusion protocols specified in CIP-164 for their resilience against protocol burst attacks and general performance characteristics. Once the Peras object diffusion mini-protocol becomes available, it should also be evaluated for applicability to Leios vote diffusion. However, the distinct performance requirements and timing constraints of the two protocols may ultimately demand separate implementations despite structural similarities.

Cryptographic infrastructure offers the most promising near-term synergy. Both protocols are based on signature schemes using BLS12-381 keys, creating an opportunity for shared cryptographic infrastructure. If key material can be shared across protocols, stake pool operators would need to generate and register only one additional key pair rather than separate keys for each protocol. This shared approach would significantly simplify the bootstrapping process for whichever protocol deploys second.

The Peras requirement for forward secrecy may necessitate the use of Pixel signatures on top of the BLS12-381 curve, in addition to BLS (as a VRF) for committee membership proofs, but this is completely independent of Leios requirements. Furthermore, the proof-of-possession mechanisms required for BLS aggregation are identical across both protocols, allowing for shared implementation and validation procedures.

Protocol-level interactions between Leios certified endorser blocks and Peras boosted blocks represent a longer-term research opportunity. In principle, the vote aggregation mechanisms used for endorser block certification could potentially be leveraged for Peras boosting, creating a unified voting infrastructure. However, such integration is likely undesirable for initial deployments due to the complexity it would introduce and the dependency it would create between the two protocols. In the medium to long term, exploring these interactions could yield further improvements to both throughput and finality properties.

4.3 Era and hard-fork coordination

As already identified in the impact analysis, Leios requires a new ledger era to accommodate the modified block structure and validation rules. The timing of this transition must be carefully coordinated with the broader Cardano hard-fork schedule and other planned protocol upgrades.

At the time of writing, the currently deployed era is `Conway`, with `Dijkstra` planned as the immediate successor. Current plans for `Dijkstra` include nested transactions and potentially Peras integration. Before that, an intra-era hard fork is planned for early 2026 to enable additional features within the `Conway` era still.

A new era is always required when the allowed encoding of block bodies and transactions change. As `Dijkstra` is the current staging era, it will also be the integration point for Leios-specific format changes. Should development timelines turn out not to align with the inter-era hard-fork schedule to `Dijkstra`, there are two options:

- Postpone Leios deployment until after `Dijkstra`, moving Leios block format changes into the subsequent era `Euler`.
- Leios block format encoding specification and implementation remains in `Dijkstra`, but ledger validation is always failing until an intra-era hard-fork enables it.

While the first option appears cleaner, it could introduce substantial delays depending on the community-agreed pace on new era definition and deployments. The second option on the other hand requires definite understanding on the serialization format ahead of time, where any further change would result in option one of targeting `Euler`, but with the added friction of feature-flagging Leios functionality before its moved to `Euler` - the worst of both options.

Deploying both Peras and Leios within the same hard fork is technically possible but increases deployment risk. Both protocols represent significant consensus changes that affect network communication patterns, resource utilization, and operational procedures. The complexity of coordinating these changes, validating their interactions, and managing the upgrade process across the diverse Cardano ecosystem suggests that sequential deployment provides a more conservative and manageable approach. Both options above would allow for that via two subsequent protocol versions, but also both in one hard-fork if the risk is deemed acceptable.

4.4 Interactions with Genesis

Ouroboros Genesis enables nodes to bootstrap safely from the genesis block with minimal trust assumptions, completing the decentralization of Cardanos physical network infrastructure. Genesis integration with Leios requires **no** changes to the existing Genesis State Machine, though practical considerations on synchronization remain important.

Genesis compatibility directly follows from the protocol design. The Genesis protocol operates on ranking block headers for chain density calculations and bootstrapping decisions. Since Leios preserves the existing ranking block sequence while only adding certificates of endorser blocks, the fundamental Genesis mechanisms remain unchanged. No modifications to the Genesis State Machine are expected, as it continues to evaluate the unchanged chain growth.

Chain synchronization in general becomes more complex under Leios due to the multi-layered block structure. Syncing nodes must fetch both ranking blocks and their associated certified endorser blocks to construct a complete view of the chain. A node that downloads only ranking blocks cannot reconstruct the complete ledger state, as the actual transactions content resides within closures of the endorser blocks referenced by certificates on the ranking blocks. The `LeiosFetch` mini-protocol addresses this requirement through the `MsgLeiosBlockRangeRequest` message type, enabling efficient batch fetching of complete block ranges during synchronization. This allows nodes to request not only a range of ranking blocks but also all associated endorser blocks and their transaction closures in coordinated requests. Parallel fetching from multiple peers becomes critical for synchronization performance, as the data volume substantially exceeds that of traditional Praos blocks.

![WARNING]

TODO: Chain synchronization / syncing node discussion could be moved to the respective section in the architecture/changes chapter

4.5 Impact on Mithril

While Mithril operates as a separate layer above the consensus protocol and does not directly interact with Leios mechanisms, the integration requires consideration of several practical compatibility aspects.

The most prominent feature of Mithril is that it serves verifable snapshots of the `cardano-node` databases. The additional data structures introduced by Leios (e.g. the EBStore) must be incorporated into the snapshots that Mithril produces and delivers to its users. Beyond that, Mithril needs to also extend its procedures for digesting and verifying the more complex chain structure including endorser blocks and their certification status.

Mithril relies on a consistent view of the blockchain across all participating signers. Hence, the client APIs used by Mithril signers may require updates depending on which interfaces are utilized. While Mithril initially focused on digesting and signing the immutable database as persisted on disk, the consideration of using `LocalChainSync` for signing block ranges introduces potential interaction points with Leios induced changes to client interface.

In summary, Leios will not require fundamental changes to Mithrils architecture but requires careful attention to data completeness and consistency checks in the snapshot generation and verification processes.

5 Risks and mitigations

[!WARNING]

TODO: Introduce chapter as being the bridge between implementation plan and concrete technical design; also, these are only selected aspects that inform the implementation (and not cover principal risks to the protocol or things that are avoided by design)

5.1 Key threats

[!WARNING]

TODO: Selection of key threats and attacks that further inform the design and/or implementation plan. Incorporate / reference the full threat model

5.1.1 Protocol bursts

[!WARNING]

TODO: important because

- was a prominent case in research
- acknowledges the wealth of data to be processed
- mitigation: freshest-first delivery / prioritization between praos and leios traffic
- motivates experiments/features revolving around resource management
- reference/include/move related RSK-.. items from impact analysis

5.1.2 Data withholding

[!WARNING]

TODO: important because

- can be done from stake- and network-based attackers
- trivially impacts high-throughput because no certifications happening
- however, more advanced, potential avenue to attack blockchain safety (impact praos security argument) when carefully partitioning the network
- mitigation: L_diff following the security argument
- motivates validation of optimistic and worst-case diffusion paths

5.2 Assumptions to validate early

[!WARNING]

TODO: Which assumptions in the CIP / on the protocol security need to be validated as early as possible?

- Worst case diffusion of EBs given certain honest stake (certifying the EB) is realistic
- The cardano network stack can realize freshest first delivery (sufficiently well)
- A real ledger can (re-)process orders of magnitude higher loads as expected
- ?

6 Technical design

[!CAUTION]

FIXME: The next few sections are basically the relevant parts of the impact analysis and ought to expanded with anything concrete implementation designs.

When transferring things from impact analysis, the **REQ-** requirements are as well as the **NEW-..** and **UPD-..** references were kept. Not sure if we need all of them to references between concepts and designs.

[!WARNING]

TODO: How to structure the changes best? Group them by layer/component or responsibility?

Behavior-based sketch: - Transaction submission and caching - EB production - EB diffusion - EB storage - Voting committee selection - Key generation - Key registration - Key rotation - Vote production - Vote diffusion - Certification - Block production: Including certificates in blocks - Chain validation: Verifying certificates in blocks - Staging area interactions?

See also this mind map of changes as created by @nfrisby:

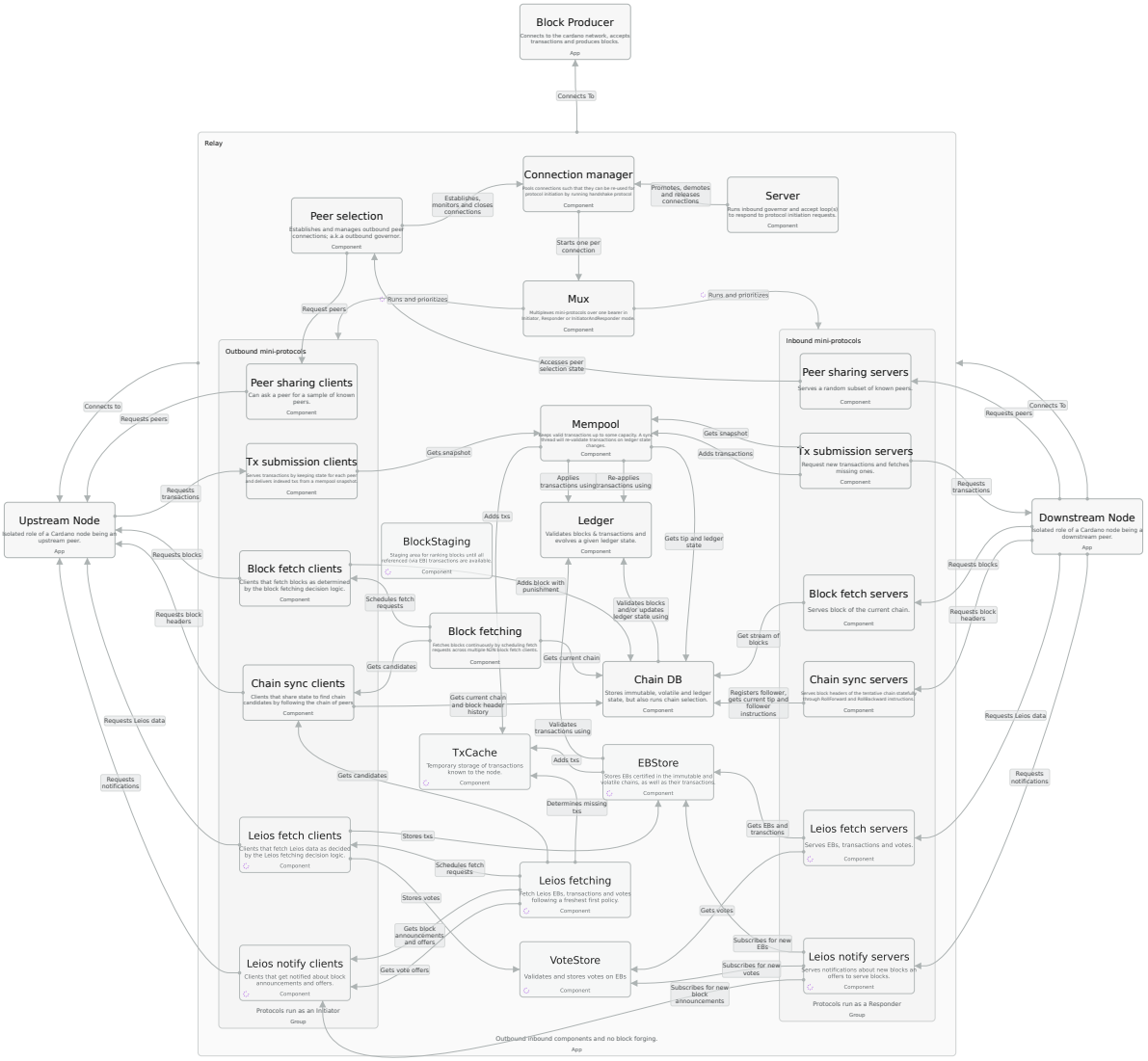
```
mindmap
  root((Leios tasks, core devs))
    ((Ledger))
      Serialization
        Certs in RB bodies<br>- akin to Peras
        Cert codecs/CDDL
      New protocol parameters
      New pool voting keys<br>- akin to Peras
      Cert validation
      New LocalStateQuery queries?
      Tune EB limits
    ((Consensus---easier))
      Serialization
        New fields in RB header
        EB codecs/CDDL
        Vote codecs/CDDL
      Storage
        EBs - imm and vol
        Txs of EBs - imm and vol
        Votes - only vol
        Tx cache
      Vote validation
      Mempool
         Increase size
         Slurp from EBs
      New Tracer events
      New LocalStateQuery queries?
      Add included EBs to NodeToClient ChainSync
```

```
((Consensus---harder))
  Prioritize Praos threads
  Vote decision logic
  Genesis State Machine transition predicates
((Network))
  Prioritize Praos traffic
  Prioritize Praos threads
((Network&Consensus))
  New mini protocols
    Message codecs/CDDL
    Tune size and time limits
    Tune pipelining depth
  Fetch decision logic
    Caught up
    Bulk syncing
  Freshest first delivery
    either: conservative pipelining depths
    and/or else: server-side reordering
((Node))
  New config data
  Feature flags for dev phases
  New CLI queries?
  New pool voting keys<br>- akin to Peras
```

6.1 Architecture

While being a significant change to the consensus protocol, Leios does not require fundamental changes to the overall architecture of the `cardano-node`. Several new components will be needed for the new responsibilities related to producing and relaying Endorser Blocks (EBs) and voting on them, as well as changes to existing components to support higher throughput and freshest-first-delivery. The following diagram illustrates the key components of a relay node where new and updated components are marked in purple:

> [!WARNING]
>
> TODO: Should consider adding Leios prefixes to VoteStore (to not confuse with PerasVoteDB), i.e. LeiosVoteDB?



[!WARNING]

TODO: Explain why focus on relay node (upstream/downstream relationship); briefly mention block producer node differences; Add similar diagram for block producer? block and vote production not shown in relay diagram

6.2 Consensus

[!WARNING]

TODO: Mostly content directly taken from impact analysis. Expand on motivation and concreteness of changes.

CIP-0164 implies functional requirements for the node to issue EBs alongside RBs, vote for EBs according to the rules from the CIP, include certificates when enough votes are seen, diffuse EBs and votes through the network layer, and retain EB closures indefinitely when certified. The

Consensus layer is responsible for driving these operations and coordinating with the Network layer (which implements the actual mini-protocols) to ensure proper diffusion.

6.2.1 Block production

The existing block production thread must be updated to generate an EB at the same time it generates an RB (**UPD-LeiosAwareBlockProductionThread**). In particular, the hash of the EB is a field in the RB header, and so the RB header can only be decided after the EB is decided, and that can only be after the RB payload is decided. Moreover, the RB payload is either a certificate or transactions, and that must also be decided by this thread, making it intertwined enough to justify doing it in a single thread.

- **REQ-IssueLeiosBlocks** The node must issue an EB alongside each RB it issues, unless that EB would be empty.
- **REQ-IncludeLeiosCertificates** The node must include a certificate in each RB it issues if it has seen enough votes supporting the EB announced by the preceding RB. (TODO excluding empty or very nearly empty EBs?)

The Mempool capacity should be increased (**UPD-LeiosBiggerMempool**) to hold enough valid transactions for the block producer to issue a full EB alongside a full RB. The Mempool capacity should at least be twice the capacity of an EB, so that the stake pool issuing a CertRB for a full EB would still be able to issue a full EB alongside that CertRB (TxRBs have less transaction capacity than the EB certified by a CertRB). In general, SPOs are indirectly incentivized to maximize the size of the EB, just like TxRBsso that more fees are included in the epochs reward calculation.

6.2.2 Vote production and storage

A new thread dedicated to Leios vote production (**NEW-LeiosVoteProductionThread**) will wake up when the closure of an EB is newly available. If the voting rules would require the stake pool to vote (now or soon) for this EB if its valid, then this thread will begin validating it. Note if multiple closures arrive simultaneously, at most one of them could be eligible for a vote, since the voting rules require the EB to be announced by the tip of the nodes current selection. If the validation succeeds while the voting rules still require the stake pool to vote for this EB (TODO even if it has since switched its selection?), the thread will issue that vote.

- **REQ-IssueLeiosVotes** The node must vote for EBs exactly according to the rules from the CIP.

A new storage component (**NEW-LeiosVoteStorage**) will store all votes received by a node, up to some conservative age (eg ten minutes). As votes arrive, they will be grouped according to the RB they support. When enough votes have arrived for some RB, the certificate can be generated immediately, which can avoid delaying the potential subsequent issuance of a CertRB by this node. A vote for the EB announced by an RB is irrelevant once all nodes will never switch

their selection away from some block that is not older than that RB. This condition is very likely to be satisfied relatively soon on Cardano mainnet, unless its Praos growth is being disrupted. Therefore, the vote storage component can simply discard votes above some conservative age, which determines a stochastic upper bound the heap size of all sufficiently-young votes.

- **REQ-DiffuseLeiosVotes** The node must diffuse votes (via the Network layers mini-protocols) at least until theyre old enough that there remains only a negligible probability they could still enable an RB that was issued on-time to include a certificate for the EB they support.

6.2.3 Endorser block storage

Unlike votes, a node should retain the closures of older EBs (**NEW-LeiosEbStore**), because Praos allows for occasional deep forks, the most extreme of which could require the closure of an EB that was announced by the youngest block in the Praos Common Prefix. On Cardano mainnet, that RB is usually 12 hours old, but could be up to 36 hours old before CIP-0135 Disaster Recovery Plan is triggered. Thus, EB closures are not only large but also have a prohibitively long lifetime even when theyre ultimately not immortalized by the historical chain. This component therefore stores EBs on disk just as the ChainDB already does for RBs. The volatile and immutable dichotomy can even be managed the same way it is for RBs.

- **REQ-DiffuseLeiosBlocks** The node must acquire and diffuse EBs and their closures (via the Network layers new mini-protocols, see below).
- **REQ-ArchiveLeiosBlocks** The node must retain each EBs closure indefinitely when the settled Praos chain certifies it.

Each CertRB must be buffered in a staging area (**NEW-LeiosCertRbStagingArea**) until its closure arrives, since the VolDB only contains RBs that are ready for ChainSel. (Note that a CertRBs closure will usually have arrived before it did.) (TODO Any disadvantages? For example, would it be beneficial to detect an invalid certificate before the closure arrives?) (TODO a more surgical alternative: the VolDB index could be aware of which EB closures have arrived, and the path-finding algorithm could incorporate that information. However, this means each EB arrival may need to re-trigger ChainSel.) The BlockFetch client (**UPD-LeiosRbBlockFetchClient**) must only directly insert a CertRB into the VolDB if its closure has already arrived (which should be common due to L_diff). Otherwise, the CertRB must be deposited in the CertRB staging area instead.

The LedgerDB (**UPD-LeiosLedgerDb**) will need to retrieve the certified EBs closure from the LeiosEbStore when applying a CertRB. Due to NEW-LeiosCertRbStagingArea, it should be impossible for that retrieval to fail.

6.2.4 Transaction cache

A new storage component (**NEW-LeiosTxCache**) will store all transactions received when diffusing EBs as well as all transactions that successfully enter the Mempool, up to some conservative age (eg one hour). The fundamental reason that EBs refer to transactions by hash instead of including them directly is that, for honest EBs, the node will likely have already received most of the referenced transactions when they recently diffused amongst the Mempools. Thats not guaranteed, though, so the node must be able to fetch whichever transactions are missing, but in the absence of an attack that ought to be minimal.

The Mempool is the natural inspiration for this optimization, but its inappropriate as the actual cache for two reasons: it has a relatively small, multidimensional capacity and its eviction policy is determined by the distinct needs of block production. This new component instead has a greater, unidimensional capacity and a simple Least Recently Used eviction policy. Simple index maintained as a pair of priority queues (index and age) in manually managed fixed size bytearrays, backed by a double-buffered mmapped file for the transactions serializations. Those implementation choices prevent the sheer number of transactions from increasing GC pressure (adversarial load might lead to a ballpark number of 131000 transactions per hour), and persistences only benefit here would be to slightly increase parallelism/simplify synchronization, since persistence would let readers release the lock before finishing their search.

Note: if all possibly-relevant EBs needed to fit in the LeiosTxCache, its worst case size would approach 500 million transactions. Even the index would be tens of gigabytes. This is excessive, since almost all honest traffic will be younger than an hourassuming FFD is actually enforced.

6.2.5 Resource management

The protocol requires resource-management to prioritize Praos traffic and computation over all Leios traffic, and prioritize younger EBs over older ones:

- **REQ-PrioritizePraosOverLeios** The node must prioritize Praos traffic and computation over all Leios traffic and computation so that the diffusion and adoption of any RB is only negligibly slower.
- **REQ-PrioritizeFreshOverStaleLeios** The node must prioritize Leios traffic and computation for younger EBs over older EBs (a.k.a. freshest first delivery).

These requirements can be summarized as: Praos > fresh Leios > stale Leios. The Consensus layer implements the scheduling logic to satisfy these requirements, while the Network layer (see below) implements the protocol mechanisms. Looking forward, Peras should also be prioritized over Leios, since a single Peras failure is more disruptive to Praos than a single Leios failure.

The fundamental idea behind Leios has always been that the Praos protocol is inherently and necessarily bursty. Leios should be able to freely utilize the nodes resources whenever Praos is not utilizing them, which directly motivates **REQ-PrioritizePraosOverLeios**. It is ultimately impossible to achieve such time-multiplexing perfectly, due to the various latencies and hystereses

inherent to the commodity infrastructure (non real-time operating systems, public Internet, etc). On the other hand, it is also ultimately unnecessary to time-multiplex Praos and Leios perfectly, but which degree of imperfection is tolerable?

> [!WARNING]
>
> TODO: Move description of protocol burst attack vector into dedicated section (above)

One particularly relevant attack vector is the protocol burst attack (**ATK-LeiosProtocolBurst**). In a protocol burst attack the adversary withholds a large number of EBs and/or their closures over a significant duration and then releases them all at once. This will lead to a sustained maximal load on the honest network for a smaller but still significant duration, a.k.a. a burst. The potential magnitude of that burst will depend on various factors, including at least the adversarys portion of stake, but the worst-case is more than a gigabyte of download. The cost to the victim is merely the work to acquire the closures and to check the hashes of the received EB bodies and transaction bodies. In particular, at most one of the EBs in the burst could extend the tip of a victim nodes current selection, and so thats the only EB the victim would attempt to fully parse and validate.

Contention for the following primary node resources might unacceptably degrade the time-multiplexing via contention between Praos and Leios:

- **RSK-LeiosPraosContentionNetworkBandwidth** This is not anticipated to be a challenge, because time-multiplexing the bandwidth is relatively easy. In fact, Leios traffic while Praos is idle could potentially even prevent the TCP Receive Window from contracting, thus avoiding a slow start when Praos traffic resumes.
- **RSK-LeiosPraosContentionCPU** This is not anticipated to be a challenge, because todays Praos node does not exhibit major CPU load on multi-core machines. Leios might have more power-to-weight ratio for parallelizing its most expensive task (EB validation), but that parallelization isnt yet obviously necessary. Thus, even Praos and Leios together do not obviously require careful orchestration on a machine with several cores.
- **RSK-LeiosPraosContentionGC** It is not obvious how to separate Praos and Leios into separate OS processes, since the ledger state is expensive to maintain and both protocols frequently read and update it. When the Praos and Leios components both run within the same operating system process, they share a single instance of the GHC Runtime System (RTS), including eg thread scheduling and heap allocation. The sharing of the heap in particular could result in contention, especially during an ATK-LeiosProtocolBurst (at least the transaction cache will be doing tens of thousands of allocations, in the worst-case). Even if the thread scheduler could perfectly avoid delaying Praos threads, Leios work could still disrupt Praos work, because some RTS components exhibit hysteresis, including the heap.
- **RSK-LeiosPraosContentionDiskBandwidth** Praos and Leios components might contend for disk bandwidth. In particular, during a worst-case ATK-LeiosProtocolBurst, the

Leios components would be writing more than a gigabyte to disk as quickly as the network is able to acquire the bytes (from multiple peers in parallel). Praoss disk bandwidth utilization depends on the leader schedule, fork depth, etc, as well as whether the node is using a non-memory backend for ledger storage (aka UTxO HD or Ledger HD). For non-memory backends, the ledgers disk bandwidth varies drastically depending on the details of the transactions being validated and/or applied: a few bytes of transaction could require thousands of bytes of disk reads/writes.

  - Note that the fundamental goals of Leios will imply a significant increase in the size of the UTxO. In response, SPOs might prefer enabling UTxO HD/Ledger HD over buying more RAM.

Both GC pressure and disk bandwidth are notoriously difficult to model and so were not amenable to the simulations that drove the first version of the CIP. Prototypes rather than simulations will be necessary to assess these risks with high confidence.

The same risks can also be viewed from a different perspective, which is more actionable in terms of planning prototypes/experiments/etc: per major component of the node.

- **RSK-LeiosLedgerOverheadLatency**: Parsing a transaction, checking it for validity, and updating the ledger state accordingly all utilize CPU and heap (and also disk bandwidth with UTxO/Ledger HD). Frequent bursts of that resource consumption proportional to 15000% of a full Praos block might disrupt the caught-up node in heretofore unnoticed ways. Only syncing nodes have processed so many/much transactions in a short duration, and latency has never been a fundamental priority for a syncing node. Disruption of the RTS is the main concern here, since there is plenty of CPU availablethe ledger is not internally parallelized, and only ChainSel and the Mempool could invoke it simultaneously.
- **RSK-LeiosNetworkingOverheadLatency**: Same as RSK-LeiosLedgerOverheadLatency, but for the Diffusion Layer components handling frequent 15000% bursts in a caught-up node.
- **RSK-LeiosMempoolOverheadLatency**: Same as RSK-LeiosLedgerOverheadLatency, but for the Mempool frequently revalidating 15000% load in a caught-up node during congestion (ie 30000% the capacity of a Praos block, since the Leios Mempool capacity is now two EBs instead of two Praos blocks).

6.2.6 Implementation notes

For the first version of the LedgerDB, it need not explicitly store EBs ledger state; the CertRBs result ledger state will reflect the EBs contents. A second version could thunk the EBs reapplication alongside the announcing RB, which would only avoid reapplication of one EB on a chain switch (might be worth it for supporting tiebreakers?). The first version of LedgerDB can simply reapply the EBs transactions before tick-then-applying a CertRB. A second version should pass the EBs transactions to the ledger function (or instead the thunk of reapplying the EB)?

The first version of the Mempool can be naive, with the block production thread handling everything. A second version can try to pre-compute in order to avoid delays (ie discarding the certified EBs chunk of transactions) when issuing a CertRB and its announced EB.

The first version of LeiosTxCache should reliably cache all relevant transactions that are less than an hour or so oldthat age spans 180 active slots on average. A transaction is born when its oldest containing EB was announced or when it *entered* the Mempool (if it hasnt yet been observed in an EB). (Note that that means some txs age in the LeiosTxCache can increase when an older EB that contains it arrives.) Simple index maintained as a pair of priority queues (index and age) in manually managed fixed size bytearrays, backed by a double-buffered mmapped file for the transactions serializations. Those implementation choices prevent the sheer number of transactions from increasing GC pressure (adversarial load might lead to a ballpark number of 131000 transactions per hour), and persistences only benefit here would be to slightly increase parallelism/simplify synchronization, since persistence would let readers release the lock before finishing their search.

The first version of LeiosFetch client can assemble the EB closure entirely on disk, one transaction at a time. A second version might want to batch the writes in a pinned mutable `ByteArray` and use `withMutableByteArrayContents` and `hPutBuf` to flush each batch. Again, the possible benefit of this low-level shape would be to avoid useless GC pressure. The first version can wait for all transactions before starting to validate any. A later version could eagerly validate as the prefix arrivescomparable to eliminating one hop in the topology, in the worst-case scenario.

The first version of LeiosFetch server simply pulls serialized transactions from the LeiosEbStore, and only sends notifications to peers that are already expecting them when the noteworthy event happens. If notification requests and responses are decoupled in a separate mini protocol *or else* requests can be reordered (TODO or every other request supports a MsgOutOfOrderNotificationX loopback alternative?), then itll be trivial for the client to always maintain a significant buffer of outstanding notification requests.

Even the first version of LeiosFetch decision logic should consider EBs that are certified on peers ChainSync candidates as available for request, as if that peer had sent both MsgLeiosBlockOffer and MsgLeiosBlockTxsOffer. A MsgRollForward implies the peer has selected the block, and the peer couldnt do that for a CertRB if it didnt already have its closure.

The first version of LeiosEbStore can just be two bog standard key-value stores, one for immutable and one for volatile. A second version maybe instead integrates certified EBs into the existing ImmDB? That integration seems like a good fit. It has other benefits (eg saves a disk roundtrip and exhibits linear disk reads for driver prefetching/etc), but those seem unimportant so far.

6.3 Network

[!WARNING]

TODO: Mostly content directly taken from impact analysis. Expand on motivation and concreteness of changes.

The Network layer implements the mini-protocols that enable the Consensus layer to satisfy its diffusion requirements (**REQ-DiffuseLeiosBlocks**, **REQ-DiffuseLeiosVotes**) and prioritization requirements (**REQ-PrioritizePraosOverLeios**, **REQ-PrioritizeFreshOverStaleLeios**) defined in the Consensus section above. While Consensus drives the scheduling logic for when to diffuse blocks and votes, Network provides the protocol mechanisms to actually transmit them over the peer-to-peer network.

Similar resource contention risks apply to the Network layer, including network bandwidth contention between Praos and Leios, networking overhead latency, and contention between fresh and stale Leios traffic.

6.3.1 New mini-protocols

The node must include new mini-protocols (**NEW-LeiosMiniProtocols**) to diffuse EB announcements, EBs themselves, EBs transactions, and votes for EBs. These protocols enable the Consensus layer to satisfy **REQ-DiffuseLeiosBlocks** and **REQ-DiffuseLeiosVotes**. The Leios mini-protocols will require new fetch decision logic (**NEW-LeiosFetchDecisionLogic**), since the node should not necessarily simply download every such object from every peer that offers it. Such fetch decision logic is also required for TxSubmission and for Peras votes; the Leios logic will likely be similar but not equivalent.

6.3.2 Traffic prioritization

The existing multiplexer is intentionally fair amongst the different mini-protocols. In the current CIP, the Praos traffic and Leios traffic are carried by different mini-protocols. Therefore, introducing a simple bias in the multiplexer (**NEW-LeiosPraosMuxBias**) to strongly (TODO fully?) prefer sending messages from Praos mini protocols over messages from Leios mini protocols would directly enable the Consensus layer to satisfy **REQ-PrioritizePraosOverLeios** and mitigate **RSK-LeiosPraosContentionNetworkBandwidth**. This multiplexer bias is the primary mechanism to ensure that Praos traffic and computation are prioritized over Leios, so that the diffusion and adoption of any RB is only negligibly slower.

It is not yet clear how best to mitigate **RSK-LeiosLeiosContentionNetworkBandwidth** or, more generally, how to enable the Consensus layer to satisfy **REQ-PrioritizeFreshOverStaleLeios** (aka freshest first delivery) in the Network Layer. One notable option is to rotate the two proposed Leios mini-protocols into a less natural pair: one would send all requests and only requests and the other would send all replies and only replies. In that way, the server canwhen it has received multiple outstanding requests, which seems likelying during ATK-LeiosProtocolBurstreply to requests in a different order than the client sent them, which is inevitable since the client will commonly request an EB as soon its offered, which means the client will request maximally fresh EBs after having requesting less fresh EBs. If the client

were to avoid sending any request that requires a massive atomic reply (eg a MsgLeiosBlockTxsRequest for 10 megabytes), then the server can prioritize effectively even without needing to implement any kind of preemption mechanism. This option can be formulated in the existing mini protocol infrastructure, but another option would be to instead enrich the mini-protocol infrastructure to somehow directly allow for server-side reordering. Whether any of this is needed requires further investigation through prototypes (EXP-LeiosDiffusionOnly).

6.4 Ledger

> [!WARNING]
>
> TODO: Mostly content directly taken from impact analysis. Expand on motivation and concreteness of changes.

The Ledger is responsible for validating Blocks and represents the actual semantics of Cardano transactions. CIP-164 sketches a protocol design that does not change the semantics of Cardano transactions, does not propose any changes to the transaction structure and also not requires changes to reward calculation. The ledger component has three main entry points:

1. Validating individual transactions via `LEDGER`
2. Validating entire block bodies via `BBODY`
3. Updating rewards and other ledger state (primarily across epochs) via `TICK`

The first will not need to change functionally, while the latter two will need to be updated to handle the new block structure (ranking blocks not including transactions directly) and to enable the determination of a voting committee for certificate verification. Any change to the ledger demands a hard-fork and a change in formats or functionality are collected into ledger eras. The changes proposed by CIP-164 will need to go into new ledger era:

- **REQ-LedgerNewEra** The ledger must be prepared with a new era that includes all changes required by CIP-164.

For the remainder of this document, lets assume the changes will go into the `Euler` era, where `Conway` is currently the latest and `Dijkstra` is in preparation at the time of writing.

6.4.1 Transaction validation levels

Validating individual transactions is currently done either via `applyTx` or `reapplyTx` functions. This corresponds to two levels of validation:

- `applyTx` fully validates a transaction, including existence of inputs, checking balances, cryptographic hashes, signatures, evaluation of plutus scripts, etc.
- `reapplyTx` only check whether a transaction applies to a ledger state. This does not include expensive checks like script evaluation (a.k.a phase-2 validation) or signature verification.

Where possible, `reapplyTx` is used when we know that the transaction has been fully validated before. For example when refreshing the mempool after adopting a new block. With Leios, a third level of validation is introduced:

- **REQ-LedgerTxNoValidation** The ledger should provide a way to update the ledger state by just applying a transaction without validation.

This third way of updating a ledger state would be used when we have a valid certificate about endorsed transactions in a ranking block. To avoid delaying diffusion of ranking blocks, we do want to do the minimal work necessary once an EB is certified and ease the protocol security argument with:

- **REQ-LedgerCheapReapply** Updating the ledger state without validation must be significantly cheaper than even reapplying a transaction is today.

Note that this already anticipates that the new, third level `notValidateTx` will be even cheaper than `reapplyTx`. Existing benchmarks indicate that `reapplyTx` is already at least one order of magnitude cheaper than `applyTx` for transactions.

6.4.2 New block structure

In Praos, all transactions to be verified and applied to the ledger state are included directly in the block body. In Leios, ranking blocks (RB) may not include transactions directly, but instead certificate and reference to an endorsement block (EB) that further references the actual transactions. This gives rise to the following requirements:

- **REQ-LedgerResolvedBlockValidation** When validating a ranking block body, the ledger must be provided with all endorsed transactions resolved.
- **REQ-LedgerUntickedEBValidation** When validating a ranking block body, the ledger must validate endorsed transactions against the ledger state before updating it with the new ranking block.
- **REQ-LedgerTxValidationUnchanged** The actual transaction validation logic should remain unchanged, i.e., the ledger must validate each transaction as it does today.

The endorsement block itself does not contain any additional information besides a list of transaction identifiers (hashes of the full transaction bytes). Hence, the list of transactions is sufficient to reconstruct the EB body and verify the certificate contained in the RB. The actual transactions to be applied to the ledger state must be provided by the caller of the ledger interface, typically the storage layer.

6.4.3 Certificate verification

In order to verify certificates contained in ranking blocks, the ledger must be aware of the voting committee and able to access their public keys. As defined by **REQ-RegisterBLSKeys**, SPOs must be able to register their BLS keys to become part of the voting committee. The ledger will

need to be able to keep track of the registered keys and use them to do certificate verification. Besides verifying certificates, individual votes must also be verifiable by other components (e.g. to avoid diffusing invalid votes).

- **REQ-LedgerStateVotingCommittee** The Leios voting committee must be part of the ledger state, updated on epoch boundaries and queryable through existing interfaces.

Being part of the ledger state, the voting committee will be stored in ledger snapshots and hence on disk in course of Ledger-HD. Depending on how exactly keys will be registered, the ledger might need to be able to access block headers in order to read the BLS public keys from the operational certificate. As this is not the case today (only block bodies are processed by the ledger), this results in requirement:

- **REQ-LedgerBlockHeaderAccess** The ledger must be able to access block headers.

  [!NOTE] This is a very generic requirement and will likely change depending on how the consensus/ledger interface for block validation is realized. It might be desirable to limit the ledgers access to block headers and only provide (a means to extract) relevant information. That is, the BLS public keys to be tracked and the voting committee to be selected from.

The voting committee consists of persistent and non-persistent voters. The persistent voters are to be selected at epoch boundaries using a Fait Accompli sortition scheme. Hence:

- **REQ-LedgerCommitteeSelection** The ledger must select persistent voters in the voting committee at epoch boundaries using the Fait Accompli sortition scheme.

Finally, block validation of the ledger can use the voting committee state to verify certificates contained in ranking blocks:

- **REQ-LedgerCertificateVerification** The ledger must verify certificates contained in ranking blocks using the voting committee state.

6.4.4 New protocol parameters

CIP-164 introduces several new protocol parameters that may be updated via on-chain governance. The ledger component is responsible for storing, providing access and updating any protocol parameters. Unless some of the new parameters will be deemed constant (a.k.a globals to the ledger), the following requirements must be satisfied for all new parameters:

- **REQ-LedgerProtocolParameterAccess** The ledger must provide access to all new protocol parameters via existing interfaces.
- **REQ-LedgerProtocolParameterUpdate** The ledger must be able to update all new protocol parameters via on-chain governance.

Concretely, this means defining the `PParams` and `PParamsUpdate` types for the `Euler` era to include the new parameters, as well as providing access via the `EulerPParams` and other type classes.

6.4.5 Serialization

Traditionally, the ledger component defines the serialization format of blocks and transactions. CIP-164 introduces three new types that need to be serialized and deserialized:

> [!WARNING]
>
> TODO: Serialization of votes a consensus component responsibility?

- **REQ-LedgerSerializationRB** The ranking block body contents must be deterministically de-/serializable from/to bytes using CBOR encoding.
- **REQ-LedgerSerializationEB** The endorsement block structure must be deterministically de-/serializable from/to bytes using CBOR encoding.
- **REQ-LedgerSerializationVote** The vote structure must be deterministically de-/serializable from/to bytes using CBOR encoding.

Corresponding types with instances of `EncCBOR` and `DecCBOR` must be provided in the ledger component. The `cardano-ledger` package is a dependency to most of the Haskell codebase, hence these new types can be used in most other components.

6.5 Cryptography

> [!WARNING]
>
> TODO: Mostly content directly taken from impact analysis. Expand on motivation and concreteness of changes.

The security of the votes cast and the certificates that Leios uses to accept EB blocks depends on the usage of the pairing-based BLS12-381 signature scheme (BLS). This scheme is useful, as it allows for aggregation of public keys and signatures, allowing a big group to signal their approval with one compact artifact. Besides Leios, it is also likely that Peras will use this scheme.

This section derives requirements for adding BLS signatures to `cardano-base` and sketches changes to satisfy them. The scope is limited to cryptographic primitives and their integration into existing classes; vote construction/logic is out of scope. This work should align with this IETF draft.

> Note that with the implementation of CIP-0381 `cardano-base` already contains basic utility functions needed to create these bindings; the work below is thus expanding on that. The impact of the below requirements thus only extends to this module and probably this outward facing class.

6.5.1 Core functionality

The following functional requirements define the core BLS signature functionality needed:

- **REQ-BlsTypes** Introduce opaque types for `SecretKey`, `PublicKey`, `Signature`, and `AggSignature` (if needed by consensus).
- **REQ-BlsKeyGenSecure** Provide secure key generation with strong randomness requirements, resistance to side-channel leakage.
- **REQ-BlsVariantAbstraction** Support both BLS variantssmall public key and small signaturebehind a single abstraction. Public APIs are variant-agnostic.
- **REQ-BlsPoP** Proof-of-Possession creation and verification to mitigate rogue-key attacks.
- **REQ-BlsSkToPk** Deterministic sk pk derivation for the chosen variant.
- **REQ-BlsSignVerify** Signature generation and verification APIs, variant-agnostic and domain-separated (DST supplied by caller). Besides the DST, the interface should also implement a per message augmentation.
- **REQ-BlsAggregateSignatures** Aggregate a list of public keys and signatures into one.
- **REQ-BlsBatchVerify** Batch verification API for efficient verification of many `(pk, msg, sig)` messages.
- **REQ-BlsDSIGNIntegration** Provide a `DSIGN` instance so consensus can use BLS via the existing `DSIGN` class, including aggregation-capable helpers where appropriate.
- **REQ-BlsSerialisation** Deterministic serialisation: `ToCBOR`/`FromCBOR` and raw-bytes for keys/signatures; strict length/subgroup/infinity checks; canonical compressed encodings as per the Zcash standard for BLS points.
- **REQ-BlsConformanceVectors** Add conformance tests using test vectors from the initial Rust implementation to ensure cross-impl compatibility.

6.5.2 Performance and quality

> [!WARNING]
>
> TODO: Move to performance or testing sections?

The following non-functional requirements ensure the implementation meets performance and quality standards:

- **REQ-BlsPerfBenchmarks** Benchmark single-verify, aggregate-verify, and batch-verify; report the impact of batching vs individual verification.
- **REQ-BlsRustParity** Compare performance against the Rust implementation; document gaps and ensure functional parity on vectors.
- **REQ-BlsDeterminismPortability** Deterministic results across platforms/architectures; outputs independent of CPU feature detection.
- **REQ-BlsDocumentation** Document the outward facing API in cardano-base and provide example usages. Additionally add a section on dos and donts with regards to security of this scheme outside the context of Leios.

6.5.3 Implementation notes

Note that the PoP checks probably are done at the certificate level, and that the above-described API should not be responsible for this. The current code on BLS12-381 already abstracts over both curves `G1`/`G2`, we should maintain this. The `BLST` package also exposes fast verification over many messages and signatures + public keys by doing a combined pairing check. This might be helpful, though its currently unclear if we can use this speedup. It might be the case, since we have linear Leios, that this is never needed.

6.6 Performance & Tracing (P&T)

> [!WARNING]
>
> TODO: Mostly content directly taken from impact analysis. Expand on motivation and concreteness of changes. We could also consider merging performance engineering aspects into respective layers/components. This also feels a bit ouf of touch with the implementation plan; to be integrated better for a more holistic quality and performance strategy. See also https://github.com/input-output-hk/ouroboros-leios/pull/596 for more notes on performance & tracing.

This outlines Leios impact on the nodes tracing system and on dedicated Leios performance testing and benchmarks.

6.6.1 Tracing

Leios will require a whole new set of observables for a Cardano node, which do not exist for Praos. These observables will need to be exposed - just as the existing ones - via trace evidence and metrics. A specification document will need to be created and maintained, detailing the semantics of those new observables. Some might be specific to the Haskell implementation, some might be generic to any Leios implementation. The work from R&D and the insights gained from Leios simulations will be the input to that document.

During Leios implementation process, P&T will need to oversee that traces are emitted at appropriate source locations wrt. their semantics, as well as properly serialized or captured in a metric in `cardano-node` itself. P&T analysis tooling - mostly the `locli` package - will need significant adjustment to parse, process and extract meaningful performance data from raw trace evidence.

6.6.2 Performance testing

For a systematic approach to benchmarking, all Leios modes of operation and their respective configurations will need to be captured in P&Ts benchmark profile library - the `cardano-profile` package. P&Ts `nix` & `Nomad` based automations need to be adjusted to deploy and execute Leios profiles as benchmarks from that library.

On a conceptual level, the challenge to benchmarking Leios - being built for high throughput - is putting it under a stable saturation workload for an extended period of time. By stable, Im referring to maintaining equal submission pressure over the benchmarks entire duration. These workloads need to be synthetic in nature, as only that way one can reliably and consistently stress specific aspects of the implementation. For Praos benchmarks, theyre created dynamically by `tx-generator`. New workloads will need to be implemented, or derived from the existing ones.

Considering all the above, the most promising approach would be finding a model, or symmetrically scaled-down Leios, which is able to reliably predict performance characteristics of the non-scaled down version - exactly as P&Ts benchmarking cluster hardware models a larger environment like `mainnet` at scale and is able to predict performance impact based on observations from the cluster. By Leios version above, Im of course referring to the exact same Leios implementation whose performance characteristics are being measured. Model or scaled versions will have to be realized via configuration or protocol parameters exclusively.

Any Leios option or protocol parameter that allows for sensibly scaling the implementation has to be identified. This will allow for correlating observed performance impact or trade-offs to e.g. linearly scaling some parameter. Comparative benchmarking will require a clearly structured process of integrating new features or changes into the implementation. When many changes are convoluted into one single benchmarkable, it gets increasingly difficult to attribute an observation to a single change - in the worst case, an optimization can obscure a regression when both are introduced in the same benchmarkable.

Finding a model / scaled Leios version is an iterative process which requires continuous validation. It will require P&T to be in constant, close coordination with both implementors and researchers.

6.7 End-to-end testing

> [!WARNING]
>
> TODO: Mostly content directly taken from impact analysis. Expand on motivation and concreteness of changes. This also feels a bit ouf of touch with the implementation plan; to be integrated better for a more holistic quality and performance strategy.

The cardano-node-tests project offers test suites for end-to-end functional testing and mainnet synchronization testing.

The end-to-end functional test suite checks existing functionalities. It operates on both locally deployed testnets and persistent testnets such as Preview and Preprod. With over 500 test cases, it covers a wide spectrum of features, including basic transactions, reward calculation, governance actions, Plutus scripts and chain rollback.

Linear Leios primarily impacts the consensus component of `cardano-node`, leaving end-user experience and existing functionalities unchanged. Consequently, the current test suite can largely be used to verify `cardano-node`s operation after the Leios upgrade, requiring only minor adjustments.

6.7.1 New end-to-end tests for Leios

New end-to-end tests for Leios will focus on two areas:

- Hard-fork testing from the latest mainnet era to Leios
- Upgrading from the latest mainnet `cardano-node` release to a Leios-enabled release

6.7.2 New automated upgrade testing test suite

The suite will perform the following actions:

1. **Initialize Testnet** - Spin up a local testnet, starting in the Byron era, using the latest mainnet `cardano-node` release.
2. **Initial Functional Tests** - Run a subset of the functional tests.
3. **Partial Node Upgrade** - Upgrade several block-producing nodes to a Leios-enabled release.
4. **Mid-Upgrade Functional Tests** - Run another subset of the functional tests.
5. **Cooperation Check** - Verify seamless cooperation between nodes running the latest mainnet `cardano-node` release and those running a Leios-enabled release on the same testnet.
6. **Full Node Upgrade** - Upgrade the remaining nodes to a Leios-enabled release.
7. **Leios Hard-Fork** - Perform the hard-fork to Leios.
8. **Post-Hard-Fork Functional Tests** - Run a final subset of the functional tests.

6.8 Node-to-client

> [!WARNING]
>
> TODO: concrete discussion on how the `cardano-node` will need to change on the N2C interface, based on client interfaces
>
> - Mithril, for example, does use N2C `LocalChainSync`, but does not check hash consistency and thus would be compatible with our plans.

7 Glossary

Term	Definition
RB	Ranking Block - Extended Praos block that announces and certifies EBs
EB	Endorser Block - Additional block containing transaction references
CertRB	Ranking Block containing a certificate
TxRB	Ranking Block containing transactions
BLS	Boneh-Lynn-Shacham signature scheme using elliptic curve cryptography
BLS12-381	Specific elliptic curve used in cryptography
PoP	Proof-of-Possession - Prevents rogue key attacks in BLS aggregation
$L_{\mathbf{hdr}}$	Header diffusion period (1 slot)
$L_{\mathbf{vote}}$	Voting period (4 slots)
$L_{\mathbf{diff}}$	Certificate diffusion period (7 slots)
FFD	Freshest-First Delivery - Network priority mechanism
ATK-LeiosProtocolBurst	Attack where adversary withholds and releases EBs simultaneously

8 References

[!WARNING] TODO: Use pandoc-compatible citations https://pandoc.org/MANUAL.html#citation-syntax

1. **CIP-164**: Ouroboros Linear Leios - Greater transaction throughput https://github.com/cardano-scaling/CIPs/blob/leios/CIP-0164/README.md
2. **Leios Impact Analysis**: High-level component design https://github.com/input-output-hk/ouroboros-leios/blob/main/docs/ImpactAnalysis.md
3. **Leios Formal Specification**: Agda implementation https://github.com/input-output-hk/ouroboros-leios/tree/main/formal-spec
4. **Cardano Ledger Formal Specification**: https://github.com/IntersectMBO/formal-ledger-specifications/
5. **Ouroboros Peras Technical Design**: Finality improving consensus upgrade https://tweag.github.io/cardano-peras/peras-design.pdf
6. **Ouroboros Genesis**: Bootstrap mechanism https://iohk.io/en/research/library/papers/ouroboros-genesis-composable-proof-of-stake-blockchains-with-dynamic-availability/
7. **Ouroboros Network Specification**: https://ouroboros-network.cardano.intersectmbo.org/pdfs/networ spec/network-spec.pdf

8. **UTxO-HD Design**: https://github.com/IntersectMBO/ouroboros-consensus/blob/main/docs/website/developers/utxo-hd/index.md