

MASTERING CARDANO



Mastering Cardano

Lars Brünjes, Joshua Ellul

Version v0.1

Table of Contents

1. Introduction	1
1.1. Where it all started, Bitcoin.....	1
1.2. Blockchain and smart contracts	2
1.3. Overview of a blockchain system	3
1.4. The importance of program correctness for cryptocurrency ledgers.....	5
1.5. Features and benefits of Haskell.....	15
2. Cryptography	24
2.1. Hashing functions.....	24
2.2. Encryption techniques.....	27
2.3. Digital signatures	31
2.4. Its role in securing the blockchain	33
2.5. Pointers to cryptography resources.....	35
3. Learn About Cardano	36
3.1. What is Cardano?	36
3.2. Who built Cardano?	36
3.3. Core principles of Cardano	37
3.4. Core pillars: security, scalability, and interoperability	39
3.5. Cardano community and ecosystem growth	42
3.6. Educating the world about Cardano	43
3.7. Suggested Reading	51
4. How Cardano Works	52
4.1. Cardano node and system layers	52
4.2. The EUTXO model.....	63

4.3. Reaching Consensus using Proof-of-Stake	86
4.4. Incentives	90
4.5. Ouroboros Consensus	106
4.6. How does Proof-of-Work select a validator (miner) ?	110
4.7. In conclusion	117
4.8. Overview of Cardano Network Protocols	118
4.9. A Brief History of Cardano Networking	118
5. Cardano governance	128
5.1. Cardano Improvement Proposals (CIPs)	128
5.2. Project Catalyst: democratizing innovation in Cardano	130
5.3. The age of Voltaire	134
5.4. Intersect: shaping Cardano's future	140
5.5. Intersect structure	145
5.6. Cardano governance: a three-part approach	151
5.7. Cardano governance flow	154
5.8. Governance actions	155
5.9. Registering as a DRep on-chain	156
5.10. SanchoNet: testing ground for Cardano's future	158
5.11. Governance tools	159
5.12. From theory to practice	161
5.13. Journey to ratification	163
5.14. PRAGMA	171
5.15. Looking forward	171
6. Wallets in the world of Cardano	174
7. 6.1 Fundamentals of crypto wallets	175
7.1. 6.1.1 Types of wallets	176
7.2. 6.1.2 Hardware wallets	177
7.3. 6.1.3 Software wallets	179
7.4. 6.1.4 Paper wallets	181
7.5. 6.1.5 Public and private keys in the context of wallets	184

7.6. 6.1.6 Wallet addresses	185
7.7. 6.1.7 Creating a wallet address	186
8. 6.2 Wallets in the Cardano Ecosystem	188
8.1. 6.2.1 Cardano wallets are designed for ada	188
8.2. 6.2.2 Types of Cardano wallets	188
8.3. 6.2.3 Staking and delegation	190
8.4. 6.2.4 Security features	190
8.5. 6.2.5 Integration with DApps	192
8.6. 6.2.6 Cardano wallets vs other blockchain wallets	192
8.7. 6.2.8 Exploring Cardano wallets	193
8.8. 6.2.9 Setting up a Cardano light wallet	202
8.9. 6.2.10 Best practices to secure and back up wallets	204
9. 6.3 Common operations	206
9.1. 6.3.1 Sending and receiving digital assets on Cardano	207
9.2. 6.3.2 Staking ada	210
9.3. 6.3.3 Governance	212
10. Stake Pools and Stake Pool Operation	214
11. Introduction	215
12. What is a Stake Pool?	216
12.1. Stake Pool Roles	217
12.2. Keys	218
12.3. Addresses	220
12.4. Pool Saturation	221
12.5. Pledge vs Stake	222
12.6. Fee Structure	224
13. SPO Requirements	226
13.1. Linux	226
13.2. Networking	227
13.3. Documentation and Learning	227
13.4. Getting Started	227

13.5. Putting it all together, long time maintenance	228
14. Assigning Leadership Slots to Stake Pools	230
14.1. Overview	230
14.2. Epochs, Blocks, and Slots	230
14.3. Playing Limbo	231
14.4. Security	232
15. Slot Battles, Height Battles, Forkers and Propagation	233
15.1. Ouroboros leader selection review	233
15.2. Types of battles	233
15.3. Resolution of battles and forks	235
15.4. Propagation	236
16. <i>cardano-cli</i>	239
16.1. Prologue	239
17. Keys generation	241
17.1. Addresses	241
17.2. Stake Pool related key pairs	245
18. Certificates	252
19. Queries	258
20. Basic transaction	260
20.1. Rewards withdrawal	264
21. Epilogue	267
22. Keeping Time	268
22.1. Comments on the example configuration :	269
23. Server Security and Hardening	271
23.1. System Administration security	271
23.2. Node Security:	274
23.3. Containerized Environments	275
24. Monitoring	279
24.1. Prometheus	279
24.2. Grafana	282

24.3. Alerting with Prometheus and Grafana	283
24.4. Zabbix	285
24.5. RTView	285
24.6. Koios gLiveview	286
24.7. Manual Cardano Log Review	286
24.8. Final Thoughts.....	288
25. Writing smart contracts	290
25.1. Prefaces.....	290
25.2. Smart contract programming languages	295
25.3. Smart contract case studies	301
25.4. Cardano addresses	304
25.5. Marlowe smart contracts	308
25.6. Plutus smart contracts.....	336
25.7. Smart contract security.....	425
26. Decentralized Applications.....	452
26.1. Traditional Web Application Architecture.....	452
26.2. Decentralized Application Architecture	455
26.3. Decentralized Web Storage	489
26.4. DApps and UI/UX Issues	490
27. Looking forward	492
27.1. A Scaling Vision.....	492
27.2. Enter Hydra	495
27.3. Mithril	503
Index	516

Chapter 1. Introduction

The past few years have seen many claims that blockchain, distributed ledger technology (DLT), cryptocurrencies, smart contracts, non-fungible tokens (NFTs) and decentralized autonomous organizations (DAOs) will change society as we know it. Yet, to most people, what these technologies are, what they mean to them, and how they will change society is unclear.

This book aims to demystify these technologies and provide readers with a starting point to use them, develop their own decentralized applications (DApps) and smart contracts/scripts, run infrastructure, or take part in the Cardano ecosystem.

1.1. Where it all started, Bitcoin.

We start with the first proposal that sprouted this decentralized movement, Bitcoin, the first cryptocurrency. Bitcoin was proposed by Satoshi Nakamoto (who remains unknown to date) in a paper released in late 2008. The paper described a novel architecture built using a collection of techniques to solve the double-spending problem—a problem which had previously limited digital currency systems to those that required a centralized server or service provider. We now refer to systems that make use of these techniques as blockchains, which are a type of distributed ledger technology .

Bitcoin's aim was to provide a digital currency system that did not rely on a centralized service provider. This would enable individuals to send and receive digital currency without any intermediary, service provider, or authority that could intervene or would be required to approve

transactions.

1.2. Blockchain and smart contracts

Blockchain, the collection of techniques that solves the double-spending problem, also provides a way of creating digital systems that can execute code in a decentralized manner, where no single administrator, company, entity, owner, or other centralized party has control over the digital system upon which that code executes. Similar to how various software applications can be downloaded and run on computers to allow them to be used for many different tasks, code can be uploaded to a blockchain and run in a decentralized manner for many different purposes. Programs that are uploaded to a blockchain for decentralized execution are called smart contracts or on-chain code/scripts (we'll cover them in more detail later).

Since smart contracts execute on a blockchain, this means that code can be executed in a manner that is tamper-proof and provides guarantees to all users that the system will do exactly what that code is written to do, without anyone having the ability to tamper with the system—including its code and associated data. Smart contracts allow blockchains to be used for much more than just sending and receiving digital currency. This use-case alone can bring positive change to many, through the democratization of and access to financial services to anyone with an internet connected digital device (5 billion people at the time of writing), including 600 million people who are unbanked. But it is not just about financial services. Other applications are being proposed such as decentralized property registries. Such systems would ensure that individuals cannot lose their property just because a paper goes missing, or a database record is deleted.

Other notable use-cases that blockchain is facilitating progress in includes:

- *supply chain management* by enabling transparent tracking of goods

from origin to consumer, thereby enhancing trust and efficiency;

- *digital identity solutions* by empowering individuals to have control over their personal data;
- *digital communities* by ensuring that decisions are made in a transparent and open manner, and by allowing automated actions to follow community decisions;

Being decentralized, blockchain promotes openness and transparency, yet also privacy—achieved through cryptography (which is a building block of cryptocurrencies, as the name implies). Personal details are not required for sending and receiving funds or to interact with other blockchain-based processes. At the same time, anonymous business relationships can be built and promises can be guaranteed and automated through code. For example, investors who back a project could be guaranteed to receive a percentage of income generated. Payments to backers could start automatically once the smart contract receives incoming revenues, without human intervention—all while preserving privacy.

1.3. Overview of a blockchain system

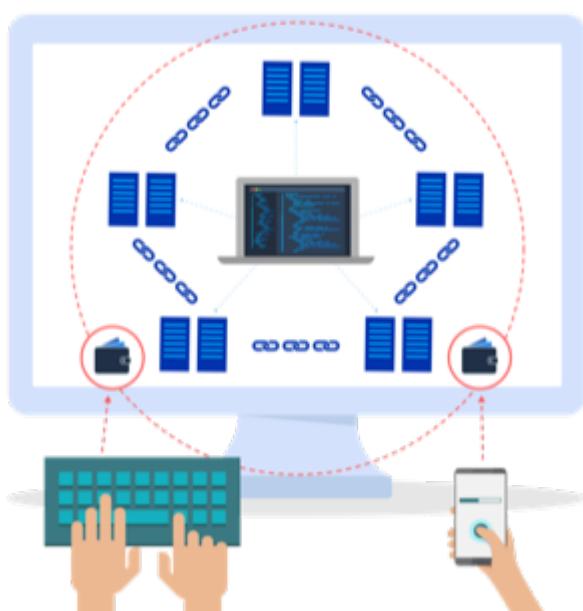


Figure 1. An overview of various blockchain network components and stakeholders.

A high-level overview of a blockchain system is presented in [Figure 1](#). Elements of the system include:

- *Users* typically interact with a blockchain in one of two ways: by sending or receiving funds; or when using decentralized applications (dApps).
- *Wallets* are used to store cryptocurrency and other types of assets, and to authenticate other users. A wallet typically has two core elements. The first is a public address that uniquely identifies the wallet and is used when specifying to which account a transfer is to be made. The second element is a private key, which can be thought of as a password that is used to authorise transfers or any other interaction with the wallet. Wallets are implemented using asymmetric cryptography (see Chapter).
- *Nodes* are computers and other computational devices that work together across the internet to keep the blockchain intact. They ensure that the rules of the blockchain system are not violated, so, for example, a user cannot spend more cryptocurrency than they have.
- *Smart contracts* allow blockchains to be used for much more than just transferring cryptocurrency. In a similar way to software applications on traditional computers, developers (and users) can write and run smart contracts on the blockchain as they deem necessary.

The blockchain can be seen as an append-only ledger where new transactions, such as transfers and any interaction with smart contracts, are stored permanently and cannot be changed—they are immutable. At the same time, the blockchain must ensure that certain rules are upheld to ensure integrity of the ledger. Such rules typically include:

- no one can spend more funds than they have;
- no one can spend another person's funds (unless authorization had been granted);

- old transactions cannot be removed or edited;
- all smart contract code executes exactly as written;
- the ledger, which includes all the history and the current state of the system and smart contracts, is kept intact and ensures only one version of truth exists.

1.4. The importance of program correctness for cryptocurrency ledgers

Guaranteeing correct behavior of cryptocurrency ledgers implemented on blockchain technology comes with a unique set of challenges. An important one stems from the fact that such programs are distributed, meaning that the users running the code participating in implementing its behavior may have conflicting incentives, and the network across which they are communicating may be unpredictable. Testing blockchain applications at scale presents a serious challenge. If issues are discovered post-release, the process of fixing these issues in such software can be very slow and difficult, as it may be impossible to deprecate existing code. It is particularly tricky to deal with past exploits of existing vulnerabilities, which may have resulted in assets being stolen, or even downtime of the entire system. Another reason for paying special attention to correctness is, of course, that users' assets are at stake.

High-assurance guarantees about the behavior of the Cardano platform must be built in from the start, and persist throughout its lifetime. Formal methods (FM) are mathematically rigorous techniques for ensuring program correctness that can provide such guarantees. This section will summarize the impact of using FM on the software development process, specifying when it is appropriate to apply FM. Specific FM techniques, with a focus on FM in Cardano are also covered.

1.4.1. What are formal methods?

The formal methods approach provides guarantees of program behavior

correctness. It involves the application of symbolic logic to construct behavior specifications, define their properties, then verify, prove, or property-test those properties. There are many different FM techniques, and a variety of tools.

A formal specification is a description of what a given program (or a programming language) does, i.e. its semantics, written in symbolic logic and mathematical notation. It may consist of a definition of how the program state (eg a ledger or smart contract state) is updated in the presence of some input. In the case of blockchain, such input can be clock ticks, incoming messages, blocks, or transactions. To consider a non-blockchain example, a *counter* program may have a state that is a natural number, and to apply an incoming increment request, the counter is incremented.

A property is a statement about program behavior, for example, 'when two increment requests arrive, the counter must be incremented twice'. A proof is indisputable logical evidence of the truth of a given property, regardless of all external circumstances. Whenever these symbolic logic formalisms (specifications, properties, proofs, etc) are programmed and checked using software, they are *mechanized*.

1.4.2. What do formal methods give us?

The use of FM does not guarantee some abstract notion of 'perfect program correctness' of the resulting implementation. Rather, applying formal methods means answering the following questions for a specific piece of software being developed, which help get closer to an implementation that has high-assurance guarantees with respect to certain behavior properties:

1. What is the scope of the specification?
2. What is the specification?
3. How to be sure that our formal specification is really what we want?

4. How to show that the implementation also satisfies this specification?

(1). A formal specification rarely models the entire piece of software that will be implementing it. Multiple specifications may be required for a given piece of software. For example, the Cardano platform has separate specifications for networking, consensus, ledger, and smart contract layers. The scope may be further limited by omitting certain functionality. For example, the ledger specification may leave the details of the signing algorithm abstract, giving only an API, or leave out the specifics of transaction serialization entirely. It is essential to be clear about the scope of what is being specified. A reasonable scope is one where there are no unpredictable external side-effects, constraints, or inputs that are not accounted for in the specification.

(2). An FM engineer builds a formal specification by thinking really hard about what they want the program to be doing, sometimes starting by analyzing an informal specification or a requirements list. For example, when a transaction is applied to the ledger, its validity is checked. The definition of transaction validity must be given using a formal specification. This includes details such as 'does the validity interval [a, b] of a transaction include the endpoints a or b?', or 'does a transaction need to specify the exact set of keys that must sign for it to be valid?'. There is not necessarily a 'correct' way to fill in these details - decisions must be made by the engineers or stakeholders in a way that is consistent and suitable for the relevant use cases.

(3). Desirable properties of a program are defined to specify how the program is expected to behave in specific situations. These properties are not an obvious outcome of the specification. They are the result of FM engineers thinking 'what do we want to guarantee about this program's behavior?'. For example, 'the cost and outcome of smart contract execution is predictable locally' is a unique and cherished property of Cardano smart contract execution, and is not necessarily shared by other blockchains. Upon formulating the desired properties of the program, they

need to be either proved or demonstrated with a high degree of assurance, if possible. If it turns out that the specification does not, in fact, satisfy the property, changes to the specification may be required.

Despite the fact that it is not possible to prove – or even list – every single desirable property of the behavior of a given program, this approach results in many important guarantees, as well as helps engineers find bugs in the process – usually way before deployment.

(4). A specification is commonly written using either no software, or a different programming language from the one in which the implementation is written. Whenever a specification is written as a text document, a human must manually check that the specification and implementation match. When a specification is mechanized, it may be possible to establish conformance testing between the two. This automates the comparison between the outputs of the specification and implementation programs for a given set of states and inputs, making conformance testing very useful for keeping the two codebases synced.

1.4.3. FM techniques used in building the Cardano platform

To answer each of the questions posed above, different techniques are applied in Cardano, tuning the level of formal rigor to the particular use case. The most prevalent FM technique within Cardano is the use of *type systems*. A type system is a set of rules that assigns a symbolic property of *type* to each term, e.g. **5** is an **Integer**, **[1 ; 2]** is a **List Integer**, **True** is a **Boolean**, etc. A type system can provide such guarantees as 'all items in this list are integers', or 'it is not possible to add an integer and an error'. A strong type system can provide strong formal guarantees, so, to quote Simon Peyton Jones, type systems are 'the most successful formal method'.

Strong type systems are famously helpful in refactoring code. This is because changing the type of a variable or a function's input or output often results in compilation errors that will prompt the developer to 'follow the types', i.e. to refactor all relevant parts of the program until the

type system accepts it. This often coincides with completion of the required refactoring. For example, if a developer changes the output of some function to allow it to output an error, all functions that call this one will have to also be refactored to account for this additional possibility.

Functional programming, while not itself a kind of FM technique, is a programming paradigm that facilitates writing programs that are highly amenable to formal specification and verification, and is widely used in core Cardano technology. The name functional is chosen because all such programs consist of functions, in the mathematical or logical sense, i.e.

```
f : A -> B  
f (a) = ...
```

where the input type is A , the output type is B , and f associates to each term a of type A a term b of type B . For example, if $A = B$, we can define $f(a) = a$, the identity function. In *pure* functional programs, there is no extra underlying state or input, such as user IO, or variable bindings, available to f within the computation. This significantly simplifies reasoning about the outputs of the function given a certain input (or set of inputs).

Haskell is the programming language of choice for building the core Cardano node technology, including networking, consensus, and ledger components. It is functional, has a strong typing system, and has been around for several decades, boasting a dedicated user and maintainer base. In fact, prior to using proof assistants for specification, Haskell was the language for writing both the specification and the implementation, with the difference between the two being the level of optimization.

Semi-formal, or non-machine-checked, techniques are also in use for building Cardano, including specification and proofs in the form of text documents. For example, for a given UTXO state and transaction, symbolic logic can express:

- How the UTXO state will be updated by this transaction, and
- What properties a valid transaction has.

While doing FM manually is not as reliable as machine-checked techniques, this more lightweight approach is still extremely useful. The Shelley and 'Shelley with multi-assets' Cardano eras had ledger specifications written in this format. Refactoring such specifications to match changes in the implementation is very difficult, as it relies on extremely careful manual analysis rather than 'following the types'.

Proof assistants are software tools that assist with the development of formal proofs and specifications. A proof assistant is made up of an integrated development environment (IDE) and a programming language. For a number of proof assistants, the supported programming language is dependently typed, functional, and treats propositions and logical reasoning as first class citizens. They support constructing and proving propositions, sometimes also supporting some degree of automation for this purpose. They can be used to provide stronger guarantees about the behavior of a given program as compared to type systems that do not support proof construction. Lean, Agda, Coq, and Isabelle are all proof assistants used for different purposes within Cardano. The choice of assistant depends on a number of factors, such as available expertise across engineers, interoperability with other system components, and support for automation.

For example, compare a simple data structure implemented in both Haskell and Agda. **Set** is a data structure that contains a collection of elements of the same type, and each element is unique in this collection. In Haskell, uniqueness of elements is ensured by defining an insertion procedure of an element `e` into a set `S` that does nothing whenever `e` is already contained in `S`, guaranteeing that duplicates do not exist in the collection. Developers then might either rely on the fact that it is easy enough to define insertion correctly or add some test cases. A set – as it is

defined in Agda – is a pair of (i) a collection of elements, and (ii) a proof that there are no duplicates in that collection. This makes it impossible to define a bad insertion procedure that allows accidental element duplication, which is a stronger guarantee than provided by the Haskell implementation.

The use of proof assistants in building Cardano has a good track record for squashing potential bugs before they cause problems in production. For example, in the process of encoding the preservation of value calculation into a proof assistant, an engineer was not able to complete the proof construction. The engineer then realized that it was due to an incorrect assumption made about certain functions, and they were able to fix the specification and the implementation accordingly.

Guarantees obtained via testing alone are not as strong as those of formal verification, such as proofs. Specialized kinds of testing are, nevertheless, integral to the impactful application of FM. The behavior of a specification is defined by constructing *properties*. If we have defined the specification correctly, these properties will be true for any execution of the program. The approach of using a proof assistant might be too heavy-handed for certain applications, i.e. too difficult or time consuming. Instead, special software called *generators* is used for generating extremely large numbers of random valid execution traces, and the desired property is then checked to hold for the generated traces. Generators are tuned to provide better, more realistic coverage. This approach is called *property testing*.

Conformance testing is a specific type of property testing. It provides high-assurance guarantees that the behavior of the implementation for an arbitrary state and input matches the behavior of the specification for the same state and input. Formally proving equivalence between specification and implementation, especially when the implementation is optimized and written in a different language, is quite unrealistic. Conformance testing is useful in formalizing certain aspects of behavioral equivalence, and providing highly reliable (if not indisputable) evidence to support

these claims.

Finally, *specialized formal models*, such as DELTA-Q Systems Development, have been developed to simulate reasoning about real systems before they are implemented and tested. DELTA-Q enables up-front performance modeling. Analysis using this tool can be carried out before creating a prototype to rule out infeasibility early on, and to give realistic performance constraints. For example, it has been used for excluding models that presume the existence of transatlantic network connections that are faster than the speed of light.

Overall, developing new tools, such as the one described above, as well as improving existing ones, is an important component of formal methods application in Cardano, and in industrial contexts in general. Other examples of tool development work done as part of the Cardano engineering include contributions to both Haskell and Agda implementations.

1.4.4. Tuning the level of formality

Not all techniques listed above are suitable for all Cardano components. The idea is to tune the approach to each component and apply heavier techniques with a greater emphasis on verification to the more tractable inner components, and apply a more lightweight approach (type-safety, at minimum) with a greater emphasis on testing the impure outer components.

The strongest formal guarantees can be specified and proved about the *ledger* and *smart contracts* executed on it. This is because both of these components are programmed in a pure way. Moreover, these components are written in a way that allows users to locally compute (ie before submission of a block or transaction) changes that a transaction will make, or the output of a contract. Like the ledger component, the consensus layer comes with certain formally proved properties about its specification. However, like the networking component, it must also deal with

concurrent computation and some unpredictability, so formal verification is not always suitable here.

The networking layer is a component for which it is more difficult to prove properties, since it has to deal with the complexity of unreliable communication. For this reason, it is instead subject to intensive property-based testing. However, because it is leaning heavily on concurrency for efficient operations, even testing proved challenging. Dedicated packages, including a special concurrency control mechanism, were developed to provide an additional layer of abstraction on top of parts of the Haskell runtime system (RTS). The same code can then be executed either by the Haskell RTS, or via a pure and fast implementation that deterministically simulates it, enabling excellent testing.

Additional off-chain components are developed alongside Cardano, which have sufficiently robust formal, statistical, and incentives-based guarantees for the system to rely on them in its operation.

1.4.5. Formal methods process and its impact

Formal methods establish a robust connection between research and implementation. Research papers, together with the associated proof-of-concept implementations, are often difficult to transform into realistic and efficient implementations. FM facilitates this process by specifying exactly what the resulting implementation has to do, while maintaining the same level of scientific rigor as in the original research work. There are also some downsides to the use of FM, so let us list the pros and cons of formal methods:

PROS:

- Provide very strong guarantees about program behavior, such as the absence (or a very low probability) of certain kinds of errors in all program executions
- Multiple implementations built in accordance with a single formal

model are guaranteed to have the same functionality.

CONS:

- Requires a lot of time, resources, effort, and expertise
- May be difficult for non-experts to understand
- Tools are not always easy to use or production-ready
- Usually requires traditional testing to be done alongside proofs
- Difficult to adjust when software updates occur (and may be an afterthought).

There is a common theme here – using formal methods is significantly more difficult compared to traditional QA. However, the resource investment (of time, funds, effort, etc) in FM has been worth it for Cardano, which has been running with zero downtime since its launch in 2017, at least up to the time of publication of this book. Moreover, the core Cardano technology has not endured any major hacks resulting in the theft of assets due to buggy implementation, and rigorous formal specification and verification certainly deserves credit for this.

The Cardano node FM strategy has been a great help in achieving the peace of mind that comes with strong guarantees about program behavior. Additionally, it helped establish a common language for communication between researchers and practitioners, provides a principled way of adding new features, and serves as valuable reference material for future development.

A lot of FM research and even application is done in an academic setting. Making such work possible in an industry setting presents some challenges, as mentioned in the CONS list above. To achieve optimal FM usage in Cardano, FM work has been (i) mechanized, e.g. including Agda specifications and conformance testing, (ii) democratized, i.e. made more accessible to a broader audience, including the Cardano community and internal engineers, (iii) industrialized, i.e. has industry-like development

practices and standards, and (iv) modified to include a broader scope of application of formal methods, e.g. cryptography.

However, more work remains to be done in all of these areas. Further verification of cryptographic protocols would be extremely valuable. Work is ongoing on the application of formal methods in additional areas of Cardano development, including compilation certification, running verified code on-chain, and additional performance and security modeling. Further work is also being done on tool improvement.

1.5. Features and benefits of Haskell

Haskell is a general-purpose, advanced, purely functional programming language. It achieves memory safety through garbage collection and compiles to native code.

It has everything one would expect from a modern programming language:

- Language server protocol (LSP)
- Editor integration (VSCode, Vim, Emacs, etc)
- A package system that manages dependencies and facilitates starting, maintaining, and publishing Haskell projects
- A big community with over 16,000 published packages
- And more.

On top of that, Haskell has a unique set of features that make it a one-of-a-kind programming language. Some features appeal to researchers who want to explore the boundaries of what's possible with programming languages, and others appeal to software engineers who want to build production software. This section explains the features and benefits relevant to people of the second group.

1.5.1. Purity and immutability

Purity in Haskell

Purely functional programming languages treat all computations as evaluations of mathematical functions. In purely functional programming languages, a function will always return the same value for a specific input. This property is amazing because it ensures referential transparency (the ability to replace functions with their definitions) and significantly reduces the cognitive load of the developer in understanding the program.

Purity in Haskell is more nuanced. A completely pure program wouldn't have any real-world effect and would be practically useless. On the other hand, allowing side effects compromises the benefits of purity. Haskell addresses this dilemma with an elegant and practical solution – explicit effects.

In Haskell, effects are explicitly tagged using types. This offers several benefits:

- The developer can write most of the code as pure functions, minimizing exposure to effects
- The developer can selectively allow only the necessary effects
- Referential transparency and equational reasoning are maintained.

The benefit of this approach is profound in terms of reliability and predictability. Because pure functions do not depend on or alter any state outside their scope, they are much easier to reason about. This isolation simplifies debugging and testing, as each function can be considered in isolation without concern for external dependencies or hidden contexts.

Immutability in Haskell

Closely related to purity is the concept of immutability – another cornerstone of Haskell. Immutability means that once a data structure is

created, it cannot be changed. To modify a data structure, a new one with the desired changes is created, leaving the original intact. While this might seem like it would lead to performance issues, Haskell efficiently manages structures under the hood, enabling the creation of high-performance programs.

Immutability eliminates a whole class of bugs related to changing states. In languages with mutable data structures, bugs often arise from unintended side effects where different parts of a program unexpectedly alter shared data. In Haskell, those bugs are impossible because all data structures are immutable. Each component operates on its own data without risk of interference, leading to safer and easier to comprehend code.

Benefits of purity and immutability

- 1. Simplified reasoning.** With pure functions and immutable data, understanding the flow and outcome of a program becomes more straightforward. Each part of the program can be examined in isolation, making it easier to model, design, and reason about.
- 2. Enhanced testability.** Testing in Haskell is often more straightforward because there is no need to mock a global state or set up and tear down test environments. Tests can be run in any order without interfering with each other, reducing the chances of brittle tests.
- 3. Refactoring confidence.** Combining purity and immutability with a powerful type system, refactoring Haskell code is less risky because the compiler's type checks ensure that changes do not introduce inconsistencies. Additionally, the absence of side effects means that changes in one part of the system are less likely to cause unforeseen issues in others.

1.5.2. Rich static type system

Haskell's type system is a defining feature that distinguishes it from many

other programming languages. It is not just a mechanism for checking correctness but a sophisticated framework that enhances every aspect of software development, from design to maintenance.

Haskell employs a static type system, which means that type checking is performed at compile time rather than at runtime. It is also *strongly typed*, preventing the misuse of data types without explicit conversions. More impressively, it supports advanced features like *type inference*, where the compiler can automatically deduce the types of expressions without explicit annotations. It also includes *algebraic data types*, enabling the construction of complex data structures that can be checked at compile time.

Another powerful aspect of Haskell's type system is its support for *type classes* and *generics*. Type classes allow for the definition of generic interfaces that different types can implement, facilitating polymorphic functions that can operate on any data type that supports a certain set of operations. This feature significantly enhances code reusability and flexibility.

Finally, if you need more power, Haskell supports advanced type-level programming such as type families, generalized algebraic data types (GADTs), data kinds, and much more. You can have as much flexibility and precision as you need to define your domain and invariants.

Benefits of Haskell's type system

- 1. Early error detection.** Haskell performs type checks at compile time, identifying many errors that other languages might only catch during runtime. This early detection saves significant debugging and testing time and reduces the risk of runtime failures.
- 2. Program correctness and safety.** Haskell's type system enforces constraints on how functions and data are used, significantly reducing the likelihood of bugs, like type mismatches or unintended type

coercions. This strict enforcement helps ensure program correctness and enhances overall software safety.

3. **Documentation through types.** Types in Haskell serve as a form of documentation. They enhance code clarity by defining the data types that functions expect and produce. This clarity is invaluable for maintaining existing code and onboarding new developers.
4. **Facilitation of refactoring.** The robustness of Haskell's type system makes refactoring a safer process. Developers can make substantial changes to the internals of their code with confidence as long as the code continues to compile. This confidence is crucial for the long-term maintenance and evolution of software projects.
5. **Aid in design.** The type system in Haskell not only checks code but also aids in its design by enforcing a level of thoughtfulness about data types and their interactions. This enforced rigor leads to better-designed, more robust, and maintainable systems.

1.5.3. Lazy evaluation

Lazy evaluation, also known as call-by-need, is a pivotal aspect of Haskell's execution model. Unlike eager evaluation (call-by-value), where expressions are evaluated as soon as they are bound to variables, lazy evaluation postpones this process. This means that when you define a variable in Haskell, you are effectively defining a promise to compute its value when it's required.

The mechanics of lazy evaluation are facilitated through thunks, which are essentially placeholders for expressions that have not yet been evaluated. When a thunk is first accessed, Haskell computes its value and then caches it for subsequent uses, ensuring that each expression is evaluated at most once.

Lazy evaluation was the catalyst of many other language design decisions, and it has an important role in the way the developer structures the code.

It allows developers to better express what they want instead of how to obtain it. In other words, it facilitates declarative programming.

Benefits of lazy evaluation

1. **Efficient memory use and reduced computation.** By evaluating expressions only when their values are needed, Haskell can be more efficient with memory usage and avoid unnecessary computations. This allows for more efficient data handling, especially with large or infinite data structures.
2. **Ability to handle infinite data structures.** One of the most striking advantages of lazy evaluation is the capacity to work with infinite data structures. For instance, Haskell can effortlessly handle lists that, in theory, never end because it only computes the elements as they are required.
3. **Increased modularity.** Laziness enhances modularity – the ability to separate a program into distinct, interchangeable components. Developers can write more general-purpose functions and compose them in various ways without worrying about performance overheads typical of such abstraction in eager languages.
4. **On-demand computation.** Lazy evaluation fits naturally with scenarios where not all the data might be needed. For example, if you're processing a large dataset to find just one item or a specific pattern, Haskell will stop processing as soon as it finds what it's looking for, rather than processing the entire dataset.
5. **Refinement of performance.** While lazy evaluation may sometimes introduce inefficiencies due to the overhead of managing thunks, it can also enhance performance when not all computations results are needed. Developers can write clear and natural code, while Haskell's lazy nature often optimizes performance behind the scenes.

1.5.4. Concurrency

Concurrency is a critical aspect of modern software development, enabling programs to handle multiple tasks simultaneously, thereby improving performance and responsiveness. With its unique features, Haskell offers a particularly robust environment for building concurrent programs.

Haskell's concurrency model

Haskell's concurrency model is built on the concept of lightweight threads, which are managed by the Haskell runtime system rather than the underlying operating system. This model allows for the creation of a large number of threads with minimal overhead, making concurrent programming in Haskell both efficient and scalable.

Additionally, Haskell's concurrency is greatly enhanced by its support for software transactional memory (STM). This mechanism simplifies handling shared mutable states across multiple threads by managing transactions on memory atomically in a way similar to database transactions. This helps to avoid deadlocks, race conditions, and other common concurrency issues while facilitating composability and modularity.

Benefits of Haskell's approach to concurrency

- 1. Simplicity and safety.** Haskell's pure functional nature significantly reduces the complexity associated with concurrent programming. Since most data in Haskell is immutable, many common concurrency problems, such as race conditions and deadlocks, are naturally avoided. This makes concurrent Haskell programs easier to write, understand, and maintain.
- 2. Efficiency at scale.** The lightweight nature of Haskell threads allows programs to scale efficiently with the number of processor cores. This is particularly beneficial in environments where high performance with

parallel processing is required.

3. **Software transactional memory (STM).** STM in Haskell abstracts the complexity of mutexes and locks typically required in other languages. It allows developers to write code that modifies shared memory in a transactional manner, automatically handling conflicts and retries, significantly simplifying concurrent algorithms' design.
4. **Composability.** Concurrency primitives in Haskell are highly composable, meaning they can be combined in various ways to achieve complex concurrent behavior. This composability stems from Haskell's modular nature and powerful type system, ensuring that components interact in well-defined ways.

1.5.5. Metaprogramming

Metaprogramming, the practice of writing programs that write or manipulate other programs, is a powerful technique that can significantly extend the capabilities and efficiency of software development. With its advanced type system and functional purity, Haskell offers a rich environment for metaprogramming.

Metaprogramming primarily revolves around two powerful features: template Haskell and type-level programming. Each serves distinct purposes and offers unique advantages.

1. **Template Haskell.** This is Haskell's facility for compile-time metaprogramming. With template Haskell, programmers can write code that generates other Haskell code during compilation. It provides the ability to perform complex compile-time computations, manipulate Haskell abstract syntax trees (ASTs), and automatically generate boilerplate code.
2. **Type-level programming.** Haskell allows for computations and logic to be embedded within types, leveraging its powerful type system. Type-level programming in Haskell can involve creating and using kinds like

type-level natural numbers or lists, and even performing type-level computations. This capability is enhanced by extensions such as **DataKinds** and **TypeFamilies**, which allow types to carry more complex structures and behaviors.

Benefits of metaprogramming in Haskell

1. **Code generation.** Template Haskell allows for the automatic generation of code, which can significantly reduce the amount of manual coding required and help avoid repetitive boilerplate. This is particularly useful in large projects where consistency and reduction of manual overhead are critical.
2. **Sophisticated abstractions.** Type-level programming enables Haskell programmers to define and use abstractions that are checked at compile time, leading to safer and more robust applications. These abstractions can encapsulate complex behaviors or constraints that the compiler verifies.
3. **Richer type system.** Metaprogramming, especially through type-level programming, enriches Haskell's type system by enabling the expression of more nuanced and powerful type constraints and behaviors. This leads to more expressive and precise type signatures that enhance code safety and clarity.
4. **Improved performance.** Metaprogramming often shifts some computational work to the compile phase, enhancing runtime performance. By reducing runtime checks and computations, the resulting program can run more efficiently.
5. **Dynamic behavior with static guarantee.** Metaprogramming in Haskell allows for a mix of dynamic-like behavior (eg generating different types of functions and structures based on external inputs) while still retaining the guarantees of a static type system. This best-of-both-worlds approach offers flexibility without sacrificing the benefits of static typing.

Chapter 2. Cryptography

Cryptography is a fundamental building block of blockchain systems, distributed ledger technologies (DLT), smart contracts, and cryptocurrencies. This chapter offers an overview of cryptography and its role in blockchain, focusing on hash functions, encryption techniques, and digital signatures, along with their contributions to securing these systems. While technical details are beyond the scope of this book, resources for further exploration are provided at the end of the chapter.

2.1. Hashing functions

When digital systems communicate, messages sent between them may be altered during transmission due to issues with the physical communication medium. Early in computing history, the need for a solution to this problem became evident. An initial solution involved adding an extra 'parity bit' to data sent between devices, serving as a safety guarantee to determine whether data changed between a sender and receiver. A computer bit can either be 0 or 1. For a naive implementation of a parity bit, if the total number of bits set to 1 in a data packet being sent is odd, then the parity bit is set to 1; if even, it is set to 0. However, since the parity bit can only store a single 0 or 1, there is a high chance that errors could occur undetected. For example, if two bits that were originally 0s were altered to 1s, the naive parity bit would still appear valid. Although better parity schemes exist, they remain insufficient for the large amounts of data being transmitted.

Building on this concept, 'checksums' were introduced. Unlike parity bits, which encode sanity check information into a single bit, checksums use

larger data sizes for encoding such checks, ranging from single to multiple bytes.

In fact, sanity checks are not only essential for ensuring data integrity during transmission between computers but also for verifying that data has not been altered when stored on the same device. While checksums are effective for providing these sanity checks, there remains a risk of undetected errors. This risk arises because two different pieces of data can yield the same checksum value.

A more effective sanity check was needed, which is where hashes came into play. Hashes serve as sanity checks similar to checksums but are designed to make it (nearly) impossible for an original piece of data and its altered version to produce the same hash value. They can be thought of as unique digital fingerprints that identify specific data. Given any arbitrary input, a hash algorithm will output a unique fingerprint, or hash. The same input will consistently yield the same hash output.

To reiterate the point, the algorithm will produce a unique fingerprint in a manner where it is (nearly) impossible for two different inputs of data to generate the same hash. If even a single bit of the input data changes, the outputted hash will change drastically. By storing or sending the fingerprint with the original data, it is possible to verify whether the original data has been altered; if it has been altered, then the fingerprint will not match. Additionally, the algorithms used to generate hashes must ensure that it is (nearly) impossible for an altered piece of data and its associated fingerprint (hash) to result in a match. In summary, hash functions provide a solution that guarantees a piece of data has not changed over time and space.

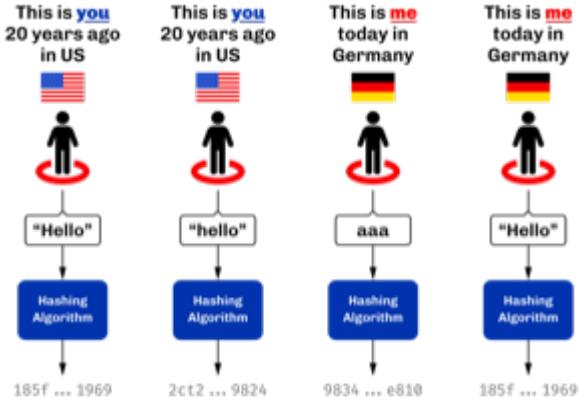


Figure 2. A depiction of different inputs fed into a hashing algorithm and outputs from the hashing algorithm over varying space (exact locations) and time (over 10 years).

[Figure 2](#) depicts different inputs and outputs for a hashing algorithm (specifically the SHA-256 hashing algorithm). In the first leftmost depiction, the text "Hello" is passed into the hashing algorithm that generates a hash (i.e. a unique fingerprint of the text "Hello") that starts with the characters "185f" and ends with the characters "1969". You can test generating hashes using online tools:

- Such as by using either of these online tools: emn178.github.io/online-tools/sha256.html or codebeautify.org/sha256-hash-generator
- In fact, if you type in the text: Hello
- You should see that you end up with the following output hash:
185f8db32271fe25f561a6fc938b2e264306ec304eda518007d17648263819
69
- Both the tools should output the exact same hash for the exact same input text. In fact, any tool that implements the "SHA-256" hashing algorithm should output the exact same hash for the exact same input.

Looking back at [Figure 2](#), comparing the hashes from the first and second depictions (from the left), it can be seen that even the smallest change in text leads to a drastically different generated hash—try it out for yourself in the hashing tools (provided above). Yet if the exact same text is inputted into a hashing algorithm, even if over larger periods of time and in different locations, the exact same hash will be generated for the same input. For example if you input the text "Hello" 20 years ago in the USA

(the leftmost depiction in [Figure 2](#)), and I also input "Hello" now (20 years later), then the exact same hash will be generated.

To summarise, hash functions can guarantee that data remains unchanged by storing the generated hash of the respective data. As long as the hash remains unchanged, it can be verified that the respective data has not been modified. However, if someone alters the data and adjusts the hash to match the new data, the receiver would be unable to detect this change. We will discuss a solution to this issue later in this chapter.

While hash functions solve a significant problem for communicating devices, they do not ensure the privacy of the data being transmitted. We will now explore encryption techniques designed to provide privacy for data communicated between different parties.

2.2. Encryption techniques

Though blockchain platforms do not primarily utilize encryption techniques to guarantee data privacy in the traditional sense, they do employ these techniques to provide other essential guarantees, which we will discuss later in this chapter. We now provide an overview of encryption techniques to help readers appreciate the content presented later.

Encryption techniques were originally proposed to enable secure communication between parties, preventing eavesdroppers from deciphering or understanding the exchanged information. These techniques are classified into **symmetric** and **asymmetric** encryption.

2.2.1. Symmetric encryption

Encryption uses keys to both encrypt (secure) and decrypt (unlock) data. *Symmetric encryption* employs the same key for both processes. Similar to a physical lock, the same key is used for locking and unlocking, specifically during encryption and decryption. This digital key is shared among the

communicating parties, which typically requires prior sharing through a different communication medium.

If an eavesdropper listens to the initial communication where the key is shared, they can decrypt future messages and inject encrypted messages without either party being aware. This occurs because the eavesdropper would have access to the encryption/decryption key. Therefore, it is crucial to share such keys securely, ensuring that an eavesdropper cannot intercept them.

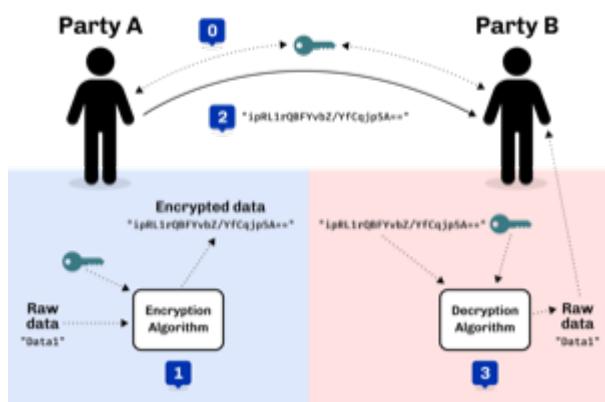


Figure 3. An overview of typical communication using symmetric encryption.

Figure 3 depicts how symmetric encryption is typically used between two parties, where *Party A* intends to send encrypted data to *Party B*. The following notation is used in the diagram:

- the different steps in the process are denoted by numbers in circles
- the solid line in step 2 represents communication between the parties
- the dashed lines at the top indicate some form of prior communication to agree on the shared key
- the dotted lines represent actions that the parties (or their computers) undertake themselves.

The steps involved in the process are as follows:

- Step 0 typically occurs before the parties communicate over the medium used to transmit encrypted data. The parties must agree on a shared encryption/decryption key, which will be used for both

encryption and decryption. This key could be generated by *Party A*, *Party B*, or even generated together—it does not matter as long as both parties have access to the same shared key. It is important to note that any party possessing the specific key can both decrypt and encrypt messages.

- Once the sending party (in this case, *Party A*) has the key, they can encrypt the data intended for the other party by applying the symmetric encryption algorithm using the key, as depicted in step 1. The algorithm will output the encrypted data.
- After that, *Party A*, the sending party, will transmit the encrypted data generated in step 1 to the intended recipient, *Party B*. Eavesdroppers will not be able to decrypt the messages if the key is known only to parties A and B, even if the encrypted data message is overheard by an eavesdropper ^[1].
- Upon receiving the encrypted data, *Party B*, the receiving party, can decrypt it with the key using the decryption algorithm.

The same key can be used by both *Party A* and *Party B* to encrypt and decrypt data, allowing them to securely send and receive messages to each other. This security relies on the key remaining confidential and not being leaked to any third party.

Symmetric encryption is straightforward to understand and implement, as it relies on a single encryption/decryption key. However, it does not offer guarantees regarding:

- **Provenance of messages:** any party with access to the shared key can encrypt data, making it impossible to determine the sender of the encrypted message.
- **Confidentiality of communication:** there is no assurance that messages intended for a specific party will be viewed exclusively by that party.

2.2.2. Asymmetric encryption

The introduction of *asymmetric encryption* in the 1970s provided a more secure solution to mitigate potential eavesdroppers. It ensures that messages can only be decrypted by the intended recipient. In asymmetric encryption, each communicating party has two keys:

- A *public key* associated with the recipient, which is made publicly available. Any party wishing to encrypt data intended for the recipient will use this key to encrypt the data.
- A *private key* that the recipient keeps confidential. This private key is used to decrypt messages sent to them that have been encrypted with their public key.

The public and private keys are intimately linked (hence the term 'key pair'), and it is impossible ^[2] to determine the private key from the public key.

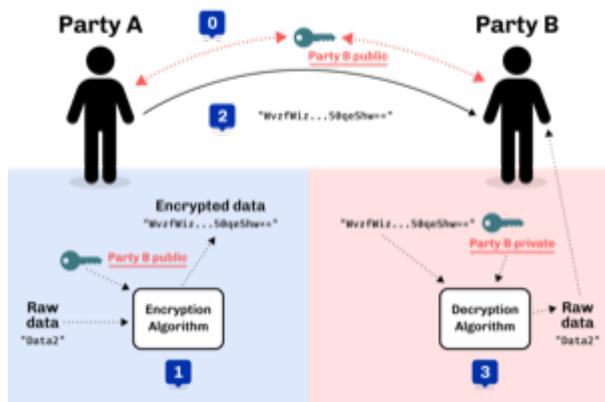


Figure 4. An overview of typical communication using asymmetric encryption.

An overview of how typical communication takes place using asymmetric key encryption is depicted in [Figure 4](#). The main differences in the process are highlighted in red, and a description of the steps involved follows:

- Instead of requiring communicating parties to agree on a shared encryption/decryption key, asymmetric encryption allows parties to disclose their public keys. Parties can make their public keys visible to the entire world. As shown in step 0, *Party B's* public key is made available to *Party A*.

- *Party A* can then encrypt messages intended for *Party B* by inputting the raw data (in this case, 'Data2') along with *Party B*'s public key into the asymmetric encryption algorithm (depicted in step 1). The encryption algorithm will produce the encrypted data.
- After that, *Party A* can send the encrypted data to *Party B* (depicted in step 2), confident that only *Party B* will be able to decrypt the data since it can only be decrypted using *Party B*'s private key, which they keep confidential.
- Finally, *Party B* can input the received encrypted data and their private key into the decryption algorithm (depicted in step 3), which will output the actual message intended for them (which is 'Data2').

Unlike symmetric encryption, which allows *Party B* to send messages back to *Party A* using the same encryption/decryption key, asymmetric encryption does not enable this. This design ensures that messages intended for a specific party can only be decrypted by that party. To reply, *Party B* can follow the same process by using *Party A*'s public key to encrypt messages they wish to send back to *Party A*.

While asymmetric encryption ensures that only the intended recipient can decrypt a particular message, it does not prevent a sender from impersonating someone else. This also applies to symmetric key encryption when the shared encryption/decryption key is compromised. For instance, consider a malicious actor, *Party C*, who has access to *Party B*'s public key. *Party C* could encrypt messages intended for *Party B* and send them, falsely claiming to be *Party A*. *Party B* would have no means to identify that the messages are actually from *Party C*. The solution to this issue is *digital signatures*, which will be discussed next.

2.3. Digital signatures

Digital signatures address the issue of sender impersonation (discussed above) by allowing anyone to verify that a message was created and

'digitally signed' by a specific sending party and that the content remains unchanged. They are established using two fundamental components: *public key encryption* and *hashing algorithms*. The process to create a digital signature is depicted in [Figure 5](#).

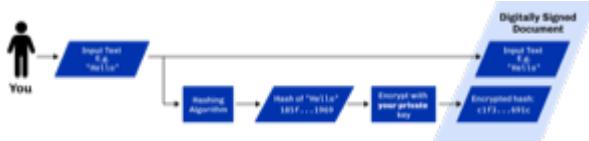


Figure 5. A depiction of how digital signatures are created.

We now walk through the steps involved to create a digital signature as follows:

- The content of the message (e.g. "Hello") is hashed by passing it through a hashing algorithm, creating a unique fingerprint (hash) of the original data (e.g. 185f...1969). This fingerprint uniquely identifies the original content.
- The hash created in the previous step is then encrypted using public key encryption with the sender's private key. The resulting encrypted hash is the digital signature (e.g. c1f3...691c).
- The sender can then transmit the message content along with the generated digital signature. Thereafter, any recipient can verify that the sender created the message and that it has not been altered.

The process to verify a digital signature is depicted in [Figure 6](#).

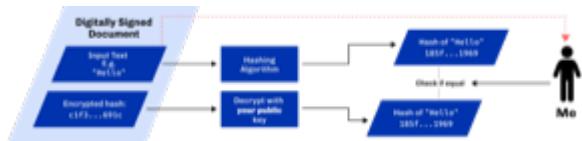


Figure 6. A depiction of how digital signatures are verified.

We now walk through the process to verify a digitally signed message as follows:

- The recipient should first have received the message (e.g. "Hello") and the associated digital signature (e.g. c1f3...691c).

- The recipient then decrypts the digital signature using the sender's public key. The decrypted value should be the hash (unique fingerprint) of the original message (e.g. 185f...1969 depicted in the bottom right of [Figure 6](#)).
- The recipient computes the hash of the received message and compares it to the decrypted hash obtained in the previous step. If the two match, the recipient can be certain that the sender created the message. Specifically, the recipient can guarantee that the sender's private/public key pair was used to generate the digital signature. Since the digital signature is the encrypted hash, and the hash uniquely identifies the specific piece of data, it can be confirmed that the data has not been altered since the digital signature was created and that it originated from the sender (using their public/private key pair).

Having introduced the main cryptographic building blocks used in blockchain systems, this section will now explore how they help maintain and secure these systems.

2.4. Its role in securing the blockchain

In this section we'll now briefly mention where the techniques discussed above are used within the blockchain systems.

2.4.1. Hashing functions

Hashing functions are fundamental to blockchains. Some examples of where hashing functions are used in blockchains are listed below, but indeed other uses exist beyond those listed here.

1. **Unique block identifier.** Blockchains consist of blocks of transactions (and other information). Each block is associated with a unique identifier, which is derived using a hashing function on the block's data.
2. **Unique transaction identifier.** Transactions are typically associated with a transaction hash—which is a unique identifier of the specific

transaction generated by hashing the specific transaction's data. Various examples of this can be seen throughout the book.

3. **Script identifier.** Hashes are often used to identify specific smart contract code/scripts—as will be seen in various parts of the book.
4. **To chain blocks together.** Blocks are immutably chained together by having the most recent block make reference to the previous block's hash. By doing so if any information is changed in the previous block, it would be immediately noticeable since the previous block's hash would no longer match the hash that had since been stored in the most recent block. In fact, it is not just the previous block that this technique provides a solution for, but if any of the history of older blocks is changed, even slightly, then its hash would be invalidated and all blocks that were generated after the respective block and their hashes would also be invalidated.
5. **Address derivation.** Hashing algorithms are often used in blockchains as a means of deriving wallet addresses—e.g. as discussed in Section .
6. **Merkleization** allows for large collections of data items to be compacted into a smaller structure i.e. a merkle tree, and still be able to prove that a particular data item is represented within the merkle tree without having to store the full data items. Merkleization is discussed later in the section.
7. **Content-Addressed Storage.** When other information or files are required to be referenced (for example from a smart contract) it is often useful to refer to the specific file (or information) using Content-Addressed Storage which refers to such information/files using a hash of their content. By doing so, it is immediately possible to determine whether the contents have been changed, since the reference to find the information/file is a hash that should match the computed hash of the content—which can be done at any point. Furthermore, such a referencing scheme allows for deduplication of information, e.g. if the same image data is referred to twice, only one version of the file needs

to be stored since both references will be computed to be the same hash.

2.4.2. Digital Signatures and Encryption Techniques

1. **Wallet Generation.** Public/private key-pairs are generated using asymmetric encryption algorithms which provide the basis for identities/wallets that can be generated on-the-fly in a decentralized manner.
2. **Signing/Proving and Verification of Transactions.** Once a user has a wallet (by generating a public/private key-pair), they can thereafter create and sign transactions that are provable to anyone that the transaction is valid and was really generated using the respective wallet. For example, when someone wants to transfer funds from their wallet to someone else, the transfer's transaction details are digitally signed using the wallet's private key. Thereafter, anyone else can verify that the transaction was really initiated by the respective wallet since they can check the digital signature against the wallet's public key.

2.5. Pointers to cryptography resources

In this chapter we only provided a cursory overview of cryptographic primitives required to appreciate the rest of this book. Some pointers towards resources that will allow for readers to dig deeper into various aspects of cryptography follow:

- Real-World Cryptography, by David Wong: www.manning.com/books/real-world-cryptography
- Cryptography Made Simple, by Nigel Smart: link.springer.com/book/10.1007/978-3-319-21936-3
- Handbook of Applied Cryptography, by Alfred J. Menezes, Paul C. van Oorschot and Scott A. Vanstone: cacr.uwaterloo.ca/hac/

[1] provided that the encryption/decryption key strength is sufficient

[2] or rather computationally infeasible

Chapter 3. Learn About Cardano

3.1. What is Cardano?

Cardano is a **third-generation, open-source, proof-of-stake** blockchain platform designed to provide a secure, scalable, and interoperable infrastructure for developing smart contracts and decentralized applications (DApps). Its primary goal is to address the limitations of earlier blockchain platforms by prioritizing these core tenets. As a fully **decentralized** network, Cardano supports a diverse range of applications, from financial services to supply chain management. Recent advancements, including the [Chang](#) and [Plomin](#) hard forks, have further enhanced its decentralization, establishing it as a leading decentralized blockchain.

Cardano boasts a [thriving ecosystem](#) of builders, developers, content creators, and educators, supported by a wide array of tools and explorers. To learn more about this vibrant community, explore the [ecosystem map](#), the [Essential Cardano website](#), and the [Essential Cardano list](#).

3.2. Who built Cardano?

The Cardano project began in 2015 and is based on **peer-reviewed research** and developed through **evidence-based methods**. It was built by leading scientists, expert developers, and the extended team at [Input | Output \(IO\)](#) (initially IOHK), under the leadership of [Charles Hoskinson](#)). IO is a renowned blockchain infrastructure research and engineering company with a global presence, fostering a focused innovation ecosystem.

The project is pioneered by IO, the [Cardano Foundation](#), [Emurgo](#), and other third parties. The platform is named after [Girolamo Cardano](#), a 16th-century Italian physician and mathematician, while its native cryptocurrency, ada, honors [Ada Lovelace](#)), the 19th-century English mathematician credited with publishing the first computer program.

Cardano's development adheres to an **evidence-based engineering approach**, applying formal logic to the software development cycle. This rigorous process, starting with published research papers and progressing to formal specifications and reference implementations, ensures correctness, safety, and code reliability. Prof. Aggelos Kiayias, chief scientist at IO, states: '*When you go about designing a system, you have to present the system in the context of the research domain in which it belongs, why the system has the benefits it claims to have, and what exactly is the problem that the system is solving. There are a lot of benefits in taking such a first principles approach.*'

3.3. Core principles of Cardano

3.3.1. Mission and open-source ethos

Cardano's core mission, as a fully decentralized platform, is to improve global systems to enable economic inclusion and empowerment for all. Charles Hoskinson))) articulated this in 2020, stating: '*Cardano is an open platform that seeks to provide economic identity to the billions who lack it by providing decentralized applications to manage identity, value, and governance.*'

At its heart, Cardano is a thriving **open-source project** with a healthy and vibrant ecosystem of active developers and builders. This commitment to open-source principles fosters **collaboration and transparency** with the community, leading to a robust and inclusive environment for users and developers. All over two hundred [research papers](#) and technical specifications underpinning Cardano are publicly available, and all

development activity is published online with weekly updates for transparency.

3.3.2. Decentralization and proof-of-stake consensus

Cardano achieves full **decentralization** through over 3,000 community-operated **stake pools**. Over 1.3 million wallets delegate their ada to these pools, representing approximately 64% of the total active ada supply. Network participants validate all blocks and transactions without reliance on a centralized authority, sharing decisions and ownership of information amongst all users. Every ada holder holds a stake in the network, allowing them to delegate their ada to a stake pool to earn rewards and participate in network operations. Stake pool operators can also pledge ada to increase their pool's likelihood of receiving rewards.

Cardano utilizes [Ouroboros](#), a **proof-of-stake (PoS)** consensus protocol. Ouroboros has been rigorously proven to provide the same security guarantees as proof-of-work protocols, while being significantly more **energy-efficient**. You can learn more about Ouroboros in the dedicated chapter.

Cardano extends Bitcoin's UTXO model with an **extended UTXO (EUTXO) accounting model**, which enables smart contracts. For more details on EUTXO, refer to the relevant chapter.

3.3.3. Seamless upgrades

One of the features of Cardano is seamless upgrades. Traditionally, blockchains upgrade using hard forks. When conducting a hard fork, the current protocol would stop operating, new rules and changes would be implemented, and the chain would restart. Cardano handles hard forks differently. Instead of implementing radical changes, the Cardano hard fork combinator technology ensures a smooth transition to a new protocol while saving the history of the previous blocks and not causing any disruptions for end users.

3.4. Core pillars: security, scalability, and interoperability

As a third-generation blockchain, Cardano combines the strengths of earlier generations like Bitcoin and Ethereum, built on the fundamental principles of security, scalability, and interoperability. These three core pillars have defined the design rationale of Cardano, as follows:

- **Security:** Cardano is developed using **formal methods**, which involve mathematical specifications and proofs to guarantee the functional correctness of its core components, providing the highest level of assurance for digital fund management. The Cardano node is primarily written in **Haskell**, a secure functional programming language that promotes building systems with pure functions, leading to isolated and testable components. Haskell's advanced features enable rigorous code correctness checks, including formal and executable specifications, extensive property-based testing, and simulation. The Ouroboros protocol further establishes rigorous security guarantees, backed by numerous peer-reviewed papers presented at top-tier conferences in cybersecurity and cryptography.
- **Scalability:** The Cardano network is designed to **scale with user demand**, processing an increasing number of transactions and enhancing network bandwidth to manage significant supportive data efficiently. Cardano employs techniques like data compression and offers Hydra, a Layer 2 scalability solution. It also uses input endorsers to improve block propagation times and throughput, supporting higher transaction rates.
- **Interoperability:** This is a fundamental design feature, enabling interaction with other systems. A key innovation is **partner chain support**, facilitating asset transfers between parallel blockchains with different rules, mechanisms, languages, or network utilization methods. Work is ongoing to support cross-chain transfers, multiple token types, and commonly used smart contract languages, with the **partner chains**

toolkit marking the initial step.

3.4.1. Sustainability and governance

Governance is central to Cardano's design, ensuring system sustainability and adaptability. A well-developed governance strategy promotes effective, democratic funding for long-term development and organic growth in a decentralized manner. Decentralized governance empowers all users with a voice and control over the protocol's future.

As Charles Hoskinson))) stated: '*Governance is not optional. No matter who you are, what you do, when you create something that other people use, the first question is how do we upgrade, maintain, and change it to meet our needs?*'

Cardano's governance model reflects a **liquid and representative democracy**. Through individual participation and immutable vote recording, ada holders can decide on the distribution of treasury funds and the platform's future development.

Designing a proof-of-stake blockchain necessitates self-sustainability. **Cardano Improvement Proposals (CIPs)** foster and formalize discussions around new features and development within the community. The Cardano governance [roadmap](#), particularly under **CIP-1694**, guides the community through a structured transition to a decentralized, community-driven governance model, allowing for extensive community feedback and refinement.

[CIP-1694](#), named after Voltaire's birth year, was community-written to discuss the future of on-chain governance. It aims to give everyone a voice in Cardano's direction by advancing the current governance system. The proposal outlines a trilateral model consisting of stake pool operators (SPOs), delegate representatives (DReps), and a constitutional committee (CC), each with distinct responsibilities.

Central to the treasury is a **democratized voting mechanism** where ada

holders decide fund allocation by voting on proposals. This ensures democratic decision-making for funding initiatives, protocol updates, and constitutional changes.

Launched in 2023, [Intersect](#) is a member-based organization for the Cardano ecosystem, placing the community at the core of Cardano's future development.

3.4.2. Advantages of Cardano

Cardano's unique strengths stem from its foundational design principles:

- **Academic research & formal methods:** Cardano is built using formal methods, including mathematical specifications, property-based tests, and proofs, which ensure high assurance for software systems and security for digital funds. All underlying research and technical specifications are publicly available, and development activity is transparently published online.
- **System design (Haskell):** The Cardano node is primarily written in **Haskell**, a secure functional programming language that encourages building a system using pure functions, which leads to a design where components are conveniently testable in isolation. Advanced features of Haskell enable employing a whole range of powerful methods for ensuring code correctness, such as basing the implementation on formal and executable specifications, extensive property-based testing, and running tests in simulation.
- **Security (Ouroboros):** **Ouroboros** (the Cardano proof-of-stake protocol) establishes rigorous **security guarantees**; it was delivered with several peer-reviewed papers presented in top-tier conferences and publications in the area of cybersecurity and cryptography.
- **Energy efficiency:** As a proof-of-stake blockchain, Cardano is significantly more energy-efficient and requires less computational power than proof-of-work systems, like Bitcoin, which consume

substantial electricity.

- **Seamless upgrades:** Cardano's **hard fork combinator technology** enables smooth protocol transitions, preserving historical data and preventing disruptions for end-users.
- **Decentralization:** Maintained by over 3,000 community-operated stake pools, Cardano is fully decentralized, with all blocks and transactions validated by network participants without central authority.
- **Functional environment for business use cases:** Cardano provides a foundation for global, decentralized finance, supporting a range of DApps with functional and domain-specific smart contracts and multi-asset tokens.

3.5. Cardano community and ecosystem growth

Cardano benefits from a vibrant and thriving ecosystem that promotes active engagement with builders, developers, content creators, and users. The Cardano ecosystem is a dynamic and rapidly-growing collection of projects, organizations, creators, and builders who are working together to improve and develop the platform even further.

As a community-driven ecosystem, there is a strong focus on innovation, collaboration, and cooperation between innovators, smart contract developers, content creators, and distributed application (DApp) developers that build on Cardano. The aim is to grow the contributor ecosystem even more each year.

Key resources for the Cardano ecosystem include:

- [Cardano Cube Interactive Map](#): Explore the diverse landscape of current projects.
- [Essential Cardano Website](#): A central community resource for understanding Cardano, its partners, mission, roadmap, and building on the platform. This evolved from the original Essential Cardano List

repository created in 2021.

- [Builder Tools and Community Channels](#): Various resources are available to navigate the ecosystem.

3.5.1. Cardano improvement proposals (CIPs)

The [Cardano Improvement Proposal](#) (CIP) process is a structured, community-led mechanism for suggesting and implementing changes and improvements. It ensures transparency and collaboration, allowing the community to shape Cardano's future. Anyone can submit a CIP, covering technical or non-technical suggestions. After adhering to guidelines and review by CIP editors, proposals are opened for community discussion, refinement, and eventual implementation on-chain. All CIPs are documented in the CIP repository, forming an audit trail of historical changes.

3.5.2. Cardano Ambassador program

The [Cardano Ambassador program](#) aims to increase awareness and adoption, fostering relationships and expanding the community. Ambassadors, from diverse backgrounds globally, work diligently as content creators, translators, moderators, and educators to strengthen relationships and educate new members.

3.6. Educating the world about Cardano

Education is a gateway for adoption and has always been an integral part of the strategy of Cardano's pioneering members: [Input | Output](#), the [Cardano Foundation](#), and [Emurgo](#). It plays a transformative role in fostering a knowledgeable and engaged global community by equipping them with the expertise, skills, confidence, and opportunities to deepen their understanding and successfully build on Cardano and thrive in the ecosystem.

Education is a gift that empowers and enhances Cardano community

members through access to knowledge and experience so that they can overcome the complexity of blockchain technology. Pioneering worldwide education on blockchain offers the opportunity to shape the field for generations and leave a lasting legacy.

Cardano's educational offerings cater to developers, academics, and business professionals, equipping them with necessary skills and knowledge.

3.6.1. Input | Output education

The IO education team possesses extensive experience in curriculum design, project management, blockchain technology, Haskell, Cardano expertise, and smart contract languages like Plutus, Aiken, and Marlowe. This ensures comprehensive and practical programs for diverse learners. The team aims to enhance understanding of Cardano technologies for various audiences, including enterprise decision-makers, and to foster a supportive learning environment. IO is committed to improving developer experience and smart contract adoption through education.

As Dr. Lars Brünjes, Director of Education at IO, emphasizes:

'Education is a cornerstone of our approach at Cardano. By equipping individuals with the knowledge and skills to navigate and innovate within the blockchain ecosystem, we empower them to build a more decentralized and inclusive future. My greatest fulfillment came from teaching the all-female Haskell course in Ethiopia, witnessing firsthand the transformative power of education. That experience reaffirmed my belief in the potential of education to create substantive, lasting change.'

IO offers various education streams:

image::mc_education_pillars.png

3.6.1.1. Mission-based education

This education stream aligns with the mission to provide free education to

the Cardano community, including:

- Haskell Course: Aimed at those looking to master the functional programming language Haskell, which is integral to Cardano's development.
- Cardano Days: Interactive events that provide a deep dive into the Cardano platform, covering its unique features and applications.
- Blockchain Workshops: In-person or virtual workshops and lectures on the fundamentals of blockchain and Cardano.
- Essential Cardano: The [Essential Cardano website](#) was launched in 2022 and has since become a thriving community resource. It serves as a resource for understanding Cardano, identifying its partners, learning about its mission and roadmap, and getting started with building on Cardano. This was preceded by the original [Essential Cardano List repository](#) which was created in 2021 as a central canonical guide to the Cardano ecosystem.

As part of this stream, IO offers comprehensive in-person courses in Haskell that run for 10-12 weeks (depending on the curriculum). [Dr. Lars Brünjes](#) and his team have delivered several of these Haskell courses, including:

- Haskell and cryptocurrency course 2017, which ran for eight weeks at the [National Technical University of Athens](#).
- [Haskell and cryptocurrency course 2018](#), which ran for eight weeks at the University of West Indies in Barbados.
- [Haskell course 2019 Ethiopia](#): this three-month course was delivered in Addis Ababa, Ethiopia, in conjunction with the Ethiopian Ministry of Innovation and Technology. It was delivered to an all-female audience of Ethiopian and Ugandan students.
- Online Haskell course 2020: This course was originally planned for Mongolia, but due to COVID-19 it was migrated to an online course.

- Haskell course 2023: Delivered with additional Marlowe and Plutus components. This comprehensive blended learning course was taught in conjunction with [the African Blockchain Center](#) and taught at their offices in Nairobi, Kenya. The team adopted a train-the-trainer approach for this course and produced a train-the-trainer kit for professors in the group. The course blended in-person and virtual interactions, allowing us to connect with attendees and understand their perspectives on the topics covered during the session

3.6.1.2. Cardano Days events

Cardano Days events were launched in 2023, and the team has held several of these very successful events around the globe at various universities, including:

- [ITESO University](#), Guadalajara, Mexico
- [University of Celaya](#), Guanajuato, Mexico
- [University of Malta](#), Valletta Campus, Malta
- [University of Wyoming](#), USA
- [University of Cantabria](#), Santander, Spain
- [Florida International University](#), Miami, USA
- [National Technical University of Athens](#), Greece
- [Autonomous University of Tlaxcala \(UATx\)](#), Tlaxcala, Mexico
- [National Polytechnic Institute \(IPN\) – Puebla Campus](#), Puebla, Mexico
- [Meritorious Autonomous University of Puebla \(BUAP\)](#), Puebla, Mexico
- [Technological University of Tecamachalco \(UTTECAM\)](#), Puebla, Mexico
- [Bilingual and Sustainable Technological University of Puebla \(UTBIS Puebla\)](#), Puebla, Mexico
- [Higher Technological Institute of Teziutlan \(TecNM Teziutlan\)](#), Puebla, Mexico

- [Tokyo Institute of Technology](#), Japan

These two-day events cover the basics of blockchain technology, Cardano, and smart contracts and proved very popular, with an NPS score of 92. More of these events are planned, so if you would like to know more about hosting this event, please get in touch by emailing education@iohk.io.

3.6.1.3. Developer education

The Cardano education program (CEP) for developers consists of a set of courses that cover all aspects of Cardano. This program includes a set of courses and flexible modules that can be tailored to the needs of each audience.

- Cardano Developer course: a blended learning course that teaches Haskell and smart contract development to aspiring blockchain developers.
- [Haskell Bootcamp](#): an immersive self-paced Haskell course. This course provides a stepping stone for people to upskill on Haskell before enrolling in the Plutus Pioneer program. It consists of videos and interactive lessons and has received very positive feedback and engagement.
- [Plutus Pioneer program](#): focuses on Plutus, Cardano's smart contract platform, offering hands-on experience in writing and deploying smart contracts.
- [DRep Pioneer program](#): prepares participants to become decentralized representatives, playing a crucial role in Cardano's governance.
- [Marlowe Pioneer program](#): specializes in Marlowe, a domain-specific language for financial contracts on Cardano, and is aimed at both developers and financial professionals.
- Tutorials: technical tutorials that describe features of Cardano and how to work with them.
- Educational videos: introduce technical aspects, new features, hard fork

events, and so on.

- Hackathon support: writing hackathon challenges and attending the event to support the participants.

The first Cardano developer course was delivered online in conjunction with the [African Blockchain Center](#) to participants from the African region and covered the core modules of Haskell fundamentals and smart contract development languages, including Aiken, Plutus, and Marlowe. This course evolved from the original Haskell course and was expanded to include lectures on Aiken. A further iteration of the course was delivered in person at the [Universidad Technológica Nacional](#) in Buenos Aires, Argentina.

We have delivered several Pioneer Programs aimed at developers and new users. These interactive online training courses aim to widen the reach of IO's education resources and have been completed by over nine thousand people. During these programs, participants attend weekly lectures delivered by Lars Brünjes, director of education at IO, who also held weekly follow-up interactive Q&A sessions. Learners are supported by a thriving community in the Discord chat system that encouraged collaboration and problem-solving.

One of the most positive outcomes of these courses was the amazing community participation on Discord. The participants supported each other, created additional learning resources, were quick to report any issues, and tested features – they really were acting as true pioneers. We also saw a wide range of innovative resources and ideas from the courses, including new wallets, training materials, and new projects on Cardano.

A self-paced Cardano Education Program (CEP) is currently being developed where participants will be able to pick and choose the modules they want to complete and work at their own pace, rather than follow a weekly schedule.

3.6.1.4. Collaborations

IO collaborates with esteemed universities and educational institutions worldwide to deliver high-quality education and is partnered with various universities and educational institutions, including:

- [University of Edinburgh](#), where there is a blockchain laboratory run by IOG's chief scientist [Prof Aggelos Kiayias](#) and his research team
- [University of Athens](#)
- [University of West Indies](#)
- [University of Wyoming](#)
- [Carnegie Mellon University](#)
- [European Business University of Luxembourg](#)
- [University of Malta](#)
- [University of Wyoming](#)
- [University of Cantabria](#)

Additionally, IO worked with [Yeovil College](#) in the UK and [Consilium Academy](#) in South Africa on curriculum design for their blockchain programs.

3.6.1.5. How to collaborate

The IO education team hosts interactive and meaningful training workshops and courses in various locations around the world each year, as well as online or blended learning options. If interested in hosting a Cardano Days event or collaborating on a training course, please contact education@iohk.io. Stay tuned for updates on the [IO Academy](#).

3.6.1.6. About the Cardano developer portal

The [Cardano developer portal](#) is an additional learning resource and is part of the Cardano.org domain: a product-and-vendor neutral contact point for technical topics.

Guided by the principle of community involvement, all content is contributed openly and transparently using a GitHub process (branch, pull request, review, merge). This approach allows anyone to submit proposals with new or updated Markdown-formatted content and thus contribute actively and constructively.

Under the hood, [Docusaurus](#) (an open-source project for building, deploying, and maintaining websites) works as a generator of web pages with extensive search functions.

The content of this developer portal is intended to practically demonstrate and exemplify how certain functions and operations can be implemented on Cardano, as well as showcase existing projects. A basic requirement for including projects is that they must be functioning and usable on the mainnet (see the guideline for [adding new projects](#)).

This portal provides resources to [integrate Cardano](#), [build with transaction metadata](#), [explore native tokens](#), [create smart contracts](#), [participate in governance](#), and [operate a stake pool](#).

Contributions from all individuals, including non-developers, are encouraged to foster continuous evolution by the community. Contributing to the portal can boost reputation and visibility, or serve as a good learning experience in the GitHub open-source and knowledge-minded contribution process. Each contribution acts as a valuable addition to your resume, potentially leading to career opportunities within the Cardano ecosystem. Ways to contribute include spreading awareness, creating issues on GitHub or the Cardano Forum, improving the copy by fixing errors or enhancing writing, and creating explanatory graphics. There is a dedicated section that describes [how best to contribute](#).

The [Builder Tools](#) section of the portal encourages the addition of valuable tools that benefit Cardano developers. Guidelines for adding tools include being an actual builder tool, having a stable domain name, and using a GitHub account with a history or presence in the Cardano community.

The documentation in the portal can always be improved, and users are encouraged to contribute by creating and enhancing tutorials. Reviewing pull requests is another way to contribute, requiring technical understanding and prior contributions. The FAQs cover topics such as the pull request review process, becoming a reviewer, getting added to the contributor list, and connecting with the developer community through Discord and the forum.

3.7. Suggested Reading

- [Cardano Docs](#)
- [IO Academy](#)
- [Essential Cardano](#)
- [Cardano Tokenomics](#)
- [Cardano Academy](#)
- [Cardano Developer Portal](#)

Chapter 4. How Cardano Works

This chapter explains how the Cardano platform works and outlines the core components of the system. Cardano has been designed in modules, with linked components that can be used in various ways. These components of the Cardano platform stack work together under the hood to support the construction and use of the live Cardano blockchain. Let's take a closer look at the software running Cardano.

4.1. Cardano node and system layers

4.1.1. Introduction to the Cardano node

The Cardano blockchain is powered by a network of interconnected nodes. A Cardano node is a fundamental, top-level component of this network, serving as a device running specific software that helps maintain and secure the system.

Nodes connect to each other within the networking layer, which is the driving force for delivering information exchange requirements. Cardano nodes maintain connections with peers that have been chosen via a custom peer-selection process. By running a Cardano node, you are participating in and contributing to the network.

The custom peer-selection process in Cardano nodes is designed to enhance the network's decentralization, security, and resilience. Here's a brief overview of how it works:

- **Dynamic peer-to-peer (P2P) networking.** Cardano nodes use a dynamic P2P networking system. This system automates the peer

selection process, allowing nodes to maintain a specific number of active peer connections. If a connection with a peer is lost, the node will automatically select alternative peers and persistently attempt connections until the desired target is reached.

- **Elimination of static configurations.** The dynamic P2P system eliminates the need for static configurations and manual input from stake pool operators (SPOs). This simplifies the process of running relay and block-producing nodes.
- **Enhanced communication.** By automating the peer selection process, dynamic P2P enables enhanced communication between distributed nodes, ensuring better data synchronization and consensus among participants.

These features collectively contribute to the robustness and efficiency of the Cardano network. As a critical piece of the Cardano blockchain, a node performs several essential functions:

- **Validating transactions.** Whenever a transaction, such as sending ada, occurs on the Cardano blockchain, nodes are responsible for verifying and validating it. This process ensures that all transactions are legitimate and prevents double-spending.
- **Maintaining the blockchain.** The blockchain is a distributed ledger that records all transactions on the network. Nodes maintain an up-to-date and accurate copy of this ledger, continuously adding new blocks of transactions, which explains the term "blockchain".
- **Propagating information.** Nodes communicate with each other to share information about transactions and blocks. When a new transaction or block is verified, nodes propagate (or share) this information across the network, ensuring consistency and synchronization among all nodes.

A node in the Cardano blockchain is implemented in the [cardano-node](#) software executable. The cardano-node software is the backbone of the

Cardano blockchain network. It is a crucial piece of software that runs on each node, enabling them to perform their essential functions. To fully understand the role of the cardano-node software and its relationship with Cardano nodes, let's dive into its key aspects and functionalities.

The cardano-node software was originally developed by IO, one of the companies that pioneered Cardano. This software is designed to run on computers (nodes) and allows them to interact with the Cardano blockchain. It is written in the Haskell programming language, known for its robustness and security features.

The relationship between the cardano-node software and Cardano nodes can be compared to the relationship between an operating system and a computer. Just as an operating system enables a computer to perform its functions, the cardano-node software allows nodes to interact with other nodes and perform their roles. There are several types of nodes, each with specific roles and responsibilities:

- **Relay nodes.** These nodes are responsible for maintaining connections with other nodes in the network. They relay (propagate) information such as transactions and blocks between nodes, ensuring the network remains synchronized.
- **Block-producing nodes.** These nodes are responsible for producing new blocks and adding them to the blockchain. They are typically operated by stake pools, which are groups of stakeholders who combine their resources to increase their chances of being selected to produce a block.
- **Edge nodes.** These nodes interact with end-users and applications. They provide interfaces for users to submit transactions and query the blockchain. Edge nodes do not produce blocks but play a crucial role in facilitating user interactions with the blockchain.
- **Stake pool nodes.** These are specialized block-producing nodes that are part of a stake pool. They validate transactions, produce blocks, and

maintain the combined stake of various stakeholders in a single entity.

Each of these node types plays a vital role in ensuring the security, stability, and efficiency of the Cardano blockchain.

Key features of the cardano-node software

The cardano-node software undergoes constant development and updates, so the key features presented here provide an overview of its capabilities. For the latest information, please refer to the [Cardano developers' website](#).

- **Blockchain synchronization.** Blockchain synchronization in Cardano uses the cardano-node software and involves a series of protocols and processes to ensure that all nodes maintain an up-to-date copy of the blockchain. Synchronization relies on mini-protocols like chain-sync, block-fetch, and tx-submission for data exchange. Nodes communicate using node-to-node and node-to-client Inter-Process Communication (IPC) protocols, facilitating the exchange of blocks, transactions, and ledger state queries. This continuous synchronization ensures that the local database remains current with the latest blockchain data, maintaining the integrity and consistency of the Cardano network.
- **Transaction validation.** One of the primary roles of cardano-node is to validate transactions. When a user initiates a transaction, the node software checks the transaction's validity. This involves verifying that the sender has enough ada to cover the transaction and ensuring that the transaction follows the network's rules.
- **Block production and validation.** The cardano-node software creates new blocks for nodes participating in block production (such as stake pool nodes). It compiles validated transactions into blocks and adds them to the blockchain. Additionally, it validates new blocks produced by other nodes to ensure they are legitimate.
- **Network communication.** The cardano-node facilitates communication between nodes. It propagates information about new

transactions and blocks across the network, ensuring all nodes remain synchronized. This P2P communication is vital for maintaining the network's decentralized nature.

- **Consensus participation.** Cardano uses a proof-of-stake consensus mechanism called Ouroboros. The cardano-node software enables nodes to participate in this consensus process. Stake pool nodes, in particular, play a significant role in producing new blocks and validating transactions based on their stake in the network.
- **Monitoring and reporting.** The cardano-node software provides various monitoring and reporting tools that allow node operators to track the performance and health of their nodes. This includes metrics on block production, network connectivity, and resource usage.

If you are interested in installing and running a node, please refer to the node documentation on the [Cardano developers' website](#). All node configuration files can be found on the [Cardano operations book](#) website.

Cardano node design principles

In proof-of-stake systems, the costs for an attacker producing data and for a defender verifying this data are more balanced than in proof-of-work systems, where the attacker's costs for producing data are significantly higher than the defender's costs for verification.

This motivated many design principles of the Cardano node, such as:

- Designing for worst-case complexity in adversarial conditions
- Relying solely on local or trusted information
- Minimizing the difference in memory use between typical and worst-case scenarios
- Ensuring graceful degradation under excess load.

The Cardano node was designed to mitigate potential attacks that could overload a node, force it to accept bad data, or block its interactions with

other nodes. These attacks might target a single node during network synchronization or block production. Additionally, the design aims to mitigate distributed attacks aimed at a significant portion of block-producing nodes. Each version of the Cardano node comes as a single package, usable by both stake pool operators for block production and by users running the node for other purposes, such as network synchronization and transaction management. This approach ensures that security features are consistent across both block-producing and non-producing nodes.

You can find more information on the design principles and decisions in the [Cardano architecture](#) video lecture by Duncan Coutts, Cardano's chief technical architect.

4.1.2. Cardano node layers

The Cardano node performs several functions that can be categorized into three layers:

- network layer
- consensus and storage layer
- settlement and scripting layer.

A Cardano node layer is represented by a set of libraries that target specific functionalities, such as networking, consensus, or settlement. The consensus and storage layer, often referred to as the consensus layer, and the settlement and scripting layer, known as the ledger layer, are two distinct components that do not depend on each other. These layers are integrated through a consensus/ledger integration layer, which configures how the consensus protocol interacts with the ledger rules. On the other hand, the network layer is more closely integrated with the consensus layer, with the boundary between these two being less distinct compared to the clear separation between the consensus and ledger layers. A simple diagram below represents these node layers:

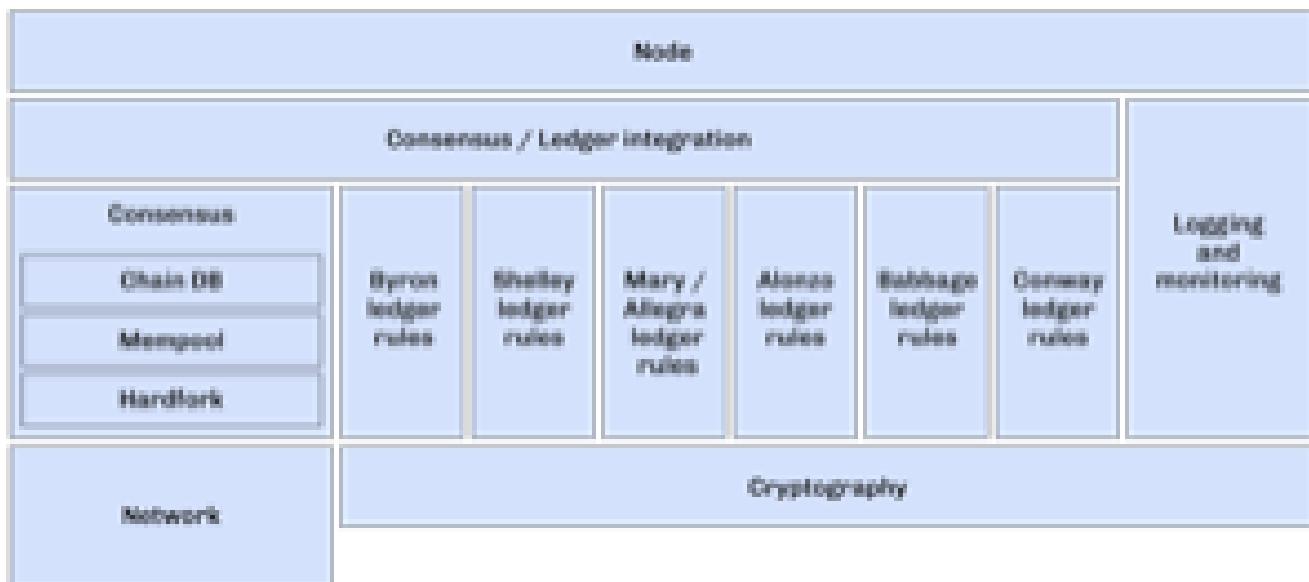


Figure 1. Cardano node layers

The Cardano node also contains an additional layer called the [node shell](#), which manages various functions surrounding the node, namely:

- logging
- monitoring
- configuration
- exception handling
- node startup.

Network layer

The network layer maintains the connections between all the distributed nodes in the Cardano network. It handles communication protocol details and peer selection, obtaining new blocks from the network as they are produced by block-producing nodes and transmitting them between nodes. This layer is a P2P system, with Cardano nodes maintaining connections with peers chosen via a custom peer selection process. Specifically designed for proof-of-stake systems, the network layer includes a framework for writing typed protocols, supporting pipelining, multiplexing, and various protections against adversarial peers.

The [Ouroboros network](#) repository contains specifications of network

protocols and implementations of the network components that run these protocols. These components support a family of Ouroboros consensus protocols. The official [network documentation](#) explains the data flow between and within Cardano nodes and the network constraints, such as congestion control and real-time coordination. It also lists types of mini-protocols that are used to communicate between multiple nodes participating in the Cardano network. You can read more about network protocols in section [Overview of Cardano network protocols](#).

Consensus and storage layer

The consensus and storage layer operates the Ouroboros blockchain consensus protocol. In a blockchain context, consensus ensures that all participants agree on the one true version of the chain. The consensus layer is responsible for making key decisions about the chain, including:

- adopting blocks and determining when to produce new blocks
- choosing between competing chains, if there are any
- selecting slot leaders to produce blocks
- coordinating the interaction between the network and ledger layers.

The consensus layer also maintains all the necessary state to perform these tasks. The Ouroboros consensus algorithm, embedded in the consensus layer, sets block adoption and production rules. To adopt a block, the protocol must validate it against the current state of the ledger. The storage layer provides efficient access to:

- the current ledger state
- recent past ledger states; useful when switching and validating competing chains
- direct access to blocks, facilitating efficient streaming to clients.

Block production occurs within the consensus layer, and to produce blocks, this layer must also maintain a memory pool (mempool) of

transactions to be inserted into those blocks. For more details about block production, read sections [Reaching Consensus using Proof-of-Stake](#), and [Ouroboros Consensus](#).

The problem of **chain selection** arises when two or more nodes extend the chain with different blocks. This can happen when nodes are unaware of each other's blocks due to temporary network delays or partitioning. This situation can also occur under normal conditions depending on the consensus algorithm. When it does, the consensus protocol is responsible for choosing between these competing chains. If the protocol switches to a different chain (a different tine of a fork), it must retain enough history to reconstruct the ledger state on that chain.

An important task of the consensus layer is **selecting slot leaders**. In proof-of-work blockchains, any node can produce a block at any time, provided that they have sufficient hashing power. By contrast, in proof of stake, time is divided into slots, and each slot has a number of designated slot leaders who can produce blocks in that slot. It is the responsibility of the consensus protocol to decide to assign slot leaders to slots. Further explanations can be found in the proof-of-stake and Ouroboros sections.

The consensus layer also **orchestrates** between the network and ledger layers. The network layer primarily transmits blocks and block headers, but does not interpret them. In a few cases, it relies on the consensus layer when making some block-specific decisions. The ledger layer deals only with high-level concerns, meaning it describes how the ledger state is transformed by valid blocks. It only sees a linear history and is unaware of multiple competing chains or the rollbacks required when switching from one chain to another. The consensus layer mediates between these layers and decides which chain is preferable and should be adopted.

The consensus layer was designed not only to perform the above-mentioned tasks, but also to emphasize the **compositionality**, making it usable with many different consensus algorithms and ledgers. This

enables the *hard fork combinator* (*HFC*) technology that allows combining multiple ledgers and regard them as a single blockchain. Because of the hard fork combinator, a new node version designed for the latest ledger era also understands all previous eras. This capability ensures that previous versions of Plutus scripts remain supported by the node even when Plutus is upgraded. The HFC enables smooth protocol upgrades without disruption for users and also preserves the chain history of all operations. You can read more about it in the [hard fork combinator](#) blog.

Generally, the term *hard fork* describes a radical change from one protocol to another. In most blockchains, a hard fork indicates block changes or a change to their interpretation. Traditionally, when a blockchain hard fork happens, the current protocol stops operating, and new rules and changes are implemented, resulting in the chain restart. There is no backward compatibility, and the old version of the protocol can not be used in parallel with the new version for the same blockchain. The term soft fork is used if the protocol changes are compatible with the previous versions.

The HFC technology allows for the combination of two incompatible protocols into one, resulting in a sequential combination of the two protocols. This enables Cardano to integrate blocks from all development phases. Furthermore, the entire network – comprising all connected Cardano nodes – can upgrade gradually, eliminating the need for simultaneous upgrades. Nevertheless, a hard fork still needs to be triggered, and the mechanism to do this will change from the genesis-key-based mechanism to the one defined in [CIP-1694](#). The genesis key mechanism requires that 70% of all SPOs first upgrade to the new node version, and then a consensus of five out of seven genesis keys needs to be reached to trigger the hard fork. The genesis keys were distributed such that three belong to IO, two to Emurgo, and two to the Cardano Foundation. After CIP-1694 will be fully implemented, those genesis keys won't be used anymore. SPOs will still play a crucial role in Cardano upgrades, and will always be free to decide whether they want to upgrade

their nodes to a new version that would signal the acceptance of a proposed hard fork.

In addition to compositionality, other design goals of the consensus layer include:

- support for multiple consensus protocols
- compatibility with multiple ledgers
- decoupling the consensus protocol from the ledger
- enhancing testability
- ensuring adaptability and maintainability
- delivering predictable performance
- providing protection against denial-of-service (DoS) attacks.

You can read more about these goals in the Cardano consensus and storage layer [technical report](#), which targets more experienced developers. This report explains how the goals were achieved, identifies areas for improvement, and presents how the design of this layer can scale to meet future requirements. This [IO blog](#) also elaborates on these goals and provides simple code examples.

Settlement and scripting layers

The settlement and scripting layers form the ledger layer that defines the rules governing blockchain data. These rules govern transaction logic for ada and other Cardano native assets. The ledger layer has a multi-era ledger implementation derived from a set of formal specifications. These formal specifications define the core Cardano components of the ledger layer and the rules for their use. The [Cardano ledger](#) repository lists all eras and provides the formal ledger specification for each. The [Formal ledger specification](#) repository will eventually replace it. Some practical ledger explanations can also be found in the [Cardano ledger docs](#).

The ledger layer is stateless and consists exclusively of pure functions that

define how the ledger is updated with each new block. These functions are derived from the formal ledger rules using the extended UTXO accounting model. The scripting layer handles rules for smart contract logic, such as spending, minting, staking, and certification script logic. Transaction logic not involving smart contracts is managed by the settlement layer, which also provides [simple scripts](#)—a basic smart contract language that enables multi-signature addresses and time locks. An overview of Cardano smart contract languages is covered in section [Smart contract programming languages](#), which presents various Cardano smart contract languages and explains the types in which they can be grouped.

The scripting layer is defined by the Plutus scripting language, sometimes also referred to as Plutus Core or Untyped Plutus Core (UPLC). It provides Turing-complete smart-contract capabilities to Cardano and can be processed by Cardano nodes. Plutus is based on untyped lambda calculus and acts as low-level interpreted assembly code. The compilation pipeline from the Haskell-based Plinth (formerly Plutus TX) smart contract language to Plutus is explained in section [Plutus security](#).

Chapter [Writing smart contracts](#), besides providing an overview of smart contract language options, also showcases code examples, security features, and learning resources for the Plinth and Marlowe smart contract languages.

4.2. The EUTXO model

4.2.1. Accounting models

Every cryptocurrency requires an *accounting model* to track ownership, and Cardano is no exception.

One widely used option is the *account-based* (or *balance-based*) model, where users hold accounts that store their balances, and transactions adjust those balances. This model will likely be familiar to most readers with a bank account, as it mirrors the system banks use and aligns with

how many of us typically understand accounting.

Ethereum is one example of a cryptocurrency that uses the account-based model. For instance, if Alice initially holds 100 ETH and Bob holds 10 ETH, and Alice sends 30 ETH to Bob, their balances would be updated as follows:

Old Balances		New Balances	
Name	Balance	Name	Balance
Alice	100 ETH	Alice	70 ETH
Bob	10 ETH	Bob	40 ETH

Alice sends 30 ETH to Bob

Figure 2. Alice sends 30 ETH to Bob in the account-based model

4.2.2. The UTXO) Model

The account-based model is not the only option, nor is it used by Cardano. Instead, Cardano follows Bitcoin's *UTXO model* and extends it to the *extended UTXO model (EUTXO)*.

The term 'UTXO)' is an acronym for *unspent transaction output* and denotes just that: the output of a transaction that has not yet been spent.

Transactions are the fundamental building blocks of all blockchains, regardless of the accounting model they employ. Transactions trigger changes and facilitate actions. However, while the account-based model updates balances, the UTXO) model spends previously unspent outputs from past transactions and generates new unspent outputs.

Transactions in the UTXO) model have one or more *inputs* and one or more *outputs*. Each input spends an existing UTXO) (thereby rendering it 'spent'), while each output creates a new UTXO), which can later be used as input for another transaction.

You can think of unspent transaction outputs as 'coins' or 'banknotes' that can be spent in future transactions. When you receive a payment, you receive a new coin or banknote, which you can use in a future transaction. When you spend money, you hand over coins or banknotes you received in the past.

In a sense, the UTXO) model is finer-grained than the account-based model – it tracks each ‘coin or banknote’ rather than just total balances.

The UTXO) model resembles cash, whereas the account-based model is more like a bank account. When paying with cash, you hand over whole banknotes or coins, which are unspent outputs from past transactions (when someone paid you). UTXOs work the same way – they must be spent in full, just like coins or banknotes. If you don’t have the exact amount, you hand over more and receive change in return.

In the UTXO) model, this change is called a *change output*, where one or more outputs from a transaction typically return to yourself.

However, the analogy with cash is not perfect. While you must spend entire UTXOs, you can create new UTXOs with arbitrary values, provided the total value of the new UTXOs does not exceed the value of the UTXOs being spent. This differs from cash transactions, where you cannot simply create new banknotes or coins.

For example, if you have an input ‘coin’ worth 100 ada, you can create two outputs: one worth 70 ada and another worth 30 ada. You cannot do this with cash, where splitting a banknote into smaller denominations is impossible without receiving change from a third party.

Let’s reconsider the example of Alice and Bob using the UTXO) model: Alice has 100 ada split between two UTXOs – one worth 60 ada and the other worth 40 ada. Bob has a single UTXO) worth 10 ada.

If Alice wants to send 30 ada to Bob, she can create a transaction using one of her UTXOs, for instance, the 40 ada UTXO). The transaction will have two outputs: one output worth 30 ada to Bob and another worth 10 ada as change back to Alice.

Alternatively, Alice could use her 60 ada UTXO) and send 30 ada back to herself. She could also combine both her UTXOs (60 ada and 40 ada) as inputs and create one or more change outputs, such as 70 ada total, or even split it into two outputs like 50 ada and 20 ada, as long as they sum up to 70 ada.

Even with this simple example, Alice has numerous ways to structure her transaction. The key is that the sum of the output values must match the sum of the input values, with the outputs to Bob adding up to 30 ada and the change outputs reflecting the remaining balance.

While this process sounds complex, in practice, users don't need to worry about the details – wallets handle it automatically. They select the appropriate UTXOs as inputs and create the correct transaction outputs. This process of selecting the right inputs is known as *coin selection*.

Coin selection works similarly to choosing the right coins and banknotes from a physical wallet when paying with cash – you don't want too many coins cluttering your wallet, you avoid receiving too many small coins^[1] as change, but you also don't want to use up all your small change. Wallets aim to strike a balance between these competing objectives.

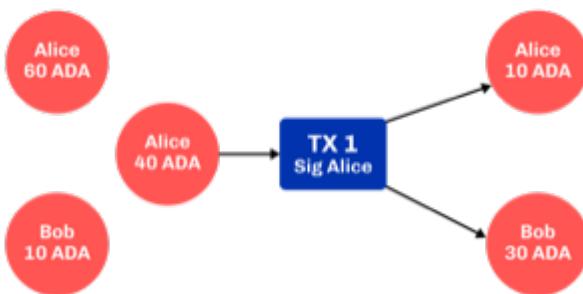


Figure 3. Alice sends 30 ada to Bob in the UTXO model

It is important to note that this example is simplified. In reality, Alice would also need to account for *transaction fees*, which means the change outputs returning to her would be slightly less than the difference between her inputs and the 30 ada sent to Bob.

In the pure UTXO model, outputs consist of an *address* and a *value*. The

address – which, in Cardano’s case, includes a *PubKey Hash* (the hash of a verification key) – specifies who is allowed to spend the output later. To spend this output, transactions must be *signed* using the corresponding signing key, and multiple signing keys can be used.

The *value* represents an arbitrary^[2] combination of ada and native tokens.

Cardano transactions must be *balanced* – roughly speaking, the sum of input values must equal the sum of output values. There are some exceptions and refinements to this simple rule:

- Each transaction needs to include *transaction fees*, which get subtracted from the input values before being compared to the output values
- Cardano native tokens (i.e. other tokens besides ada) can be *minted* (created) or *burnt* (destroyed) in a transaction
- Staking rewards can be withdrawn, getting added to the outputs without having compensating inputs^[3].

4.2.2.1. About change

Change is essential for a transaction to have any real-world effect. Without it, a transaction serves no purpose. In the account-based model, change is in the account balances. For those familiar with imperative programming, account balances in this model are like *global* variables that get updated by transactions. In the UTXO) model, change happens within the set of UTXOs – some UTXOs are spent, and new ones are created. However, no output is ever modified. Once created, an output remains unchanged permanently. The transaction’s effect lies solely in consuming existing outputs and generating new ones. For those familiar with functional programming, UTXOs are comparable to *immutable* data structures, which are *consumed* but never modified by transactions.

In the example above, before Alice sends 30 ada to Bob, the UTXO) set consists of Alice’s two UTXOs (60 ada and 40 ada) and Bob’s UTXO) of 10 ada.

Old UTXO Set

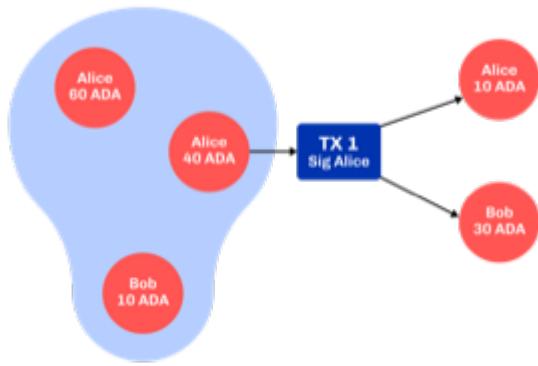


Figure 4. The old UTXO set before Alice's transaction

Alice's transaction consumes her UTXO) worth 40 ada and generates two new UTXOs – one worth 30 ada sent to Bob and another worth 10 ada returned to Alice.

So while no individual output changes, the *set* of unspent outputs changes – one output is removed from the set (because it is now *spent*), and two new ones are added.

New UTXO Set

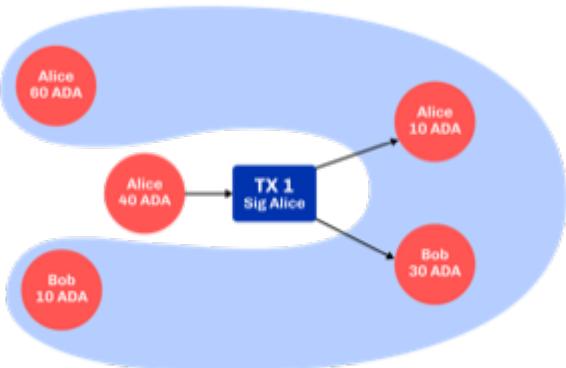


Figure 5. The new UTXO set after Alice's transaction

Of course, there may be many more UTXOs on the blockchain that are not considered here, but they have no impact on the specific transaction examined here.

4.2.3. Extending it: the EUTXO model

The UTXO model is simple and elegant, allowing users to send and receive funds without restriction.

These days, however, users expect more from a blockchain than just the

ability to send and receive funds. They want to create smart contracts that can do more than merely move funds; they seek the ability to implement arbitrary logic and enforce complex rules. They also want to create fungible or non-fungible tokens (NFTs) and trade them on decentralized exchanges or sell them on decentralized marketplaces.

In the UTXO model, a transaction can spend inputs locked at a specific address if it is signed by the corresponding signing key.

In the extended UTXO model, this concept is generalized by replacing the requirement for specific signatures with arbitrary logic.

In addition to using hashed public keys as part of addresses, the EUTXO model introduces addresses that contain hashed *scripts*, written in a programming language (*Plutus Core* in the case of Cardano).

During validation, when a transaction has an input at a script address, the corresponding script (also referred to as a *validator* in this context) is executed. If the script execution completes without error, spending the input is considered valid; otherwise, it is deemed invalid.

To make this idea work, three additional components are needed:

- In the UTXO model, a transaction output consists of an address and a value. The EUTXO model adds a third component – a piece of data called a datum.^[4]
- A transaction attempting to spend an output at a script address must include another piece of data in the input, known as the *redeemer*. The redeemer acts as a ‘key’ to ‘unlock’ an input – a generalization of the signature used to unlock outputs at public key addresses.
- When a Plutus Core script is executed for validation, it receives the datum, redeemer, and *context* as arguments. The context contains the transaction being validated along with all its inputs and outputs, but no other information.

This design strikes a balance between expressiveness and security:

- Bitcoin provides smart contract capabilities through *Bitcoin script*, but these scripts can only access the output being validated and the Bitcoin equivalent of a redeemer, not the entire transaction with all inputs and outputs. As a result, Bitcoin script is highly limited and cannot support the sophisticated smart contracts that users expect from blockchains like Ethereum.
- Ethereum smart contracts are powerful and flexible but also extremely challenging to implement correctly. Their context encompasses the entire state of the blockchain, complicating the prediction of execution outcomes. This has led to several notorious exploits and bugs, resulting in the unexpected loss of millions of ether.

Cardano's EUTXO model, incorporating datums, redeemers, and contexts, is both powerful and flexible enough to match the capabilities of Ethereum while remaining simple enough to enhance predictability regarding transaction outcomes.

Cardano transactions can be validated *locally*, without needing to submit them to the blockchain first, since the context includes only the transaction itself along with its inputs and outputs. Although a transaction may fail upon submission – such as when another transaction has already spent an expected input – if it succeeds, it will yield the predicted result.

As a result, Cardano transactions incur a fee only if they succeed and are included in the blockchain. In contrast, Ethereum transactions can fail yet still cost gas. This scenario is unlikely to occur on Cardano, provided users adhere to the established safety mechanisms.

Transaction determinism is extremely important and deserves further explanation:

On a blockchain like Ethereum, the outcome of a transaction can potentially be influenced by *any* activity occurring on the blockchain. This

makes it impossible to determine the effect of a transaction off-chain before submission.

In contrast, on a blockchain using the EUTXO model, the outcome of a transaction is solely determined by the transaction itself, its inputs and outputs, and nothing else. Therefore, it *is* possible to predict the effect of a transaction off-chain before submission.

As mentioned above, the only aspect that may change on an EUTXO blockchain is the set of UTXOs. However, the outputs themselves remain immutable. A transaction may encounter a situation where its inputs are consumed by other transactions before it is submitted, leading to failure without incurring a fee. Nonetheless, if all inputs remain unspent, the transaction will produce the predicted outcome.

One exception to this rule – related to the handling of time – will be discussed later.

4.2.3.1. Atomic swaps

Let's clarify this with an example – *atomic swaps*.

We have mentioned *native tokens* and NFTs before, and we will explore them in more detail later, but for now, let's discuss a simple example.

Let's say Alice is the owner of an NFT, and she is willing to sell it to Bob for 100 ada.

She doesn't necessarily trust Bob, so she doesn't just want to send her NFT to Bob and hope he will pay her 100 ada later.

Likewise, Bob doesn't trust Alice and doesn't want to send her 100 ada, hoping she will send him the NFT afterward.

This problem can be addressed using the UTXO model, even without smart contracts. For instance, Alice or Bob can create a transaction with two inputs: Alice's NFT and Bob's 100 ada. The outputs would be 100 ada

for Alice and the NFT for Bob. Alice can then partially sign the transaction and send it to Bob off-chain (for example, via email). Bob can subsequently add his signature and submit the transaction to the blockchain.

This approach is secure because neither party can submit the transaction without the other's signature. Additionally, Bob cannot manipulate the transaction to his advantage prior to signing, as doing so would invalidate Alice's signature.

This transaction exemplifies an *atomic swap* – it facilitates the exchange of Alice's NFT and Bob's 100 ada 'atomically,' without requiring trust. Either Alice receives her 100 ada, and Bob receives the NFT, or neither party completes the transaction.

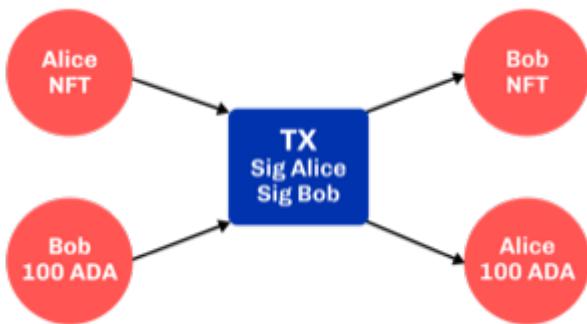


Figure 6. Alice and Bob perform an atomic swap in the UTXO model

This approach has at least two issues: first, the partially signed transaction must be sent off-chain, which is not ideal and can be cumbersome for users. Second, Alice must find Bob and agree on the terms of the swap, which again requires an off-chain process.

Within the EUTXO model, this can be enhanced by employing a script to enforce the swap's terms. Alice can create a transaction that spends her NFT and generates an output locked by a script that requires *someone* (potentially Bob, but Alice does not need to specify) to send 100 ada to Alice to unlock it.

So how does this work?



Figure 7. Alice has an NFT she wants to sell

Alice creates an atomic-swap script, sends her NFT to the corresponding *script address* (given by the hash of the script), and sets the price (100 ada in our example) within the *datum* of the output.



Figure 8. Alice locks her NFT in a script output guarded by the atomic-swap script

To unlock that UTXO and spend it, the script will verify that the spending transaction includes an output of 100 ada directed to Alice. The script can 'see' the entire spending transaction (but nothing beyond that), allowing it to check for an output of 100 ada to Alice.

In practice, the script will likely also enable Alice to reclaim her NFT at any time. Without this functionality, she might encounter difficulties retrieving her NFT if no one expresses interest in purchasing it.

Anyone will be able to spend this UTXO and obtain Alice's NFT, provided they also send 100 ada to Alice.

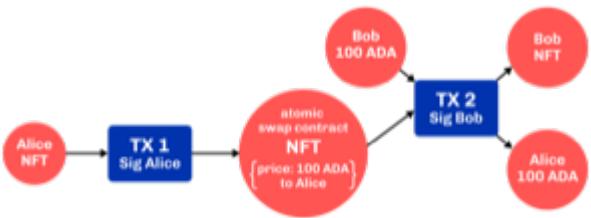


Figure 9. Alice and Bob perform an atomic swap in the EUTXO model

In this example, Alice ceases control of her NFT by sending it to a script address while ensuring that she will receive 100 ada if someone other than herself wishes to spend that UTXO).

Note that in this case, the second transaction only requires a signature from Bob to authorize the spending of the 100 ada that belongs to him. Spending the NFT is permitted by executing the script, rather than requiring anyone to sign the transaction.

Later, we will discuss a potential issue with this smart contract, known as the *double satisfaction* problem, but it can be effectively resolved within the EUTXO model.

4.2.3.2. Validation

So far, we have briefly touched on how Cardano nodes validate transactions.

Validation occurs in two *phases*.

4.2.3.2.1. Phase 1

The first phase consists of ‘cheap’, quick checks. These checks do not incur a fee, even if they fail. They include, but are not limited to, the ‘indeterministic’ aspects of validation – things that cannot be verified before submission.

One such check concerns the availability of inputs: a transaction is only valid if all its inputs remain *unspent*. It is possible for a transaction’s inputs to be consumed between its creation and submission and the time when a node validates it. This means that while the transaction may appear valid upon submission, it can become invalid if a concurrent

transaction spends one of its inputs before it is included in a block.

Another check is the *balance check*: the sum of input values must equal the sum of output values minus transaction fees (ignoring the minting or burning of *native tokens* for simplicity). This check is deterministic and can be performed before submission.

Transactions also include a *validity interval*, specifying a time range within which the transaction is valid. Both ends of this interval can either be unrestricted or tied to specific slots. For a transaction to be valid, the block's slot must fall within this interval, so during validation, the node ensures this condition is met before including the transaction in a block.

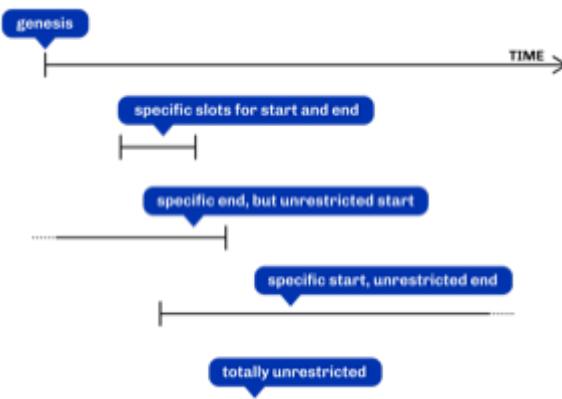


Figure 10. Validity intervals

Each transaction also includes a set of *required signatures*, which nodes verify during the first validation phase to ensure that all the necessary signatures are present.

4.2.3.2.2. Phase 2

The second validation phase is more costly, but it is only performed after all phase 1 checks have passed. This phase can also be conducted off-chain before the transaction is submitted.

In this phase, scripts are executed. Most importantly, if a transaction attempts to spend script inputs, the corresponding scripts are evaluated one by one. If any script fails, the entire transaction is deemed invalid, and validation fails.

As we will discuss later, scripts are also used in other contexts, such as native tokens and staking, and all related scripts are executed during this validation phase.

4.2.3.2.3. Script outputs

To execute scripts during phase 2 validation, nodes must *have* these scripts. Since script addresses are determined by the *hash* of the script rather than the script itself, the transaction must include all relevant scripts, as it is practically impossible to reverse the hashing process and recover the script from its hash.

However, including scripts directly in transactions can lead to duplication on the blockchain and larger transaction sizes, especially when the same script is reused multiple times.

To mitigate this, Cardano introduced *script outputs*. In addition to address, value, and datum, an output can optionally include a script. Transactions can then reference these script outputs instead of including the entire script, as long as a relevant script output already exists.

The decision to create a script output depends on the expected usage of the script:

- If the script is only used once, creating a script output is unnecessary.
- If the script is reused frequently, creating a script output can be beneficial. Although the initial transaction with the script output may be larger and more expensive, future transactions referencing that script will be smaller and cheaper.

4.2.3.2.4. Collateral

It is technically possible to force the submission of a transaction that will fail during phase 2 validation, although there is never a legitimate reason to do so. When this occurs, nodes must perform unnecessary and resource-intensive work.

To discourage this, transactions requiring phase 2 validation, such as those attempting to spend a script input, must include *collateral*. This is an input from a **PubKey** address that holds a minimum amount of ada. If phase 2 validation fails, the collateral is forfeited.

However, in practice, this scenario is unlikely to happen because invalid transactions typically fail earlier, preventing unnecessary validation if users follow the standard processes.

4.2.3.2.5. Determinism and time

As mentioned earlier, *determinism* is a key feature of transactions in the EUTXO model: the outcome of a transaction is determined solely by the transaction itself, its inputs, and its outputs.

However, certain smart contracts must account for *time*. For example, a *vesting contract* aims to release funds only after a specified period.

This raises a question: how can a transaction that depends on time remain deterministic? The success of the unlocking transaction clearly depends on whether the appropriate amount of time has elapsed; if enough time has passed, the transaction succeeds, and if not, it fails.

The solution lies in the concept of the *validity interval*, which is included in every transaction.

Since phase 2 validation occurs only after phase 1 has succeeded, and phase 1 checks the validity interval, a script can safely assume that the transaction's validity interval includes the current time. While the script does not 'know' the exact time, the current time is guaranteed to fall within the specified validity interval.

This ensures that the script's execution remains completely deterministic, even though it takes time into account.

4.2.3.2.6. A vesting example

Consider the example of creating a vesting contract that restricts spending

until after January 1, 2050.

In this case, the script must check that the *start* of the transaction's validity interval is after January 1, 2050.

While the script does not know the exact current time, it does know that the current time falls within the validity interval. Therefore, if the interval starts after January 1, 2050, the current time must also be after that date.

If the validity interval starts before January 1, 2050, the script will fail, since the current time *could* be before that date. Even if the current time is after January 1, 2050, the script cannot verify this with certainty and will therefore reject the transaction.

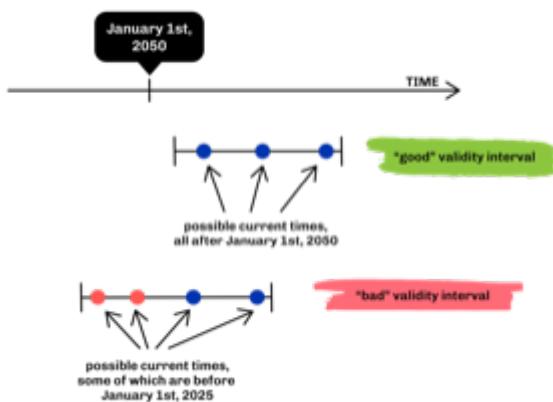


Figure 11. Vesting example

4.2.3.3. Composability

One of the key strengths of the EUTXO model is the *composability* of smart contracts.

In the account-based model, while smart contracts can interact with each other, these interactions can be unpredictable and may lead to unintended outcomes.

In the EUTXO model, each script output is protected by its validator. Although different outputs can be governed by the same script, they can also have separate validators. Each validator independently decides whether the transaction meets the conditions to spend the input it protects without depending on other validators. As a result, well-constructed

scripts can easily be combined in a transaction without concerns about unforeseen interactions.

Let's recall the [atomic swaps](#) example:

Suppose Alice has several NFTs she wants to sell and creates a script output secured by the atomic-swap script for each of them. Bob can then create a transaction that spends all the outputs containing the NFTs he wishes to purchase and generates payment outputs for each of them.

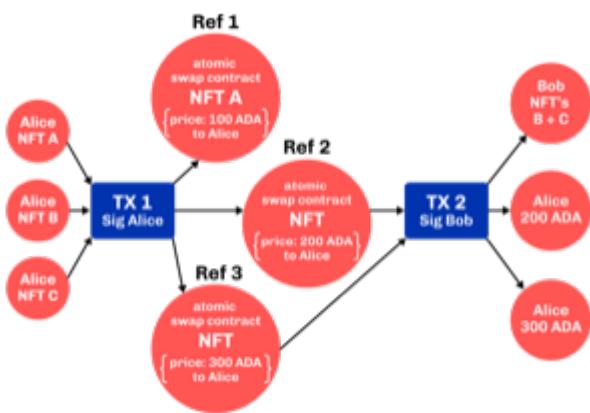


Figure 12. Bob buys two NFTs from Alice in a single transaction

Combining multiple atomic swaps into one transaction does not require explicit implementation in the atomic-swap script; it is a natural consequence of how the EUTXO model works.

4.2.3.3.1. The double satisfaction problem

As previously mentioned, there is a challenge with the atomic-swap script known as the double satisfaction problem.

Let's recall the example of Alice and Bob, this time assuming that Alice changes the price of NFT C from 300 ada to 200 ada.

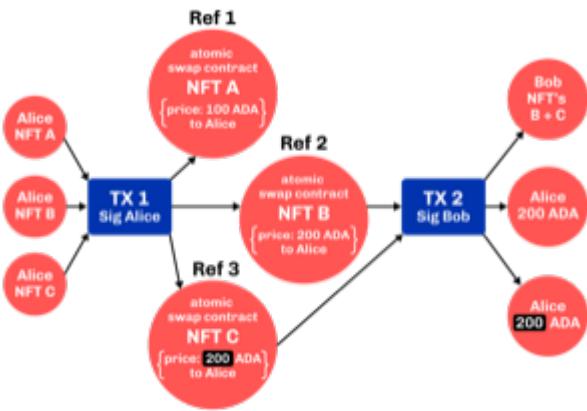


Figure 13. Bob buys two NFTs from Alice, but for different prices

This looks fine, and both Alice and Bob get what they want.

Unfortunately, Bob can instead do the following and cheat Alice out of 200 ada:

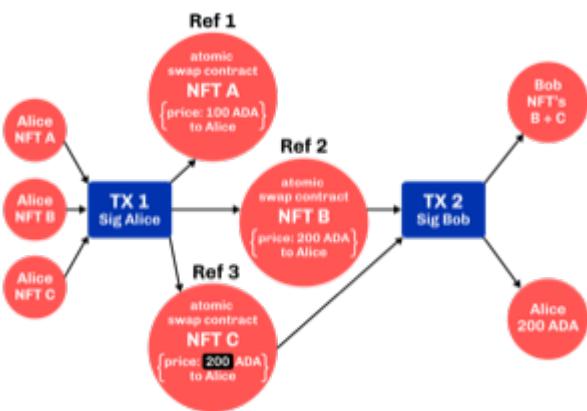


Figure 14. Bob cheats Alice by making one payment for two NFTs

So instead of creating one payment output to Alice for each NFT he buys from her, Bob only creates a single payment output, thus paying 200 ada instead of 400 ada.

His transaction (Tx 2) will validate correctly: as explained above, the scripts guarding the NFTs will be executed sequentially:

- The script for NFT B will check whether the transaction contains a payment output to Alice worth 200 ada, find it, and validate the transaction.
- The script for NFT C will do the same, verifying *the same* payment output to Alice and also validating the transaction.

This issue arises from how validation works during phase 2 – all validator scripts run sequentially and independently, lacking a mechanism to share information between them. Consequently, the first script cannot ‘mark’ the payment output it finds or ‘claim it for itself.’

Note that this situation can also occur in real life. For example, Alice runs a mail order business, and Bob orders one item for 200 USD at the beginning of the month. Later that month, he orders another item for 200 USD.

Alice sends him two invoices, but Bob only pays one. At the end of the month, Alice reviews her accounts and checks whether all invoices have been paid. She examines the first invoice, sees an incoming payment of 200 USD, and marks it as paid. Later, she checks the second invoice, sees the same incoming payment of 200 USD, and marks that invoice as paid as well.

Why does this not cause problems in ‘real life’? Because Alice will likely include an order number or invoice number in her invoice, which Bob must reference in his payment. This way, Alice can identify which payment corresponds to each invoice.

Bob cannot cheat because he must include the invoice number with his payment, but he cannot include both invoices. Therefore, he cannot get away with making only one payment.

Fortunately, we can apply the same principle to fix the atomic-swap contract and secure it against the double satisfaction problem.^[5]

Instead of merely searching for a payment output to Alice with the correct price, the script can look for such an output that also includes the *UTXO reference*^[6] of the NFT output in its datum. Remember that any output can carry a datum, not just script outputs.

UTXO) references are unique on the blockchain; no two different UTXOs can have the same reference. In our example, the output containing NFT B

will have a reference distinct from the one for NFT C. To satisfy validation for spending the output containing NFT B, Bob must include a payment output to Alice that contains the UTXO) reference of the output for NFT B in its datum. The same requirement applies to the output for NFT C. Since these two references differ, Bob can no longer cheat by providing only a single payment output to Alice.

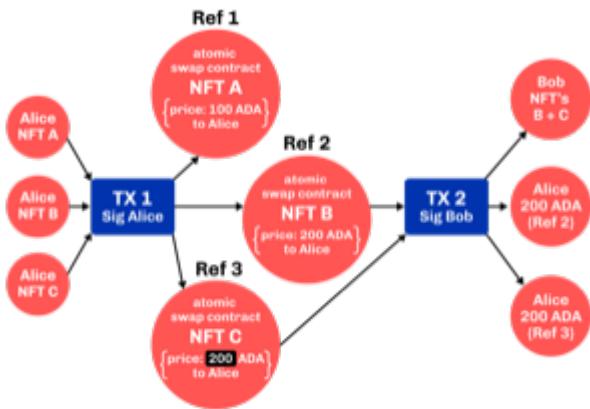


Figure 15. Bob buys two NFTs from Alice with no way of cheating

4.2.3.3.2. Flash loans

The composable nature of smart contracts in the EUTXO model often results in unexpected features ‘for free’ that would need to be explicitly implemented in other models.

As discussed earlier, one example is the ability to combine atomic swaps into a single transaction.

Another example is *flash loans*. A flash loan is a loan taken out and repaid within the same transaction. On Ethereum, the popular decentralized exchange (DEX) Uniswap offers flash loans as a special feature that required explicit implementation in its smart contract code.

Our simple atomic-swap script enables flash loans ‘out of the box’, without the need for explicit implementation.

For instance, suppose Alice wants to sell 100 ada for 45 DJED, and Bob wants to sell 50 DJED for 100 ada. Both Alice and Bob use the simple atomic-swap script to list their offers on the Cardano blockchain.

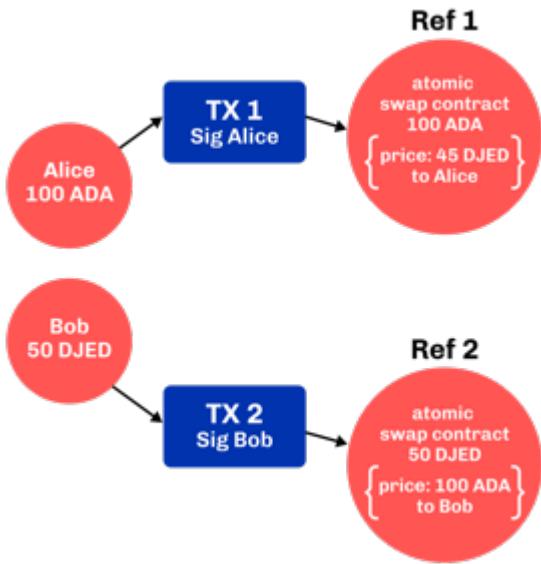


Figure 16. Alice and Bob offer atomic swaps

Charlie notices both offers and realizes he could make a profit by temporarily borrowing 45 DJED:

- Charlie takes out a loan of 45 DJED
- He uses those 45 DJED to buy 100 ada from Alice
- He then uses the 100 ada to buy 50 DJED from Bob
- Finally, he repays the loan and receives a 5 DJED profit.

Charlie can complete this entire process in a single transaction on Cardano without needing to explicitly borrow funds.

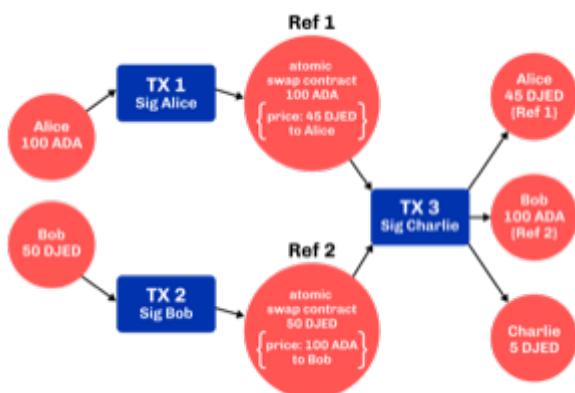


Figure 17. Charlie uses an automatic flash loan to earn 5 DJED

During phase 2 validation, the script guarding Alice's 100 ada checks for a payment output to Alice of 45 DJED with the correct reference, finds it, and approves the transaction.

Similarly, the script guarding Bob's 50 DJED checks for a payment output to Bob of 100 ada with the correct reference and allows the transaction.

The transaction is balanced (100 ada + 50 DJED go in, 100 ada + 45 DJED + 5 DJED go out), so validation succeeds, leaving Alice, Bob, and Charlie satisfied, each receiving what they wanted.

4.2.4. Beyond validation: other uses of smart contracts

One way to understand the transition from the UTXO model to the EUTXO model is by examining addresses. In the UTXO model, whenever a transaction tries to spend a UTXO, it must be signed using the signing key corresponding to the address. The EUTXO model introduces script addresses, meaning that transactions trying to spend outputs from script addresses must be validated by the corresponding script.

The address of a UTXO determines how spending it is validated. By moving from the UTXO model to the EUTXO model, a new method for validating inputs is added – by executing scripts.

On Cardano, however, addresses not only define how UTXOs are *spent* but also determine if and how the UTXO is *staked*.

To better understand this, we need to look closely at the ‘anatomy’ of Cardano Shelley^[7] addresses and their relation to staking.

4.2.4.1. The anatomy of a Cardano address

Every address contains a *spending* or *payment* part, which dictates the conditions under which a UTXO at that address can be spent:

- If the payment part of an address is represented by a *payment public key hash* – the hash of a *payment verification key* – then any transaction attempting to spend a UTXO from that address must be signed by the corresponding *payment signing key*.
- Alternatively, the payment part of an address can be represented by the hash of a *script*, in which case the script is executed when validating a

transaction attempting to spend a UTXO) at that address.

So far in this chapter, we have focused only on this mandatory payment part of an address.

On Cardano, however, every (Shelley) address may also include an optional *staking* part, known as *staking credentials*. Like the payment part, the staking part can either be a *staking public key hash* – the hash of a *staking verification key* – or a *script hash*.

As a result, Cardano supports six different types of addresses:

Payment part	Staking part
Payment PubKey hash	No staking
Script hash	No staking
Payment PubKey hash	Staking PubKey hash
Script hash	Staking PubKey hash
Payment PubKey hash	Script hash
Script hash	Script hash

While the payment part of an address defines the conditions under which a UTXO) at that address can be spent, the staking part determines whether the ada in a UTXO) is *staked* to a stake pool and, if so, to which one. Staked ada earns staking rewards.

Beyond consuming inputs and producing outputs, Cardano transactions can also handle *staking certificates*, *delegation certificates*, and *withdraw rewards*.

Once a staking certificate is registered for specific staking credentials and a delegation certificate is created for a stake pool, staking rewards accumulate and can be withdrawn later.

All staking-related actions must be validated, similar to UTXO) spending:

- A transaction that performs a staking-related action for staking credentials given by a staking public key hash must be signed by the corresponding staking signing key.
- A transaction that performs a staking-related action for staking credentials given by a script hash must be validated by the corresponding script, which is evaluated during phase 2 validation.

4.2.4.2. Cardano native tokens

Finally, scripts play a crucial role in the minting and burning of *Cardano native tokens*.

While ada is the native currency of the Cardano blockchain, Cardano also supports the creation of *custom tokens*, which can be either *fungible* (like ada) or *non-fungible* (like NFTs).

Minting and burning custom tokens must have restrictions, otherwise, they would be pointless if allowed freely. Therefore, these actions are also governed by scripts.

A Cardano native token is identified by two components—the *policy ID* and the *token name*:

- The token name is an arbitrary byte string, up to 32 bytes long
- The policy ID is the hash of a script.

Whenever a transaction mints or burns a token, the corresponding script is evaluated during phase 2 validation.

These *minting scripts* can range from completely permissive (allowing unrestricted minting and burning) to more controlled cases requiring specific signatures or enforcing complex rules, such as ensuring the uniqueness of tokens to create NFTs.

4.3. Reaching Consensus using Proof-of-Stake

This section explores the concept of proof of stake (PoS) and discusses its

importance for blockchain technology. It also compares and contrasts it to other consensus mechanisms.

Introduction to proof of stake PoS is a type of [consensus mechanism](#) that uses the amount of stake (or value) held in the system to determine consensus among protocol participants. Essentially, a consensus protocol determines the rules and parameters governing the behavior of blockchains. Other examples of consensus mechanisms include proof of work (PoW), proof of useful work (PoUW), and proof of burn (PoB). A PoS consensus protocol ensures distributed consensus by determining whether transactions are legitimate. Transactions are the data elements generated within the blockchain, such as cryptocurrency, contracts, or records.

Think of consensus as a ruleset that each network participant adheres to. Since blockchains operate without a central authority, a consensus protocol enables distributed network participants to agree on the history recorded on the blockchain. This agreement allows them to reach a consensus on what has happened and continue from a single source of truth.

Let's exemplify how consensus works with an everyday example. Imagine you and your friends keep a shared notebook to track who owes money after lunch outings. Anyone can make entries, but you need a way to agree on which entries are valid to avoid disagreements.

Here's how you might use a consensus algorithm:

- Proposal. They propose it to the group whenever someone wants to add an entry (like 'Alice owes Bob \$10').
- Validation. Everyone checks the proposed entry to ensure it's correct (eg, Alice agreed to owe Bob \$10).
- Agreement. If most of the group agrees that the entry is valid, it gets added to the notebook.

Similarly, a consensus algorithm ensures that all participants in a network agree on the same data, keeping the system accurate and reliable, even when many different people are involved.

PoS works in a synchronous and partially asynchronous network topography. It ensures that during the generation of these transactions, a temporary authority is required to determine the data used and shared within the collection. This authority's job is to make and broadcast that call to the rest of the network. This is where the PoS protocol operates to determine authority for these transactions during the mining process. Proof refers to proving that the transactions are legitimate, while stake is the value held by the various addresses on individual nodes. The value refers to the amount of individual value elements, or coins, in operation within a slot ledger at any time.

We can illustrate how PoS works with the following example. Imagine a community garden where you and your neighbors decide what to plant each season. Instead of having everyone vote equally, the neighbors decide that the more time and effort someone has invested in the garden, the more their vote should count.

Using PoS to decide what to plant may work as follows:

- Stake. In the community garden, your 'stake' is the time and effort you've put into maintaining the garden. The more hours you've worked or the more plants you've contributed, the more influence you have in deciding what gets planted next.
- Selection. When it's time to decide on new plants, instead of everyone voting, the decision is made by a random selection process that favors those with a larger stake. So, if you've spent a lot of time and effort in the garden, you have a higher chance of being selected to make the decision.
- Responsibility. If you're chosen, you get to decide which plants to add. Because you've invested a lot in the garden, you're motivated to make

good decisions to benefit the whole garden and ensure it thrives.

- Rewards. As a reward for your effort and good decisions, you might get some of the produce or extra benefits from the garden, encouraging you to keep contributing.

In terms of a blockchain network, PoS works as follows:

- Stake. Every ada holder has a stake in the network that they can delegate to a pool from their wallet. The process is safe because no ada leaves the user's wallet.
- Selection. The network randomly selects a pool to validate new transactions and create a new block. The chances of being selected increase with the amount of cryptocurrency staked.
- Responsibility. The selected pool validates transactions, ensuring they are legitimate, and adds them to the blockchain.
- Rewards. In return for validating transactions and securing the network, the pool earns rewards.

4.3.1. PoS features and advantages

One of the key features of proof of stake is that as a user's value increases, so does their opportunity to maintain the ledger and produce new blocks. The creator of a new block is chosen based on a combination of random selection and the user's stake, or wealth. A type of leader election occurs within the chain, and the new block is timestamped.

It is worth remembering that participants accumulate the transaction fees within a PoS protocol, thereby adding to their wealth as they go. This encourages steady and stable growth of the blockchain and reduces the instances of stalled transactions that can prevent chain growth.

Some of the primary advantages of proof of stake include:

- Rigorous security protocols – are incorporated into a PoS protocol

- Decentralization – the network is fully decentralized and not controlled by a single party or group
- Energy efficiency – energy consumption is extremely efficient, as a smaller amount of electricity, as well as hardware resources, are needed to produce and secure the blockchain
- Cost efficiencies – PoS currencies are far more cost-effective than those operating on PoW protocols.

All of these advantages point to how important proof of stake is within the context of the development of blockchain systems. In addition, a key factor in this evolving space is that validator addresses are known identities, sometimes actual addresses. This is an important fraud prevention feature.

In contrast, proof of work is a synchronous protocol that encourages miners to compete to be the first who can solve any problems within the block. A rewards system is used to incentivize this problem solving. However, this approach comes at a cost, with increased electricity usage and longer time spans to solve problems within the chain. These factors can slow the network down significantly and be costly to maintain.

4.4. Incentives

4.4.1. What are Incentives?

One of the most attractive and important features of a blockchain like Cardano is the fact that it is decentralized—there is no single point of failure, and the responsibility for maintaining the protocol and adding new blocks is distributed among many different Cardano nodes.

Due to its permissionless nature, everyone with the necessary enthusiasm and technical skills can run a Cardano node and participate in the network.

Running a Cardano node comes with costs, however. Even though

Cardano's proof-of-stake consensus mechanism ensures that the energy consumption of the network is many orders of magnitude lower than that of a proof-of-work blockchain like Bitcoin, running a Cardano node still requires a certain amount of computational power, computer memory, and bandwidth.

These resources are not free, a node operator either has to buy appropriate hardware or rent it from a cloud provider.

In addition, the node operator has to spend a lot of time setting up and maintaining the node, monitoring its performance, and performing necessary software updates.

So while decentralization is great, it comes with costs, and it is important to properly incentivize people to participate in the network and to do so in a way that is beneficial for the network as a whole.

For Cardano in particular, people need to be incentivized to become node operators and ensure their node is online when needed, participate in the election process of slot leaders and create a block when they themselves are elected.

Not every user on Cardano has the necessary technical skills, the time or the desire to run a node, however. Most users will simply delegate their ADA to a stake pool, earn staking rewards, and let the pool operator do the work. It is important to set the right incentives for this majority of people as well, so that they delegate to stake pools that are online when needed and faithfully follow the protocol. The more delegation a stake pool receives, the more blocks it will be able to produce, so delegating wisely is important for the security and efficiency of the network.

4.4.2. Types of Incentives

When we talk about incentives, we usually talk about *monetary incentives* in the form of *staking rewards* that are paid out in ADA.

However, there are also other types of incentives, such as idealism, morality, and the general desire to "do the right thing".

The main design goal of the Cardano incentives mechanism has been to align these monetary incentives with those non-monetary ones, so that the financial interests of individual node operators align with the interests of the network as a whole.

Nobody should be forced to neglect their own financial interests in order to do the right thing for the network.

This goal is by no means trivial to achieve, and a look at other blockchain projects shows that the conflict between individual and collective interests is a common problem.

Take Bitcoin as an example—in Bitcoin, node operators (often called *Bitcoin miners*) are paid for each block they add to the Bitcoin blockchain, and those rewards are *proportional* to the number of blocks they add.

Bitcoin mining comes with significant costs, and some of those costs can be reduced by centralizing the mining operation, for example by joining a mining pool.

If two Bitcoin miners form a mining pool, their expected rewards will stay the same, but their costs will be lower, as they can share the costs of running the mining hardware.

So by following their own financial interests, the miners will tend to centralize the network and form ever bigger mining pools.

This is bad for the security of the network as a whole, which relies on the assumption that no single entity controls more than 50% of the mining power.

The danger is very real and not just a theoretical concern: In 2014, the mining pool GHash.io controlled more than 51% of the Bitcoin mining power for several days, which caused a lot of concern in the Bitcoin

community. Eventually, GHash.io voluntarily reduced its mining power and promised not to exceed 40% of the total mining power in the future.

There was a real conflict between the financial interests of the miners and the security of the network, and the conflict was only resolved by the miners voluntarily reducing their mining power.

Cardano incentives have been designed from the ground up to prevent such conflicts of interest—by simply following their best financial interests, Cardano node operators and delegators will also do the right thing for the network, keeping it decentralized and secure.

This alignment will of course never be perfect, because people are obviously driven by more than just money. For example, somebody might delegate to a stake pool that pays less rewards than another pool, because the first pool is run by a charity or a non-profit organization that uses the rewards to fund a good cause. But the incentives scheme does not need to be perfect. It just needs to be "good enough" and work well even if people are not perfectly rational.

4.4.3. Desired Configuration

The security of the Cardano blockchain is guaranteed as long as there is an *honest majority of stake*. The majority of stake on Cardano must be delegated to stake pools who faithfully follow the Cardano protocol.

The more such stake pools there are, the more decentralized the network becomes. And ideally, those stake pools each should control a roughly equal amount of stake, so that none of them become too big and powerful.

On the other hand, the more stake pools there are, the less *efficient* the network becomes, both in a technical sense and from a financial point of view. More stake pools imply more network traffic and more redundant copies of data, and the operators of all those stake pools need to be incentivized with ADA, which means those funds are no longer available for other purposes like the treasury or staking rewards for ADA holders.

So there is a trade-off between security and decentralization on the one hand and efficiency on the other.

For this reason, the Cardano incentives mechanism is parameterized by a parameter k which denotes the desired number of (roughly equally sized) stake pools.

The goal of the incentives mechanism is to lead to a configuration where a solid majority of stake is delegated to a number of k stake pools of roughly equal size, whose operators' nodes are online when needed and provide additional infrastructure (like relay nodes).

4.4.4. Sources of Incentives

So far we have discussed why incentives are needed in the first place and what behavior they should actually incentivize. Of course those incentives have to come from *somewhere*, and in this section, we look at those sources, of which there are two on the Cardano blockchain.

4.4.4.1. Transaction Fees

Each transaction on Cardano incurs *transaction fees*, which consist of two parts: a low *flat fee* and a part that is proportional to the size^[8] of the transaction.

One purpose of transaction fees is to prevent *DDoS*^[9] attacks, where an attacker would try to flood and overload the network by issuing a large number of small transactions. Such an attack would be prohibitively expensive for the attacker, because the small flat fees for each transaction would add up to a significant amount, no matter how small each individual transaction is.

The other purpose of transaction fees is the one relevant for this chapter: Transaction fees are used to fund the incentives mechanism.

4.4.4.2. Monetary Expansion

Cardano has a maximum supply of 45 billion ADA, but when the

blockchain started operating, only 31 billion of those were in circulation.

The difference of initially 14 billion ADA is called the *ADA Reserves*, and those reserves are used to fund the incentives mechanism and indirectly - as we will see later - the *treasury*.

Every *epoch* (every five days), a certain fixed^[10] percentage ρ) of remaining reserves is taken for this purpose. This means that remaining reserves contribute less and less to the incentives over time when the amount of ADA in circulation gradually increases. This process is called *monetary expansion*.

Let us look at a very simple example and assume that $\rho=1\%$ of remaining reserves are taken each epoch (the actual value of ρ) is *much* smaller). Then for the first epoch, we would take 1% of 14 billion ADA, which is 140 million ADA. In the second epoch, remaining reserves have shrunk to 13,860,000,000 ADA, and 1% of that is 138,600,000 ADA, so available rewards for the second epoch are slightly less. For the third epoch, 13,721,400,000 ADA are remaining, and 137,214,000 are used for rewards and so on and so on. But remember that the actual decline is much more gradual!

The hope is that this decline in rewards that are coming from monetary expansion is compensated by an *incline* in rewards coming from transaction fees when more and more people start using Cardano and submit more and more transactions over time.

4.4.5. Distribution

Distribution of rewards happens once every epoch, so once every five days. Each time rewards are calculated and paid out, all transaction fees that have been collected since the last distribution are combined with a part of the remaining rewards given by ρ) into a virtual *rewards pot*.

A fixed percentage τ) of the rewards pot is given to the treasury. The rest is

distributed among the stake pools.

4.4.5.1. Splitting Rewards in a Pool

Once the rewards for a specific pool have been determined, they are distributed among the pool operator and ADA holders delegating to the pool.

In order to compensate the pool operator for his time and expenses, he can take a fixed amount of ADA and a *margin*, a percentage of what remains, from the pool rewards (both the fixed costs and the margin are set by the pool operator when he registers the stake pool).

After that, each delegator to the pool takes a share of what remains that is proportional to the amount of ADA that delegator delegated to the pool. (Note that the pool operator can and normally will be a delegator to his own pool, so he will get a share on top of his costs and margin as well.)

For example, consider pool operator Alice and ADA holders Bob and Charlie who delegate to Alice's pool. Alice has declared costs of 200 ADA and a margin of 1% when she registered her pool.

Let us assume that Alice delegates 100,000 ADA to her own pool, Bob delegates 200,000 ADA, and Charlie delegates 300,000 ADA. Let us further assume that the pool rewards for the epoch we look at are 1,000 ADA.

- Distribution starts by Alice taking her fixed costs of 200 ADA.
- After that, Alice takes her 1% margin of the remaining 800 ADA, which is 8 ADA.
- Finally, the remaining 792 ADA are distributed among Alice, Bob and Charlie proportional to their stake, i.e. in proportions 1:2:3. This means that Alice gets 132 ADA, Bob gets 264 ADA, and Charlie gets 396 ADA.

In the end, Alice received $200 + 8 + 132 = 340$ ADA, Bob received 264 ADA, and Charlie received 396 ADA.

4.4.5.2. Basic Idea

Now that we have seen how rewards *within* a pool are distributed, let us turn to the question of how rewards *among* stake pools are distributed.

The basic idea is simple: Pool rewards should be proportional to pool stakes. The more ADA delegated to a pool, the more rewards that pool should receive.

4.4.5.3. Problems

There are a number of problems with this basic idea, however:

- *Large Pools*: If rewards are proportional to stake, then the same problem that Bitcoin suffers from arises. Two pools will always have an incentive to merge, as the rewards for the merged pool will be the same as the sum of the rewards for the two pools, but the costs of the merged pool can be lower than the sum of the costs of the pools. This is bad for decentralization, because it will lead to a few very large pools controlling most of the stake.

For example, let us assume that Alice and Bob are both stake pool operators. Alice has a pool with 1 million ADA delegated to it, and Bob has a pool with 2 million ADA delegated to it.

Alice and Bob have both declared costs of 200 ADA per epoch.

Let us assume that for a specific epoch, Alice's pool's rewards are 10,000 ADA. Without refinements, when rewards are proportional to pool stake, Bob's pool's rewards will therefore be 20,000 ADA.

If Alice and Bob merge their pools, the new pool would have 3 million ADA delegated to it^[11] and would receive 30,000 ADA in rewards.

However, by merging their pools, Alice and Bob can save costs. So maybe instead of $200 + 200 = 400$ ADA per epoch, they can reduce costs to 300 ADA per epoch.

By receiving the same total rewards as before but having lower costs, Alice and Bob will be better off than before and have an incentive to merge their pools.

- *Being Online*: The whole point of having an incentives mechanism in the first place is to ensure the smooth operation of the Cardano blockchain. If a stake pool is not online when it is its turn to create a block, then the network will suffer. If pool rewards are solely based on pool stake and completely ignore pool performance, then pool operators have no incentive to be online when needed.
- *Sybil Attack*: An attacker could easily create many "attractive" pools with low costs and low margin, using different public keys for each of them to hide the fact that all of them are controlled by the same person. This way the attacker could capture more than 50% of delegations and gain control over the network. This kind of attack is called a *Sybil attack*, so named after the book "Sybil" by Flora Rheta Schreiber, which tells the story of the treatment of *Sybil* Dorsett for dissociative identity disorder.

The Cardano incentives mechanism has been designed to address these problems. While the basic idea still roughly holds, a number of refinements have been added to it to make the incentives mechanism work as intended.

In the following sections, we will look at each of those refinements in turn.

4.4.5.4. First Refinement: Large Pools

To prevent pools from becoming too large, the maximum proportion of the rewards pool that a stake pool can receive is limited by $1/k$, where k is the number of desired pools as explained in [Desired Configuration](#).

If k is 1000 and 10 million ADA are in the rewards pool for a specific epoch, then the maximum rewards that a single pool can receive are $10,000,000/1,000 = 10,000$ ADA.

Note that this does not constrain delegators in any way. They can still choose to delegate to large pools that have already attracted more than $1/k$ of total stake. It just makes such large pools financially unattractive to delegators, because they will receive less rewards.

Assuming $k=1000$ again, let's look at two pools, one with 0.05% of total stake, one with 0.15% of total stake. The first pool will receive 0.05% of the rewards pool. The second pool will *not* receive 0.15%, but only 0.1%. So while the delegated stake in the large pool is three times as large as the delegated stake in the small pool, the rewards are only twice as large. That means, all other things being equal (performance, margin etc.) that one staked ADA in the large pool will only earn 66.67% of the rewards that one staked ADA in the small pool will earn. This will gently nudge some delegators of the large pool to leave and join a smaller pool, one that is not yet *saturated*, i.e. one that has attracted less than $1/k$ of total stake.

4.4.5.5. Second Refinement: Being Online

Stake pools should be penalized for not being online when it is their turn. Rewards will be proportional to performance, ensuring that pools follow the protocol faithfully. Whenever a pool is elected to create a block, it should create that block. If it fails to do so, it should be penalized. We therefore want to modify the pool rewards by a *performance factor*, which is given by the number of blocks a pool *did* produce in an epoch divided by the number of blocks it *should* have produced. So a pool missing half its blocks should only receive half its rewards.

There is, however, a problem with implementing this idea directly. Leader

election on Cardano is *private*, so that only the elected leaders themselves know that they have been elected. We therefore do not *know* how many blocks a pool *should* have produced in an epoch.

We can, however, *estimate* this number. The probability to be elected slot leader in a given slot is proportional to the pool's stake. A pool with twice the stake will—on average—be elected twice as often. We also know that—again on average—there *will* be a leader every twenty slots. Combining these two pieces of information, we can estimate how many blocks a pool *should* have produced in an epoch and use that estimate to calculate the (approximate) performance factor.

Let us look at a pool that has 0.05% of total stake and has produced 8 blocks in a given epoch.

Because on average, there will be a block every 20 seconds (using the fact that a slot lasts one second at the moment), there will on average be 3 blocks per minute, 180 blocks per hour, 4,320 blocks per day and 21,600 blocks per epoch.

Our example pool has been delegated 1/2000 of the total stake, so on average, it will be elected slot leader $21,600/2000 = 10.8$ times in each epoch. If it produced 8 blocks, then we estimate its performance factor as $8/10.8 \sim 74\%$.

Note that where the "true" performance can never be greater than one (because a pool can never produce a block if it has *not* been elected slot leader), the *estimated* performance factor *can* exceed one. Leader election is (pseudo-)random, and a pool can get lucky and be elected more often than its stake would indicate.

This is no problem, however, because this effect will average out over time. Some pools get lucky, others have bad luck, but in the long run, this will even out.

4.4.5.6. Third Refinement: Sybil Prevention

With the refinements so far, a pool operator could still create many pools (under different names) and attract more than 50% of total stake while keeping each individual pool below the $1/k$ threshold.

To prevent this, the rewards of a pool are not only based on the stake of that pool and its performance, but also on the stake that the pool operator puts into his own pool, the so-called *pledge*.

Upon registration of a pool, the pool operator has to declare a pledge, a certain amount of ADA that he will delegate to his own pool. He is not forced to actually *honor* his pledge, but if he does not, then his pool will receive no rewards.

If he *does* honor his pledge and delegates the promised amount (or more) to his own pool, then pool rewards will depend on the amount of that pledge as well - the higher the pledge, the higher the rewards^[12].

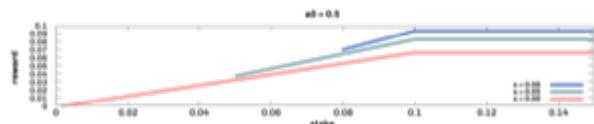


Figure 18. The effect of pledge on pool rewards: Rewards rise linearly until the pool is saturated and then remain constant, but the slope of the curve is steeper and the final plateau is higher for pools with higher pledge.

This means that a bad actor who wants to launch a Sybil attack now has a problem—nobody can stop him from creating many different pools, but he only has a limited amount of money, which he will have to split among all his pools. This means that each of his pools will have less pledge and therefore receive less rewards, thus making each pool less attractive for delegators, making it much harder for him to attract a significant proportion of total stake.

4.4.6. Undistributed Rewards

These refinements can lead to situations where not all funds in the rewards pool are distributed. This, however, is a feature, not a bug. When

this happens and some rewards remain in the rewards pool after rewarding all the stake pools, then the remaining funds are sent to the treasury, where they can be put to good use to improve and maintain Cardano.

4.4.7. Not Being Short-Sighted

Both pool operators and delegators might be tempted to change their strategies^[13] for short-term gains. A popular pool with many delegators, for example, could suddenly decide to increase their own margin. This would be short-sighted, however, because delegators would leave the pool, and the pool operator would end up with less rewards than before. Similarly, delegators might refrain from delegating to a good pool that just started, not taking into account that the pool will attract more delegators in the future.

This is handled by careful *ranking* of pools when they are displayed to users to decide where to delegate. Instead of basing that ranking) on the rewards of the last epoch, the ranking) is instead based on the expectation that in the long run, only the k) most "attractive" pools (with the most favorable combination of cost, margin, pledge and performance) will become saturated). This for example means that a pool that has just started and has not yet attracted many delegators will still be ranked high if it has a good combination of pool parameters that make it attractive, even if rewards will be somewhat lower while the pool is still growing.

4.4.8. Game Theory

Game theory is a branch of mathematics that studies strategic interactions between "rational" agents, i.e. agents that try to maximize some objective (like profit) rationally.

Real people in the real world are of course not always rational, but the assumption of rationality is a good starting point to understand how people will behave in a given situation.

One of the pioneers of game theory was John von Neumann, who contributed to a staggering amount of different fields in mathematics, physics and economy and is—among other things—famous for his work on the Manhattan project and on computer architecture. He has been called one of the most intelligent people in modern history.



Figure 19. John von Neumann (Los Alamos). By LANL - www.lanl.gov/history/atomicbomb/images/NeumannL.GIF (archive copy at the Wayback Machine), Attribution, commons.wikimedia.org/w/index.php?curid=3429594

Another pioneer of game theory was John Nash, who won the Nobel prize in economics in 1994 for his work on the subject. His fascinating story, full of triumph and tragedy, has been made popular for millions of people by the 2001 movie *A Beautiful Mind*.

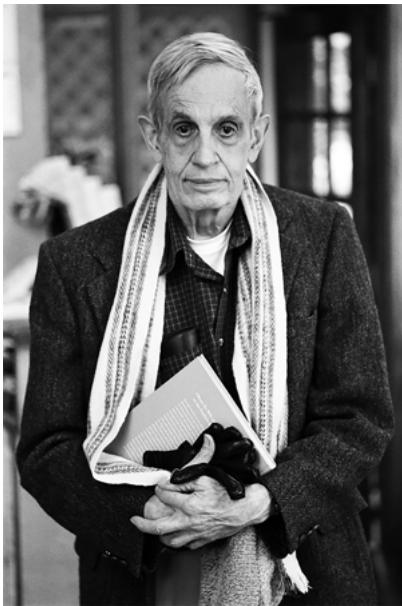


Figure 20. John Forbes Nash Jr. (June 13, 1928 – May 23, 2015) was an American mathematician and economist. Serving as Senior Research Mathematician at Princeton University during the later part of his life, he shared the 1994 Nobel Memorial Prize in Economic Sciences with game theorists Reinhard Selten and John Harsanyi. By Peter Badge / Typos1 - submission by way of Jimmy Wales, CC BY-SA 3.0, commons.wikimedia.org/w/index.php?curid=6977799

In game theory, Nash's concept of a *Nash Equilibrium* is of central importance. A Nash Equilibrium is a situation where no player can improve his payoff by unilaterally changing his strategy. So if all other players stick to their strategies, it would be irrational for any player to change his own strategy.

A famous example of a *game* in the sense of game theory is the so-called *Prisoner's Dilemma*: Two suspected burglars are being interrogated by the police in two separate interrogation rooms, unable to communicate with each other. The police want to get a confession from at least one of them, and they offer both of them a deal: If one of them confesses and the other one does not, the one who confesses will be set free, and the other one will be sentenced to ten years in prison. If both confess, both will be sentenced to five years in prison. If neither of them confess, both will be sentenced to one year in prison.

	Burglar B confesses	Burglar B stays silent
--	----------------------------	-------------------------------

Burglar A confesses	Burglar A: 5 years, Burglar B: 5 years	Burglar A goes free, Burglar B: 10 years
Burglar A stays silent	Burglar A: 10 years, Burglar B: goes free	Burglar A: 1 year, Burglar B: 1 year

This game has exactly one Nash Equilibrium: Both burglars confess and spend five years in prison.

Let us convince ourselves that this is indeed a Nash Equilibrium!

If Burglar A unilaterally changes his strategy and stays silent, while Burglar B sticks to his strategy of confessing, then instead of going to prison for five years, Burglar A will have to go to prison for ten years. The same is true for Burglar B. If Burglar B switches his strategy from confessing to staying silent, then he will have to go to prison for ten years instead of five, provided Burglar A sticks to his strategy of confessing.

On the other hand, none of the other three possible outcomes of the game are Nash Equilibria:

- If both stay silent and thus need to go to prison for one year, then each of them has an incentive to switch his strategy and confess instead, because then he will go free.
- If one of them stays silent and the other confesses, then the one staying silent can reduce his prison term from ten years to five years by confessing as well.

(The one confessing has *no* incentive to switch his strategy in this case, because then instead of going free, he would need to spend one year in prison. However, for an outcome to be a Nash Equilibrium, *none* of the players must have an incentive to switch his strategy.)

Many people find this result surprising and somewhat depressing—"Clearly", if both simply stayed silent, they would be

better off than if both confessed. And they maybe both *would* stay silent if they had a chance to communicate and coordinate. But the Nash Equilibrium is that both confess, and that is the only outcome that is stable in the sense that no player has an incentive to unilaterally change his strategy.

In a 2022 research paper^[14], game theory was applied to the Cardano incentives mechanism. It was shown that if the reward distribution follows the rules explained above, then each Nash Equilibrium of the "staking game" will have k) stake pools of the same size.

As stated before, real people in the real world do not always behave rationally, and they have other objectives than just maximizing their profit. But the mathematical result that under some idealizing assumptions, all Nash Equilibria of the staking game leads to the desired outcome of k) equally sized stake pools, is a strong argument that the Cardano incentives mechanism is well-designed and will indeed set the right incentives for pool operators and delegators, resulting in a secure and highly decentralized network for everyone to enjoy.

4.5. Ouroboros Consensus

Ouroboros is a family of Proof-of-Stake protocols powering the Cardano blockchain, among others. It is backed by years of research and numerous peer-reviewed publications. #

TODO: Insert References to the papers#

In this chapter we are going to build an intuition of how Ouroboros Work, so that when you go and read the formal papers describing the algorithm, you already have a good intuition about how it works, so that the mathematics make more sense.#

Let's start with a definition:

"Ouroboros is the name of a family of Proof-of-Stake consensus [1] protocols which provides foundation for Cardano and other blockchains. Distributed Consensus [2] is the mechanism through which a single, linear, eventually consistent [3], chain of blocks is established among all participants of the network. [4]" Intersect MBO

Such a short definition, entails 100s of human-years of engineering effort. We will spend the rest of the chapters going on each element of the definition to make sense of it.

4.5.1. Why do we need Consensus Algorithms?

All blockchains have a consensus algorithm, Bitcoin has Proof-of-Work, Filecoin has prof-of-storage, and Cardano has Ouroboros Proof-of-work.# But before going forward, we need to ask ourselves: Why are Consensus Algorithms necessary?

The fundamental reasons are simply facts of nature: * the **CAP Theorem** * The speed light is not instantaneous, so there is always **Latency**.

For example, Let's say you want to have Financial system that settles the transactions in all of the planet and you setup a Server on every continent.

image::ouroboros_distributed_system.png

You would want to: * Partition Tolerance, so that if there is Tsunami in the Atlantic, the system can continue to operate. * Strong Consistency, so that the transactions, on all the continents are registered and we can avoid having **double spending** i.e. a bad actor trying to use the same money in two different transactions. * Have low latency, so that all the transactions on each continent are registered very fast to speed up the payment system.

But as it is explained in the CAP and PCAELC theorems, *we can't have what we want, but [using a consensus algorithm] we might get what we need.*

4.5.1.1. The CAP and PACELC Theorems

Professor Eric A. Brewer defined the CAP theorem and paraphrasing it means:

When having a distributed [data store] system, you can only have two out of the three properties:

- Partition Tolerance: The system, continues to operate even in the presence of network errors.
 - Availability: Every request to a server receives a response.
 - Consistency: Every read receives the most recent write or an error.
-

Later Dr. Danil Abadi extended the CAP theorem with the PACELC Theorem. Paraphrased says:

If there is a Partition, decide between Consistency or Availability. Else choose between Low latency or Consistency. ---

image::ouroboros_PCAELC_Theorem.svg

In our example of a Global Financial system. A Partition would be like there is a tsunami that severs the interoceanic cables, leaving the continents separated in regions. If we want to maintain Consistency, then we can forbid new transaction on both sides of the Atlantic. So that we can always respond with the most up to date information. Or if we want to keep to keep the transactions to be registered (Availability) then the different servers records would slowly drift apart and loose Consistency.

Even if there is no tsunami. Just the fact that speed of light takes a few seconds to go to the other side of the planet, means that if we want to have a Consistent system. We should give up on the idea of Low Latency, to register each transaction.

But despite what the PCAELC theorem says. We **must** provide high consistency and low latency and be partition tolerant. So what do we do?

4.5.1.2. Nakamoto Consensus (Longest Chain Rule)

Satoshi Nakamoto introduced a solution to the problem of decentralized consensus by inventing the **blockchain** data structure and the **Longest Chain Rule**, forming what is now known as *Nakamoto Consensus*.

Transactions are collected into **blocks**, each of which references the hash of the previous block, forming a cryptographically linked chain. Participants in the network propagate new transactions using a **gossip protocol**, so that each node has a (roughly) consistent view of pending transactions.

These pending transactions are stored in each node's local **mempool** (short for "memory pool"). The mempool is a temporary buffer of unconfirmed transactions waiting to be included in a block. Each node maintains its own version of the mempool, and while they are usually similar, they can differ slightly due to network latency or policy differences (e.g., minimum fee requirements). Miners select transactions from their mempool when building a new block, often prioritizing those with higher transaction fees.

image::ouroboros_mempool.svg

Network participants (**miners**) [in Cardano SPOs] independently assemble candidate blocks from their mempool and **compete** to solve a cryptographic puzzle (Proof of Work) [in Cardano Proof of Stake]. The first miner to find a valid solution broadcasts the block to the rest of the network.

Once verified, the new block is appended to each node's local copy of the blockchain, and the transactions it contains are removed from the mempool. The process then repeats.

If the network becomes partitioned, or two miners [SPOs] solve the puzzle at nearly the same time, **temporary forks** may occur. Eventually, one branch becomes longer as more blocks are added.

image::ouroboros_longest_chain.svg

All honest nodes converge on this **longest valid chain**, discarding blocks from shorter forks. Transactions in the discarded blocks may be re-added to the mempool if they haven't already been confirmed.

image::ouroboros_longest_chain_consensus.svg

In this point is important to mention how similar the Original Nakamoto Consensus and the Ouroboros Consensus are. In fact Ouroboros is Nakamoto Consensus. The main difference is how they select the Network Participant (i.e. the Miner in Bitcoin or the SPO in Cardano).

Bitcoin uses Proof-of-Work and Cardano uses Proof-of-Stake or more precisely a Verifiable Random Function.

4.6. How does Proof-of-Work select a validator (miner) ?

In proof of work the Network Participants (a.k.a. Miners). **compete** with each other to find a "magic number" (a.k.a. Nonce) that make the block, fulfill a rule ("complexity"). So Proof-of-Work is like **race** where all the **miners** have to jump through hoops to be rewarded.

Proof of Work then is like hurdling race. Where there is only one winner.

4.6.1. But what is the "race" about?

We can see a block of the Bitcoin Network as data structure with the following fields:

image::ouroboros_basic_block_structure.svg

And then the "race" is about trying to calculate the Hash Function that

given the data and the Nonce, Returns a Current Block Hash that has the desired number of trailing zeros ("0000").

But in order to calculate the Current Block Hash the only way is to try Nonce numbers at random. like in this image we try with 0, 17... and so on

image::ouroboros_trying_with_nonce.svg

Until finally, we find the Nonce that gives a correct Current Block Hash:

image::ouroboros_valid_nonce.svg

As you can imagine this approach of random Nonce generation and testing if the hash calculated satisfies the complexity we desire is very computational intensive. That is the "Work" in the "Proof-of-Work"

But this approach has some disadvantages

4.6.2. Proof-of-Work disadvantages.

- Wasting Electricity Bitcoin is famous for wasting the same electricity as a small country. Going back our analogy the fact that all marathon runners have to run every race, With hopes of winning one reward. Wastes a lot of energy.
- It leads to centralization in Mining Pools. A mining pool is an association where miners, get together and decide to collaborate, with their computing power. To calculate the hash, and share the rewards. In our analogy is like if the Marathon runners decided to create teams, run together. And if one person of the team wins, it shares the rewards with its team.
- It leads to manufacturing centralization and e-waste. Since the equipment that mines in proof-of-work only has to do one operation (calculate a hash). This has created the development of specialized hardware to do it ("miners"). However, this also generates e-waste since once the miners are obsolete, they can't be used to anything else.

And although this disadvantages make headlines today. There was group of visionaries, leaded by Professor Aggelos Kisayas Chief Scientist at IO Research that saw them. And started to work on an alternative to Proof-of-Work. In the idea of Proof-of-Stake and Ouroboros in Particular.

4.6.3. Proof-of-Stake

If Proof-of-Work is a marathon, Proof-of-Stake is a **relay race**.

Only one runner, called the **slot leader**, runs each segment (block) of the race. That runner delivers the message (a block of transactions) to the next runner, who is randomly selected from a thousand others waiting to be chosen.

From this perspective, the benefits of Proof-of-Stake become clear:

- Only one runner means no wasted electricity.
- The hardware requirements are minimal: any generic computer capable of calculating a cryptographic hash function can participate.
- There is no incentive to form mining pools (teams), since the chance of being selected as the next slot leader is proportional to the amount of stake — i.e., one's **investment** in the network or the trust of other users that delegate their stake to the SPOs.
- This reduces incentives for centralization.
- The protocol is open: the hardware is not controlled by any one manufacturer, and even the software can be implemented by multiple independent teams.

4.6.3.1. How does Ouroboros (Praos) work?

Time in Cardano is divided into **epochs**, and each epoch is further subdivided into **slots**. Currently (2025), One epoch has 432000 slots. And each slot lasts 1 second. So each epoch is approximately 5 days.

During each slot:

- Servers (nodes) gather and broadcast transactions using a **gossip protocol**.
- These transactions accumulate in each node's local **mempool**.

Even though slots last 1 second. Not every slot results in a block. In fact, Cardano is parameterized so that on average one block is produced every 20 seconds. According to a parameter called "active slot coefficient" currently set at (0.05 or 5%).

At the end of a slot, If the slot happens to be one of the 5% of active slots. then it produces a block.

If a block is generated, a cryptographic lottery takes place.

All stake pool operators compute a Verifiable Random Function (VRF). This VRF takes as input: * a **random seed** that is updated each epoch. * the SPOs private key * and a label to distinguish repeated uses of the VRF.

The random seed is derived from data in the previous blocks.

The VRF produces a random output and a proof. The beauty of a VRF is that others can later verify the output was computed correctly from the given inputs without being able to guess it beforehand. Each node's VRF output is essentially that node's "lottery number" for the slot, and the proof is like a signed ticket.

On each slot, each SPO effectively asks (itself):

"Am I the slot leader for this slot?"

If the result of the VRF falls below a certain threshold, determined by the amount of stake the operator controls, then the operator becomes the **slot leader**.

That slot leader:

- Selects transactions from the mempool.
- Constructs a new block.
- Signs the block.
- Broadcasts the signed block to the network.

Cardano accumulates rewards (from block minting and fees) and distributes them to stake pools and delegators at the end of each epoch according to an incentive formula.

Then the stake pool operators (all), in the following slot, verify the previous block's validity including:

- The block's signature (to ensure it was signed by a registered pool's key).
- verify the VRF proof included in the block, which proves the slot leader indeed had an output below the threshold. Using the VRF proof, any node can confirm that "Yes, the creator of this block had the right to do so for slot N." This prevents malicious nodes from faking leadership. The Ouroboros Praos spec calls this the "proof of leadership" included in each block. If a block's proof is invalid or the node was not actually eligible, the block is rejected by others.
- They also validate all transactions in the block (checking signatures, UTXOs, etc.) as with any blockchain.
- Once the block passes validation, it's appended to the node's copy of the chain.

In Cardano, rewards are paid to stake pool operators (and delegators) at the end of each epoch, but with a delay of one full epoch after the one in which the rewards were earned.

The delay allows the network to:

Finalize the stake snapshot (used to calculate each delegators share)

Calculate the actual rewards based on the number of blocks produced, the active stake, fees collected, and the pool's parameters (margin, fixed cost)

And the process restarts for the next Epoch.

4.6.3.2. Why is it called "Ouroboros"?

The name **Ouroboros** — the ancient symbol of a snake eating its own tail — reflects how each epoch feeds into the next.

In Ouroboros, each slot's randomness (used to determine slot leaders) is derived from the data of previous epochs. The blockchain uses its **own past** to seed its **own future**, creating a secure, self-referential cycle.

That is how the snake eats its own tail.

4.6.3.3. Different versions of Ouroboros

The version of Ouroboros we have described can be better thought as Ouroboros Praos however different versions of Ouroboros exist by relaxing different assumptions.

- Ouroboros Classic (2017): first PoO with security proof, but required synchronous communication and had a public deterministic schedule.
- Ouroboros BFT (2018): interim federated version (used during Cardano Byron reboot)
- Ouroboros Praos (2018): introduced private VRF leader lottery, semi-synchronous security
- Ouroboros Genesis (2018): improved fork-choice, allowing trustless bootstrap and dynamic availability
- Ouroboros Chronos (2019): added secure time synchronization to Ouroboros (not yet implemented)
- (There are also Ouroboros Crypsinous (privacy-preserving variant)
- and Ouroboros Leios (throughput scaling)

4.6.3.4. Ouroboros Classic (2017)

The first version of Ouroboros demonstrated that a proof-of-stake protocol could match the security guarantees of proof-of-work, provided that at least 51% of the stake is controlled by honest participants. However, this version assumed a synchronous network, where all nodes are online and messages are delivered within a known, fixed delay. In this regard it was a leap forward but not yet practical.

4.6.4. Ouroboros BFT (2018)

Used during Cardano Byron reboot.

Allowed the federated blockchain.

Where trusted parties (IOG, Emurgo and Cardano Foundation), ran their own nodes.

4.6.4.1. Ouroboros Praos (2018 – Used in Cardano today)

The problem with Ouroboros classic is that it requires a random and distributed way to select the next stake pool operator to be selected. and when it comes to computers there is nothing absolutely random. So Ouroboros Praos implemented the concept of the Verifiable Random Function, that took as seeds of the random generation function, things that couldn't be controlled or predicted by anyone # the block number# the signing key of the stake pool operator that had to be submitted in advance# the amount of stake delegated in the stakepool operator# and the contents of the transaction in the block# including the hash of the previous block.# as you can see no single entity can predict or control any of those values that creates the randomness#

4.6.4.2. Ouroboros Genesis (2018 – Improved chain selection and bootstrap)

With Ouroboros Praos the main hurdles to have a correct proof of stake system were fulfilled now the next is to make it fast the first hurdle is that starting a new Cardano node from the beginning was very slow, we are talking about 36 hours slow, trying to catch up with the tip of the blockchain.# the naive solution to this is to have snapshots of the status of

the blockchain at a certain point in time. download that one big file, and assuming the file is correct, start to synchronize the copy of the blockchain from that point on. Ouroboros Genesis does it even better, in genesis, several points in the blockchain can be considered reliable, and therefore you don't even need to download all the history, the sync with tip can start immediately!

4.6.5. Ouroboros Chronos (2020/2021 – Decentralized time synchronization)

Chronos is a more recent development in the Ouroboros family, focusing on an often-overlooked aspect: time synchronization in a distributed system. By design, Ouroboros assumes some global notion of slots (1 second intervals). In practice, nodes rely on their local system clocks to know when slots start/end. If an adversary could significantly skew clocks or if there was no agreed time, consensus could break (e.g., nodes disagreeing on slot numbers). What Chronos introduces: A mechanism for nodes to securely synchronize their clocks using the blockchain itself as a reference. It effectively turns the blockchain into a decentralized time oracle. Chronos removes dependence on external time sources (like NTP servers), which could be central points of failure or attack. Instead, nodes periodically run a protocol (embedded in the blockchain process) to agree on the current time, detecting and correcting any drift or malicious deviations. This makes the system more resilient to time-based attacks (for example, an attacker can't easily isolate a node with a wrong clock to mess up its slot scheduling).

4.7. In conclusion

At this point, I hope you see how the development of the first practical proof of work consensus algorithm in Ouroboros was possible. how each iteration was an improvement over the limitations and assumptions of what came before. how the security of the algorithm is a mathematical probability given by the parameters in the blockchain. and how by varying some of our assumptions we can create tuned versions of the

protocol appropriate for certain applications.

4.8. Overview of Cardano Network Protocols

The Cardano network is a distributed infrastructure of interconnected Cardano nodes that communicate with each other to maintain distributed consensus of the Cardano ledger at a global scale. Contained within the node is a stack of protocols and mini protocols that form the network layer or data diffusion layer, which is responsible for opening and maintaining connections between nodes, diffusing transaction data, informing peers of new blocks, and sharing those blocks throughout the rest of the network.

4.9. A Brief History of Cardano Networking

Since its inception, the Cardano network has evolved and continues to change with the purposes of facilitating increased decentralization while meeting the growing demands of the Cardano blockchain.

- **The Federated Network:** Introduced in the Byron era in 2017, the network was maintained by IOG, Emurgo, and the Cardano Foundation. Nodes relied on static topologies to connect with one another.

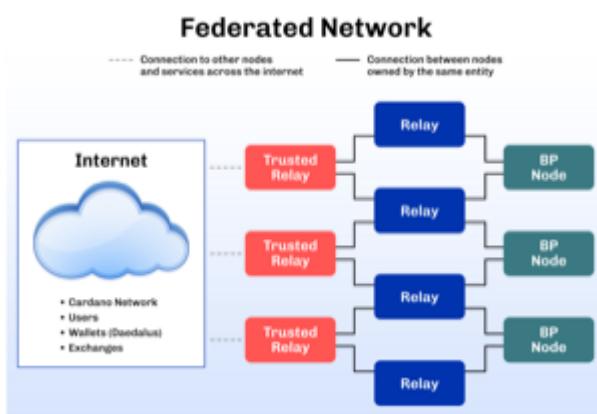


Figure 21. Federated Network

- **The Hybrid Network:** When the Shelley development phase began in 2020, the network was handed over to stake pool operators resulting in further decentralization of Cardano network. While nodes still required statically configured topologies to connect with one another, the community-made [Topology Updater](#) script helped automate this

process. The script lists peers based on geographical distance. It selects, skips, and again selects the peers to the end of the list to ensure that each node connects to a well-distributed variety of peers.

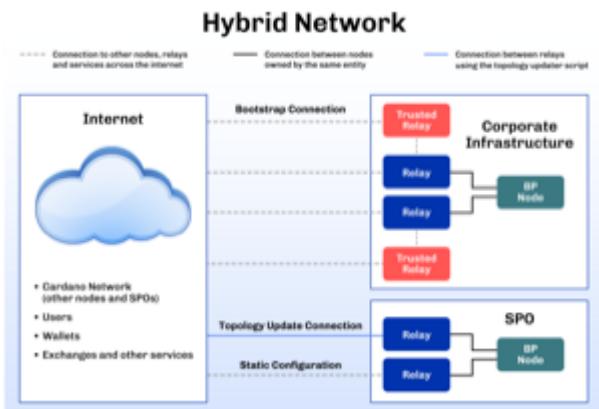


Figure 22. Hybrid Network

- **Dynamic Peer to Peer (P2P):** This evolution of the Cardano network allowed nodes to automatically connect with other peers. However, trusted peers or relays are still required to be statically configured in order to bootstrap and synchronize to the network.

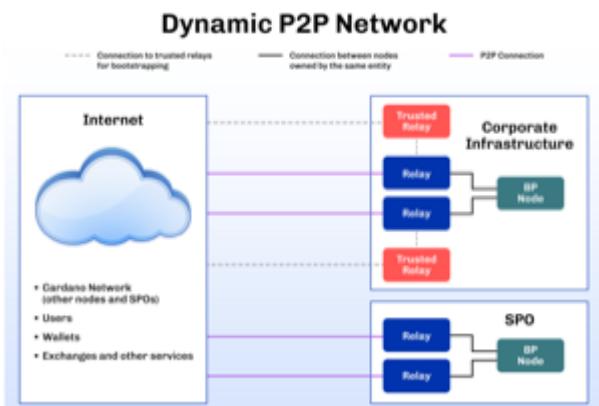


Figure 23. Dynamic P2P

- **Ouroboros Genesis:** The fully realized iteration of P2P, Ouroboros Genesis will allow nodes to self-bootstrap from the Cardano network, shedding the requirement for nodes to connect to trusted peers. However, peer selection still utilizes on-chain registered stake pool relays.

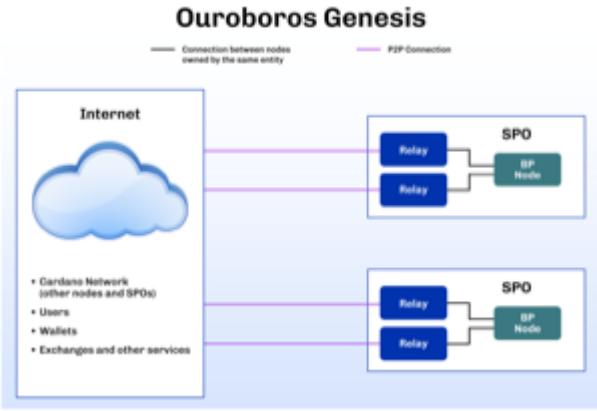


Figure 24. Ouroboros Genesis

- **Peer Sharing:** Peer sharing will allow the discovery of any node connected to the Cardano network, rather than only registered relays. This will allow non SPO nodes to contribute to running the network.

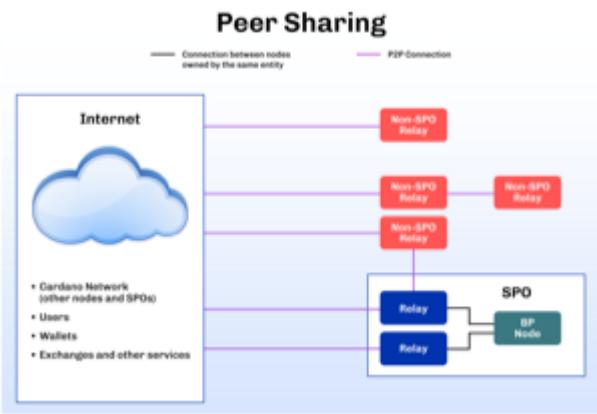


Figure 25. Peer Sharing

4.9.1. Protocol Overview

The Ouroboros network is currently designed to work over the Transmission Control Protocol/Internet Protocol (TCP/IP), the set of communication protocol standards the internet and most networks are shaped around. TCP/IP gives the Cardano networking protocols access to two-way communication between nodes. The guaranteed ordered delivery of network packets makes the TCP/IP protocols well suited for blockchain networks.

4.9.2. Mini Protocols

The *networking layer* of the Cardano node consists of smaller building-blocks called mini protocols. Structuring the Cardano network stack

around modular mini protocols allows for flexibility and reduces design complexity. At any given time a node will run many instances of mini protocols, often including many instances of the same mini protocol.

4.9.3. All Implemented Mini Protocols

- **Ping Pong Protocol:** A simple protocol used between a client and server to check responsiveness. A ping is sent to the server at regular intervals, and if the server is connected and responsive it will reply with a pong message.
- **Request Response Protocol:** Functions similar to the ping pong protocol, but further allows data exchange between server and client.
- **Handshake Mini Protocol:** Used to negotiate protocol version and parameters between client and server. It runs once upon connection initiation and consists of a single request from the client and a single response from the server.
- **Keep Alive Mini Protocol(keep-alive):** A member of the Node-to-Node(NtN) protocol suite, this protocol provides keep alive messages and measures the round trip times of these messages.
- **Chain Synchronization Protocol(chain-sync):** Allows for local synchronization of the blockchain via communication with several upstream and downstream nodes. This protocol is responsible for the transfer of full blocks when used as part of the Node to Client(NtC) protocol, whereas instances used as part of the Node to Node(NtN) protocol only transfers block headers.
- **Block Fetch Protocol(block-fetch):** Enables a node to request and download blocks from other nodes.
- **Node-to-Node Transaction Submission Protocol(tx-submission):** A protocol used to transfer transactions between nodes, where the initiator requests new transactions and the responder returns them. In this protocol, the server acts as the initiator while the client acts as the responder, making it well suited for a trustless setting.

- **Local Transaction Submission Protocol(local-tx-submission):** Used by local clients (typically wallets or CLI tools) for the purpose of submitting transactions to the local node. The client sends a request with a single transaction and the server either accepts the transaction (returning a confirmation) or rejects it (returning the reason).
- **Local State Query Protocol(local-state-query):** As part of the NtC protocol suite, the local state query mini-protocol allows querying of the consensus and ledger state. Queries depend on the era (Byron, Shelley, etc.) and basic operations include acquiring and releasing the ledger state or running queries against the acquired ledger state.

4.9.4. Inter-Process Communication(IPC) Protocols

The Cardano node IPC protocols are best described as protocol suites containing a number of mini protocols that allow communication between Cardano node processes. These protocol suites come in two flavors: Node-to-Node (NtN) and Node-to-Client (NtC).

4.9.4.1. Node-to-Node(NtN) IPC

The NtN IPC is responsible for transferring transactions, block header transfers, and fetching blocks between TCP connected nodes. This IPC uses the following mini protocols:

- **chain-sync**
- **block-fetch**
- **tx-submission**
- **keep-alive**

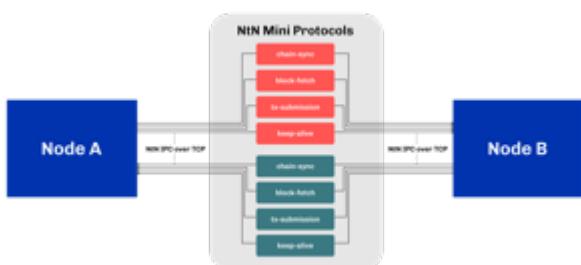


Figure 26. Node-to-Node

4.9.4.2. Node-to-Client(NtC) IPC

The NtC IPC facilitates the connection between a local full node and a local client that consumes data but does not actively participate in consensus, such as a wallet. The NtC allows local applications to interact with the blockchain through the connected full node. Much like the NtN IPC, the NtC IPC uses a similar design but rather than using TCP to connect to other nodes over a public network, the NtC uses local pipes. This IPC uses the following mini protocols:

- **chain-sync**
- **local-tx-submission**
- **local-state-query**

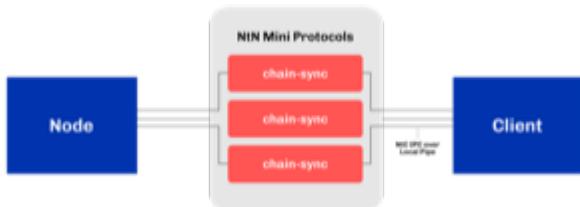


Figure 27. Node-to-Client

4.9.5. Multiplexing

Within the *network layer* of the Cardano node is a *multiplexing layer*, which allows the NtN protocol suite mini protocols to run in parallel through a single channel via TCP. The *multiplexing layer* is implemented via the **network-mux** standalone multiplexing library. This multiplexer uses a MUX thread to split the de-serialized messages from the mini protocols, assign a segment header and transmit the segments to the receiving DEMUX thread of another connected node, which in turn uses the segment headers to reassemble the messages from the sending node's MUX thread.

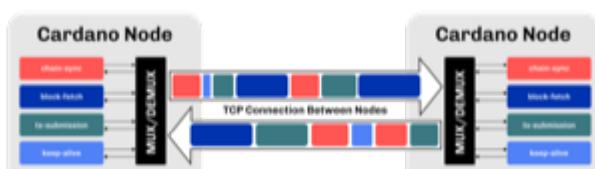


Figure 28. mux

4.9.6. Peer to Peer(P2P) Networking

The Cardano network benefits from the capabilities of a dynamic P2P system where nodes may automatically search for, connect with, and actively manage connections with other nodes allowing the network to be robust, decentralized, and flexible. The P2P stack is under continual development with regular increases in functionality. Through active peer selection and policy based exclusively on local information of the node, Cardano's P2P system significantly reduces data diffusion times across the network.

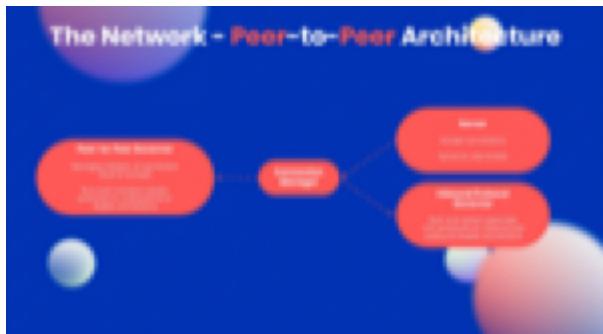


Figure 29. p2p components

The following components make Cardano's P2P system possible:

- **Outbound governor:** Manages outbound connections and classifies peers. It creates a connectivity map of the network and is responsible for dropping poorly-performing peer connections.
- **Server:** Responsible for accepting incoming connections.
- **Inbound protocol governor:** Once a connection is established, the inbound protocol governor manages the mini protocols running over that connection.
- **Connection manager:** Tracks the state of classified inbound connections, which determines whether connections are allowed to participate in consensus or whether they simply keep connectivity.

Outbound Governor

The outbound governor bears the responsibility of peer classification,

which includes regular promotion and demotion of peers into three distinct categories:

- **Cold peers:** A category of peers known to the node, but currently lack an established network connection.
- **Warm peers:** A connected peer used for network measurements without implementation of any NtN protocols.
- **Hot peers:** A connected peer with full NtN utilization.

All newly discovered peers are automatically added to the cold peer set. From this cold peer set, the outbound governor will begin the process classifying, demoting and promoting peers among the three peer classifications.

The outbound governor establishes connectivity between nodes by:

- promoting cold peers to warm peers
- demoting warm peers to cold peers
- promoting warm peers to hot peers
- demoting hot peers to warm peers

It is also responsible for establishing and maintaining:

- a number of cold peers (100 for example)
- a number of warm peers (between 10-50 for example)
- a number of hot peers (between 2-20 for example)
- a diverse set of warm peers in terms of hop distance and geographic locations based on the connectivity map
- frequent churn for hot, warm, cold, and unknown peer changes

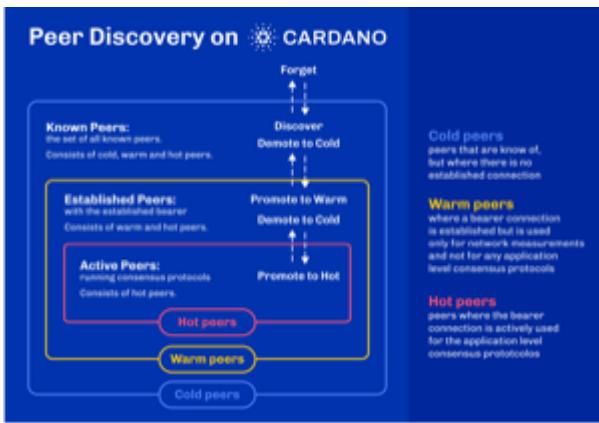


Figure 30. Peer Discovery

Research found 2-20 to be ideal for hot peers since block headers, rather than full blocks are exchanged among them, with the block body only being requested once and being fetched by the peer with the quickest path to the requesting node.

Warm peers can be quickly promoted to hot peers as candidates during the hot peer churn. The promotions and demotions between warm and hot rely on upstream measurements, while the churn between warm and cold is intended to discover network distances with the purpose of continually searching for better peers in a constantly changing network, where nodes may join and leave the network at any time.

For those wishing to further research the mechanisms behind Cardano networking protocols, please refer to the following sources:

- [Cardano Docs Networking Page](#)
- [The Shelley Networking Protocol Spec](#)
- [Introduction to the design of the Data Diffusion and Networking for Cardano Shelley Design Document](#)
- [Essential Cardano: Engineering dive into Cardano's Dynamic P2P Design](#)
- [IOHK Blog: Approaching full P2P node operations](#)
- [IOHK Blog: Dynamic P2P is available on mainnet](#)
- [IOHK Blog: Ouroboros Genesis: enhanced security in a dynamic](#)

environment

- IOHK Blog: Boosting network decentralization with P2P
- IOHK Blog: Cardano's path to decentralization TO DO:
 - Time of last update: Thurs 17th July 2025.
 - Asciidoc format = gist.github.com/powerman/d56b2675dfed38deb298

[1] In the cryptocurrency context, small-value UTXOs are known as *dust*.

[2] A certain minimum amount of ada – *minimum deposit*, which always needs to be included.

[3] These staking rewards come from an internal reward account, not a regular UTXO).

[4] Adding a datum to an output is optional, but outputs at script addresses without a datum are unspendable. Datums can also be added to public key addresses.

[5] The solution to the double satisfaction problem presented here is not the only option. For example, one could modify the atomic-swap script to allow only one input from the corresponding script address. However, this would hinder composability.

[6] On Cardano, the UTXO reference is represented by a pair consisting of the hash of the transaction that created the output and the index of that output in the list of all outputs of that transaction. The first output has index #0, the second output has index #1, and so on.

[7] There is an older type of address on Cardano, known as a *Byron address*, introduced during the first era of Cardano, the Byron era. Shelley addresses were introduced in the Shelley era. In the Byron era, Cardano used the plain UTXO model without staking, so Byron addresses are not discussed in this chapter.

[8] Here "size" refers to the size in *bytes* and not to the amount. A transaction sending ten million ADA (10,000,000,000,000 lovelace) is only slightly more expensive than one sending ten ADA (10,000,000 lovelace), and that slight difference is due to the fact that serializing 10,000,000,000,000 needs a few more bytes than serializing 10,000,000.

[9] DDoS stands for "Distributed Denial of Service".

[10] This percentage is given by a *protocol parameter*.

[11] This is of course not guaranteed. Delegators could object to the merger and redelegate their stake to other pools. We ignore this issue in this example.

[12] The magnitude of the effect of pledge on rewards depends on a system parameter called a_0 - the higher a_0 , the higher the effect.

[13] The strategy of a pool operator is to set costs and margin, the strategy of a delegator is to choose one or more pools to delegate to.

[14] *Reward Sharing Schemes for Stake Pools*, Lars Brünjes, Aggelos Kiayias, Elias Koutsoupias, Aikaterini-Panagiota Stouka, Euro S&P 2022

Chapter 5. Cardano governance

IO has been researching on-chain governance for some time, focusing on this crucial area since the original [Why Cardano](#) essay. Research papers dating back to 2017 have proposed a treasury for Ethereum Classic.^[1] Another paper, authored at Lancaster University,^[2] explored the concept of a treasury system and a viable, democratic approach to long-term development financing for Cardano.

Formal methods, machine-comprehensible specifications, and integrating a treasury system with this process for financial incentives are just some of the solutions pursued. Let's discuss some of the mechanisms used thus far to enable governance on Cardano. Each component complements the others, contributing valuable lessons and experience as the age of Voltaire unfolds.

5.1. Cardano Improvement Proposals (CIPs)

Similar concepts exist for other blockchains, such as the Bitcoin Improvement Proposals (BIPs) or the Ethereum Improvement Proposals (EIPs). While these share similarities, each works quite differently. In November 2020, [CIP-0001](#) was drafted to explain what a CIP is. A CIP is a formal, structured document proposing a solution to a common problem, highlighting various options and their trade-offs.

Anyone with ideas for enhancing Cardano can present CIPs. While CIPs are not a formal component of Cardano governance, they help steer the protocol and tooling in the right direction, benefiting the entire ecosystem.

The first CIP, created in October 2019 and later named [CIP-1852](#), extends

[BIP44](#) and documents how Cardano wallets manage keys and addresses. This document predates the formal CIP process. The CIP was named CIP-1852 by Sébastien Guillemot after the year Ada Lovelace passed away, with her birth year, 1815, used in the cointype field.

A CIP follows a standard format with a templated proposal structure that facilitates debate and evaluation. This structure allows community members to provide feedback on improvement recommendations or issues within a proposal. CIPs are stored as text files in a versioned [GitHub repository](#), where their revision history provides the proposal's historical record. For those not on GitHub, the auto-generated sister site, cips.cardano.org, serves as a user-friendly resource maintained by the Cardano Foundation.

Every CIP has the following format:

- Preamble
- Abstract
- Motivation
- Specification
- Rationale
- Path to active
- Copyright

The concept is developed as a written proposal and submitted as a pull request to the CIP repository after initial discussion and feedback. The CIP editors then publicly process the updated draft CIP as follows:

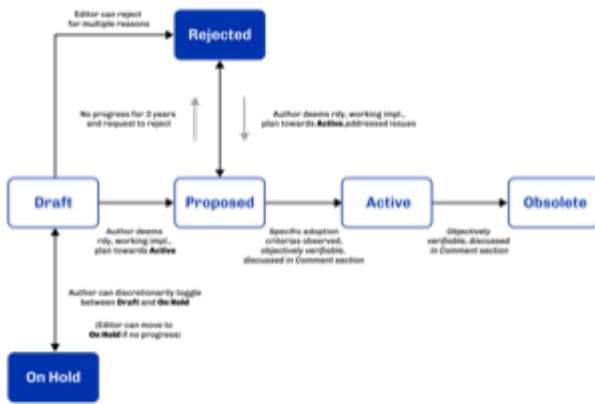


Figure 5.1: CIP workflow from CIP-0001

CIPs are processed in a semi-formal manner. Editors of CIP proposals meet regularly to discuss and evaluate ideas. The fortnightly meeting [minutes](#) are publicly available.

5.2. Project Catalyst: democratizing innovation in Cardano

[Project Catalyst](#), one of the world's largest decentralized innovation funds, is a key component of the Cardano ecosystem. It aims to drive Cardano's evolution through community-driven funding and development initiatives to address real-world challenges. Catalyst empowers the Cardano community to propose, vote on, and fund proposals to foster innovation and growth within the ecosystem.

Launched in 2020, Project Catalyst represents a significant milestone in the Cardano roadmap. It reflects the project's commitment to decentralized governance and community empowerment and marks the transition from a research-driven initiative to a community-led innovation hub.

The primary goals of Project Catalyst are to:

- foster innovation and development within the Cardano ecosystem
- enable community participation in governance and decision-making
- distribute funds transparently to high-potential projects that can solve

real-world problems.

5.2.1. How Project Catalyst works

The funding mechanism

Project Catalyst's funding mechanism is designed to support various innovative projects within the Cardano ecosystem. It operates through a series of iterative funding rounds, known as funds, each with its own budget, themes, and goals.

The funds for Project Catalyst are sourced from the Cardano treasury, which is replenished through a combination of:

- transaction fees, which are small fees collected from each transaction processed on the Cardano network
- reserve funds, which come from a portion of Cardano's initial supply allocated to a reserve for long-term ecosystem development funding.

Structure of funding rounds

Each funding round operates cyclically, typically lasting several weeks to a few months. Each fund is identified numerically (eg, Fund1, Fund2) and has a defined budget. The structure of a funding round generally includes the following phases:

1. **Proposal submission:** innovators submit their project proposals on the [IdeaScale platform](#). Each proposal must align with the thematic goals set for that particular fund.
2. **Community review:** during this phase, the community reviews and provides feedback on the submitted proposals.
3. **Voting:** any ada holder can vote on proposals using the Catalyst voting application. Voting power is proportional to the amount of ada held, incentivizing ada holders to participate actively in the governance process.

4. **Funding allocation:** based on the voting results, funds are allocated to the awarded proposals. Proposals with the highest votes receive funding until the budget for that fund is exhausted.

For more information about how Project Catalyst works, read the '[How it works' section](#) on Project Catalyst's website.

Voting process

Voting is a critical component of Project Catalyst, as it empowers the Cardano community to have a direct say in which proposals receive funding. This process ensures that decisions are decentralized and reflective of the community's collective priorities. Here's an in-depth look at how the voting process works:

1. **Download the Catalyst Voting App**—available on mobile devices.
2. **Registration.** The registration process involves taking a snapshot of the participant's ada holdings at a specific point in time. This snapshot determines the voting power of each participant.
3. **Reviewing proposals.** All proposals are publicly accessible on the *IdeaScale* platform and the voting application. Detailed information about each proposal, including the problem statement, proposed solution, team details, and budget, is available for discussion and feedback.
4. **Voting begins.** For each proposal, voters typically have multiple options to express their support or opposition. Once you've cast a vote on the blockchain, you cannot change it.
5. **Counting votes.** After the voting period ends, votes are tallied.
6. **The results are announced** publicly, detailing which proposals have been selected for funding.

After each funding round, feedback from the community is collected to identify areas for improvement in the voting process. Based on

community feedback, enhancements are made to the voting process, such as improving the user interface of the voting application, increasing security measures, and refining the proposal evaluation criteria.

Transparency and accountability

Project Catalyst places a strong emphasis on transparency and accountability through the following measures:

- publicly accessible proposals: all proposals and their progress are publicly accessible on the IdeaScale platform, allowing the community to track their development
- regular updates: funded projects are required to provide regular updates on their progress, including milestones achieved and funds spent
- community oversight: the community plays an active role in monitoring and evaluating the progress of funded projects, ensuring that funds are used effectively.

5.2.2. Success metrics in Project Catalyst

Measuring the success of Project Catalyst is essential to ensure that the initiative effectively fosters innovation and contributes to the growth of the Cardano ecosystem. The following metrics provide a comprehensive evaluation of its impact and effectiveness:

1. number of proposals submitted
2. number of proposals funded
3. community participation in voting
4. diversity of funded projects
5. impact of funded projects
6. budget utilization and efficiency
7. community feedback and satisfaction

8. growth in Project Catalyst participation
9. long-term sustainability and scalability.

Catalyst continues to improve and evolve with shared learnings. For example, funded projects from Funds 9–13 can now avail themselves of legal support from *Storm Partners*. This may include:

- Contract Review and Protection
- Global Dapp Compliance
- Crypto-Friendly Incorporation
- Decentralized Payment Compliance
- Smart Contract Legal Advice
- Token Classification Support.

The Catalyst team released their [Catalyst Horizons report](#) documenting various milestones. To dig deeper into the stats and trends, visit the [Reports](#) section of projectcatalyst.io for the latest data.

5.3. The age of Voltaire

As Catalyst took a ‘tactical pause’ after Fund9, there was a timely *Systemization of Knowledge (SoK)* research paper^[3] published, reflecting on the state of governance in ten blockchains, including Bitcoin, Ethereum, and Cardano.

The paper lists **seven properties** to assess different requirements for effective blockchain governance:

1. **Suffrage:** this property deals with participation eligibility. How inclusive is the governance mechanism?
2. **Confidentiality:** are decision-makers’ inputs protected from ‘external influences’?
3. **Verifiability:** can decision-makers confirm their input has been

considered in the output?

4. **Accountability**: are decision-makers held accountable for their input?
5. **Sustainability**: are decision-makers suitably incentivized?
6. **Pareto efficiency**: how effectively can decision-makers' intentions be turned into actions?
7. **Liveness**: how quickly can a blockchain's governance mechanism produce outputs efficiently?

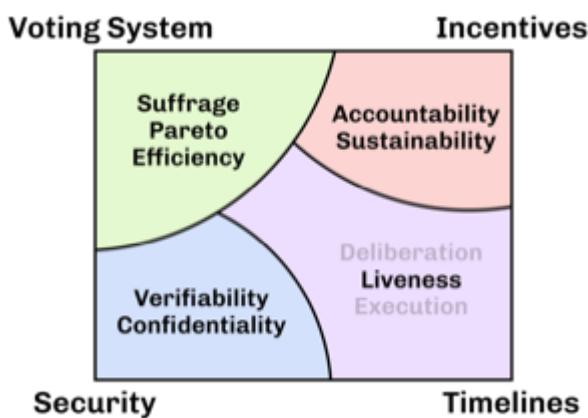


Figure 5.2: The partition map of governance properties from the 'SoK: Blockchain Governance' paper

The paper concludes that while each blockchain displays some of the properties, no blockchain meets all the requirements for effective governance. It was food for thought just before the dawn of the *age of Voltaire*.

CIP-1694

[CIP-1694](#) was named after the philosopher Voltaire's year of birth. It is arguably the most important CIP to date as it is a proposal to bootstrap the age of Voltaire. Co-authored by Charles Hoskinson, it is the first CIP he has gotten directly involved with. It's clear a lot of thought went into it, and it was intentionally written as a transitional, living document.

When Cardano was formed, there was a tripartite structure with EMURGO, the Cardano Foundation (CF), and IOG with remits for

ecosystem growth, governance, and engineering, respectively. The intention was always to move to a stake based governance model, where ada holders determine the future of the protocol.

CIP-1694 is the fruit of years of research. IOG has been working on a decentralized update system for some time. For example, they wrote a paper [_Updateable Blockchains^{\[4\]}](#) to explore ways to implement this vision.

Early in 2022, IOG and the CF held workshops to hammer out a way forward for the Voltaire development phase. The first question was ‘What is good governance?’. Charles Hoskinson explained in his ScotFest keynote that the answer was based on three different categories:

1. The **concept of representation** involves consent regarding decision-making authority. There are two types: **direct representation**, where individuals vote personally, and **delegated authority**, where individuals hand their vote to someone else. In CIP-1694, this role is called a delegate representative (DRep). This concept was discussed, but as yet not implemented, in Project Catalyst.
2. Governance requires a set of rules, often called a constitution, which serves as guardrails to provide stability. In a blockchain context, a constitution can be machine-readable. Formal specifications can act as blueprints for Cardano, enabling integration with an update system. Once a voting system is established, the constitution can be ratified, hashed, and embedded in a transaction. This allows users to sign a type of ‘end user agreement’ by signing the transaction.
3. **Institutions** are often seen as targets for decentralization. If the goal is to ‘kill the middleman’, why do institutions matter? At their best, institutions set standards and provide a review process conducted by domain experts. Institutions are essential for good governance as they are the custodians of knowledge and best practices. People can be biased, so objective, neutral bodies are sometimes necessary for

guidance. After careful consideration, it was determined that the most important ‘anchor’ institution would be a **members-based organization** (MBO) which should operate similarly to other open-source initiatives like the Linux Foundation, or the Cloud Native Computing Foundation (CNCF).

What is an MBO?

The MBO is a central hub that unites different groups, including thousands of stake pool operators, Cardano ambassadors, open-source projects running on Cardano, IOG, CF, EMURGO, and all ada holders. Members will own and run the MBO, staffing the steering committees.



Figure 5.3: Governance concepts defined, based on the slide from ScotFest 2022

The MBO, later christened *Intersect*, is Voltaire’s anchor institution, but it is not the only one. IOG has been steadily building out its presence in universities all over the globe, as well as opening the Hoskinson Center for Formal Mathematics, the Zero-Knowledge Lab, and—not forgetting—the Edinburgh Decentralization Index (EDI). Other institutions and MBOs will follow with different focuses and priorities.

“Institutions... their only job is to take complexity and turn it into simplicity”—Charles Hoskinson^[5]

CIP-1694 could fill a book on its own and, like all CIPs, is a living document that evolves with feedback. It aims to bootstrap the Voltaire development phase, integrating on-chain and off-chain components for ecosystem self-governance. The ultimate aim is a fully end-to-end, on-chain governance

layer for Cardano.

Where we came from – the five out of seven system

Before the Chang hard fork, governance transactions (eg, hard forks, parameter changes, etc) required a signature from at least five out of the seven Cardano governance (genesis) keys, currently held by the three founding entities. This process was always intended to be an ephemeral form of governance as we got through the earlier phases of the roadmap before Voltaire. There have traditionally been just two types of governance transactions:

- protocol parameter updates using transaction field n°6 of the transaction body
- movements of the treasury and the reserves using Move Instantaneous Rewards (MIR) certificates.

Where we are now

The CIP-1694 proposal encompasses two new ledger eras. The first era is called Conway, after the celebrated English mathematician John Horton Conway. The Conway ledger era:

- introduced SPO voting for hard forks
- provided an on-chain mechanism for rotating the governance keys
- rewired the ledger rules involving governance as outlined in CIP-1694.

For CIP-1694 to succeed, it is essential to realize the vision presented in the *Road to a Polyglot Ecosystem for Cardano* whiteboard [video](#). The new governance mechanisms will support multiple clients, enabling different development teams to employ different approaches, programming languages, and commercial unique selling propositions (USPs).

Charles Hoskinson's keynote at ScotFest 2022:^[6]

"So that's Voltaire ...it's deeply philosophical, it's the hardest thing I've ever

done in my life, it's the hardest thing you're ever going to do in your life, and we're going to get it done, because it needs to get done and I'm damn tired of our industry failing, and it's about time we can point to something and say 'you know what, we did it the right way'. We have to tend to our own gardens first. That was a lesson of Candide. So we have to fix Cardano's governance before we have the right to complain about any other person's governance."

2023 was all about debating how to implement CIP-1694. The CIP was written in a deliberately high-level, approachable format to stimulate discussion and feedback. The community did not disappoint with 50 [workshops](#), 30 in-person and 20 online, with over 1,000 participants from 20+ countries.

In addition to community-led workshops, IOG, EMURGO, and the Cardano Foundation co-hosted three governance workshops. The CF workshop took place in Zug, Switzerland, followed by EMURGO's workshop in Tokyo, Japan. The final workshop, hosted by IOG in Edinburgh in July 2023, marked the conclusion of the CIP-1694 design feedback loop.

Dozens of blogs have been written, and contentious issues have been argued over Reddit, X (Twitter), and Telegram. It is impossible to acknowledge every voice here, but you can dig into the finer detail by following Nicolas Cerny's [diary of events](#) on the Cardano Forum.

Governance on Cardano hit a milestone on Friday, June 30, 2023, when the [CIP-1694 pull request](#) was merged into the main branch of the Cardano Foundation CIP repository. The proposal's status advanced to the 'Proposed' stage.

As almost everything in Cardano takes the form of a transaction, getting the metadata standard correct is critical. Metadata allows developers to embed information specific to the context of the transaction. For example, the NFT standard (see [CIP-25](#), [CIP-68](#), [CIP-60](#)) on Cardano has evolved with new capabilities, unlocking with each roadmap release. Pi Lanningham

authored [CIP-0100](#) to clear up what metadata standards need to be introduced to enable the on-chain governance mechanisms proposed in CIP-1694.

[CIP-95](#) is a crucial CIP, which extends CIP-30 and describes the interface between webpage/web-based stacks and Cardano wallets. More specifically, it is a specification that defines the API of the JavaScript object that is injected into web applications. The CIP enables voting capabilities for governance tools. At the Edinburgh hackathon, decisions were made around open [questions](#), and the base design was approved.

As governance can be subjective, it's best you read CIP-1694 yourself, especially the Rationale and Changelog sections, which add context. If 2023 was the year we discussed governance, 2024 was all about implementation with Intersect as the main driving force.

5.4. Intersect: shaping Cardano's future

Intersect is a member-based organization for the Cardano ecosystem, founded in 2023. It serves as an aggregation point for the entire Cardano community, placing the community at the heart of Cardano's future development and harnessing the untapped potential of collective wisdom and economic energy. Intersect brings together companies, developers, individuals, institutions, and other ecosystem participants to shape and drive the future development of Cardano. It acts as a steward of the underlying blueprints and technology for the community, beginning with the Cardano node, core technology libraries, and components required to operate the protocol, along with all of its accompanying documentation, knowledge, and contributors.

This governance structure is designed to enhance decentralized growth within the Cardano ecosystem. It seeks to enable community-driven decision-making through democratic voting, defines clear roles and responsibilities, and ensures accountability. Intersect, as an MBO,

manages funds for ecosystem projects, aligns efforts with long-term strategic goals, and fosters inclusive community participation. It also improves coordination, increases accountability, and supports sustainable growth by providing a structured yet decentralized framework. Implementation involves community consensus, framework development, regulatory compliance, securing funding, and ongoing management. This model empowers the Cardano community and aligns with its vision of decentralization and transparency.

[Intersect](#) empowers a distributed network of builders and contributors who believe that every voice holds value and that collaboration leads to stronger outcomes. Members forge a secure, collaborative ecosystem to ensure Cardano's sustained growth and evolution in a safe space.

5.4.1. How Intersect operates

Intersect aims to administer the governing processes for Cardano's continued roadmap and development of the Cardano protocol. Intersect is currently facilitating the rollout of Cardano's governance features. Visit the Intersect latest [news](#) page to keep up to speed with the latest developments.

All Cardano ecosystem participants are welcome to become Intersect members. Made up of a distributed group of participants, including the foremost experts on Cardano and current ecosystem contributors, Intersect aims to facilitate healthy discussions and sound decision-making amongst its members and the community to uncover pain points and champion successes.

The five pillars of Intersect

1. Community support: Hosts events, hackathons, and conferences designed explicitly for developers within the Cardano ecosystem.
2. Governance: Champions and oversees Cardano's community-driven governance system, implemented through CIP-1694.

3. Technical roadmap: Helps orchestrate the delivery of the Cardano technical roadmap.
4. Continuity: Ensures system stability, Intersect facilitates Cardano's ongoing continuity.
5. Open-source development: Plays a role in coordinating the open-source development of Cardano's core technologies.

Intersect has a central governing board, similar to a city council, chosen and managed by its members. This board is supported by various committees and working groups, each focusing on specific areas or interests within the Cardano ecosystem.

Intersect's governing board started with five seats. Three were filled by founding members (seed funders, Input Output Global, and EMURGO), with the Intersect chief operating officer (COO) holding a temporary seat. Another seat was offered to the University of Wyoming's Blockchain Center for a one-year term.

The remaining two permanent seats were filled later in 2024 through an election process with Kavinda Kariyapperuma and Adam Rusch voted in by Intersect members. The Board now plans to expand from five to seven, with the number of seats elected by Intersect members rising from two to four. This doubles the community representation, which will make up the majority of the seats at the board leadership level. The board meets monthly and publishes agendas and [minutes](#) for transparency, and can be contacted at board@intersectmbo.org.

Intersect's funding

Input Output Global and EMURGO initially funded Intersect to get things running. For future funding, the community will be asked to vote on using funds from the Cardano treasury.

The Cardano Development Holdings (CDH), established in the crypto-friendly Cayman Islands, funds and facilitates the maintenance,

development, and growth of the Cardano ecosystem. It may receive direct funding from the Cardano treasury but can also receive donations from external sources for Cardano's development. All CDH funds are administered by Intersect. This structure was chosen for reasons related to accounting, legal clarity, and liability management.

Becoming a founding member comes with the following benefits:

- participate in steering groups, committees, and advisory boards, with the potential to establish new committees that will define Cardano's future governance
- access grants and contribute to developing Cardano's codebase while guiding a grant program to strengthen the Cardano protocol and ecosystem
- collaborate with other Cardano enthusiasts to build new partnerships and connections
- showcase contributions through member events, conferences, marketing materials, and member spotlights
- attend monthly meetings for updates on progress, committees, events, and funding opportunities
- participate in the annual meeting (in-person or virtually), focusing on Intersect activities, including voting on proposals. There are many [Community Hubs](#), located worldwide, hosting events.

Amending Intersect membership governance

Proposals to change Intersect's membership governance must be clearly documented. The board can approve amendments by a simple majority vote. There are various streams regarding the ongoing work that maintains and improves Cardano. Think of 'continuity' as the essential technical services needed to keep Cardano running smoothly. This includes bug fixes, upgrades, and new developments like CIP-1694. It's important to note that continuity focuses on the core infrastructure and

many other exciting community projects and applications are being built on top.

Cardano's vision and backlog refer to Cardano's future development, including new features and functionalities. These features may still be in the research phase or identified by the community for further exploration.

Open-source development

Cardano is an open-source project, with over 40 code repositories maintained by Intersect and its members. You can find more information and explore these repositories on [GitHub](#).

True open source means having the flexibility to choose different options. The Cardano Foundation also follows an open-source strategy. [Veridian Identity Platform](#), [Reeve](#) (enterprise financial reporting on-chain), [Aiken](#), [Kupo](#), and [Ogmios](#) all follow open-source principles and make life easier for developers on Cardano.

Acknowledging that Java is still the preferred language for many enterprise developers, the CF created [Yaci Store](#) (a modular library for Java developers) and the Veridian Identity Platform, as open-source tools with this audience in mind. The Veridian Identity Platform features a W3C-compatible mobile wallet for managing self-sovereign identities across Cardano and other blockchains. The wallet supports multiple standards, integrating key event receipt infrastructure (KERI) for interoperability to fit a broad range of use cases and enterprise adoption.

In addition, The Cardano Ballot project, a [Merkle Tree](#) in Java/Aiken, the Cardano conversions [library](#), and state channels layer 2 ([hydra-java Client](#)) were all made open source. The CF also made the [rewards calculation](#) open source to enable anyone to perform and validate the rewards calculation independently of a single implementation.

Open source office (OSO)

The OSO manages Cardano’s open-source program and community. They ensure open and effective communication with the wider open-source community. Intersect manages contracts with companies working on Cardano’s development. Office hours are held twice a month, with an open format welcoming new topics. The OSO regularly produces content for the community. For example, the [Contribution Ladder](#) serves as a framework to help new contributors engage with a project.

Delivery assurance ensures that projects are completed on time and according to specifications. This involves managing risks, tracking progress, and taking action to ensure successful completion. The approach varies based on the project’s size, complexity, and potential risks. You can review information about contract work completed, and in progress, on the Intersect knowledgebase.

5.5. Intersect structure

Intersect operates on the principle of community leadership for Cardano’s development. This is achieved through standing committees formed and led by its members. At present, seven standing committees report to the ISC (Intersect Steering Committee).

Standing committees are permanent committees covering various functions critical to guiding Cardano’s ‘continuity’ (ongoing maintenance and development), shaping Cardano’s constitution, and supporting internal membership needs. While changes can be made as the committees and their goals evolve, they are intended to be long-lasting.

Working groups are temporary and typically support a standing committee’s broader objectives. They may also be formed to tap into expertise outside of Intersect’s membership. Flexible and less formal than committees, working groups can address diverse topics relevant to Cardano’s development.

5.5.1. The civics committee

The civics committee acts as a guide and supervisor for the Cardano community on governance issues:

- develop and manage ways for the community to actively participate in Cardano's governance
- collaborate with subject matter experts when needed
- assist the Cardano constitutional committee as requested.

This committee is crucial for ensuring Cardano's governance system is accessible, fair, inclusive and transparent. The civics committee addresses topics like:

- ratifying the constitution: facilitating a period for community approval
- off-chain discussions: tracking and maintaining a record of informal discussions about proposals before they are formally presented
- on-chain voting tools: monitoring these tools to ensure they are fit for purpose
- voting guidelines: developing and updating clear instructions and best practices
- governance improvements: providing non-binding recommendations based on community input to enhance Cardano's governance system.
- budget guidance, for example, they released a [guidance document](#) for the 2025 process.

5.5.2. Membership and community committee (MCC)

The MCC helps build a strong Cardano community within Intersect. They achieve this by:

- attracting new members through effective sales and account management
- supporting existing members with helpful resources and events

- offering [grants](#) to create useful community tools
- providing education and hosting engaging events

This committee creates a space for Cardano enthusiasts to connect, share knowledge, and collaborate on projects. For example, a grant was awarded to Ryan Wiley (Cerkoryn) for his changwatch.com dashboard. This tool displays real-time governance action data through donut charts, breaking down participation in governance actions by DReps, SPOs, the CC, and an aggregated total of all groups. This highlights which entities sway over each proposal type based on stake-weighted delegation and voting thresholds. Anyone in the Cardano ecosystem can flag specific centralization concerns with this user-friendly dashboard.

The MCC manages Intersect memberships, ensuring everyone gets the most out of the program and can contribute to Cardano's development. They also review proposals for community working groups. Public meetings are held every four weeks, and the minutes are [public](#).

5.5.3. Growth and marketing committee

This committee takes the form of a strategic planning body, focused on impactful marketing and ecosystem growth and adoption. The *Marketing Strategy Working Group* sits alongside the committee. Its mission is to formulate Cardano's long-term marketing strategy and go-to-market plan. It endeavours to leverage the hive mind of the Cardano community and DReps.

5.5.4. The technical steering committee (TSC)

The TSC oversees Cardano's technical health, ensuring that decisions are based on solid technical knowledge and best practices.

This committee brings together key players to ensure Cardano's development runs smoothly. They handle contracts with developers, create technical proposals, and review ideas from the Cardano

community, like updates or major changes to the network.

The TSC leads in guiding the development of Cardano's ongoing technical foundation. They provide in-depth technical analysis and advice for everything from development projects to network settings. Think of them as the guardians of Cardano's technical well-being. The minutes from their meetings are [public](#).

5.5.5. The parameters committee (PC)

The PC is a subcommittee within the TSC that focuses on optimizing Cardano's settings. They ensure these parameters are set based on the best technical knowledge available. They consider factors like economics, security, and network performance when recommending updates to Cardano's core settings. Regular meetings discuss updates and consider proposals from the community to adjust parameters. There are advisory groups within the PC, such as:

- economic parameters advisory group
- network parameters advisory group
- technical parameters advisory group
- governance parameters advisory group.

Membership in this technical group is by invitation only. However, anyone can submit suggestions for parameter changes on the Cardano Forum. The PC also participates in monthly calls with Cardano's stake pool operators to share updates and answer questions.

Matthew Capps' [X thread](#), *Protocol Change Proposal-001: Chronology of Documented Events*, provides insight into the careful consideration and deliberation involved in a parameter change.

5.5.6. The open source committee (OSC)

The OSC owns the roadmap (strategy) for Cardano's open-source projects,

advising others on open-source best practices, and acts as a central point for anyone building within Cardano’s open-source environment. This committee helps developers navigate the world of open-source development on Cardano.

The OSC tackles several key areas:

- defining what ‘open source’ means for Cardano projects
- developing and maintaining Cardano’s open-source strategy
- overseeing pilot projects for open source on Cardano
- establishing best practices for open-source development within Cardano
- creating a model for future open-source projects within Intersect
- running the *Developer Advocate Program*

An Open Source strategy can incur risks if relying on unpaid contributors to regularly show up. With this in mind, Christian Taylor proposed a solution called the [paid open source model](#), which can be adapted to other projects, offering hope for a more sustainable and secure open source ecosystem.

5.5.7. Cardano budget committee

The budget committee manages Cardano’s operational costs and creates a yearly budget for community review and approval. The committee provides clear information on Cardano’s core expenses, ensuring transparency for the community.

How it works:

- the *product committee* provides a list of approved projects
- the *budget committee* will then assign costs to these projects and create a budget proposal
- the community will vote on the budget proposal at the annual members

meeting (AMM)

- upon approval, funds will be allocated from the Cardano treasury through on-chain voting.



Figure 5.4: Provisional Budget process timeline

Product committee

The product committee manages and tracks the roadmap for development items. Their responsibilities include facilitating processes to converge on a shared vision and roadmap. The committee encouraged the community to submit projects for consideration for the 2025 roadmap, with an [explainer](#) to guide them through the process.

Working groups

Intersect forms temporary groups called working groups to address specific needs as they arise. These groups can focus on any topic and operate less formally than the permanent committees. Each working group has the following specifications:

- defines its purpose, operating procedures, and member roles and responsibilities in a terms-of-reference document
- observes participation limitations, like application processes or elections, are set with board approval
- establishes meeting frequency and procedures
- works under a specific committee but may collaborate with others
- reports their progress and findings to their overseeing committee(s).

Intersect's **hard fork working group** was one of the busiest as it oversaw the Chang and Plomin upgrades. It began as just three attendees on the first call in February 2024. As the working group has met at least weekly, often more, the attendee list grew to over sixty. The group's last call was almost a year later on Tuesday, February 11, 2025, fulfilling its remit to oversee all aspects of the Chang and Plomin hard forks to a successful conclusion.

To learn more, head over to the Intersect [working groups](#) space for a complete list and further details. Committees are elected by Intersect members only, using a one-member, one-vote system. Elections take place twice yearly. Half of each committee's members were elected in the first elections in October 2024, and the remaining members were elected in 2025. The official final numbers for newly elected committees were made up of a "Who's Who" of Cardano's brains trust:

Committee	Seats	Candidates	Votes
Intersect Steering (ISC)	2	11	
Cardano Civics (CCC)	6	13	
Membership & Community (MCC)	5	22	
Growth and Marketing (GMC)	5	26	
Technical Steering (TSC)	5	10	
Open Source (OSC)	5	12	
Cardano Budget (CBC)	5	16	
Cardano Product (CPC)	3	9	

Figure 5.5: Committee Election results

5.6. Cardano governance: a three-part approach

Cardano's future governance leans on three key pillars:

- 1. On-chain decisions:** this system (detailed in CIP-1694) allows ada holders to directly influence Cardano's development through proposed governance actions voted on-chain.
- 2. Cardano constitution:** this evolving document outlines core rules to guide Cardano's growth during its transitional governance phase. A fully-fledged constitution will be drafted with community input

throughout the year, culminating in a final version ratified by both delegates and ada holders. [CIP-0120 \(constitution specification\)](#) proposes a standardized technical format to make the document accessible for tools to read, render, and write.

3. Institutions:

these provide spaces for discussion, collaboration, and recommendations that ultimately feed into on-chain decision-making.

These three elements work together to create a robust governance system that can adapt and improve over time, driven by the Cardano community. The age of Voltaire is still in its infancy, and four key roles will be pivotal as CIP-1694 becomes a reality.

Ada holders

Ada holders play a crucial role in Cardano's governance. They can:

- delegate their vote: choose representatives (DReps) to cast votes on their behalf
- become a DRep: represent themselves or others in on-chain voting
- shape Cardano's future: propose changes to the network by submitting on-chain governance actions
- stay informed: review submitted governance actions and cast their vote on them.

By actively participating, ada holders collectively drive Cardano's development.

DReps

The *age of Voltaire* introduced delegate representatives (DReps), a new concept central to Cardano's governance as defined in CIP-1694. DReps, alongside stake pool operators and the constitutional committee, will vote on proposals that shape Cardano's future.

Any ada holder can become a DRep. This means ada holders can choose to

directly participate in voting or delegate their voting power to DReps they trust. There are two predefined DReps: the *abstain* and the *no confidence* DReps. These options allow ada holders to either not participate in governance or automatically express a yes vote on any *no confidence* action, providing a directly auditable measure of confidence in the constitutional committee.

Why delegate?

Delegation allows ada holders to empower representatives who are potentially better equipped to make informed decisions on their behalf. This fosters a more democratic system where everyone has a say, even if they don't have the time or expertise to delve into every proposal.

The first community DRep workshop took place on January 20, 2024, in Oslo. This initiative was funded by a Catalyst Fund10 [proposal](#) from Eyetein Hansen, Adam Rusch, Ekow Harding, Jose De Gamboa, Thomas Lindseth, and Yuki Oishi. Many more workshops followed.

Intersect collaborated with the IOG education team on the [DRep Pioneer program](#), an online interactive training course for nominated delegate representatives (DReps) involved in Cardano's proposed governance structure.

Stake pool operators (SPOs)

Think of SPOs as the caretakers of Cardano's network. They run stake pools, which are essentially servers that keep the blockchain running smoothly. These operators typically:

- own or rent servers running the Cardano node (both block-producing and relay nodes)
- hold the pool's key
- maintain and monitor the network nodes.

SPOs play a vital part in Cardano's on-chain voting governance by:

- proposing changes: they can submit governance actions to improve the network
- shaping the future: they can review and vote on proposed governance actions.

The constitutional committee (CC)

Unlike other Cardano governance bodies, the CC operates independently and entirely outside of Intersect. It is one of three key groups (alongside SPOs and DReps) that vote on proposals to change Cardano's core systems through governance actions. The CC's primary function is to review proposed changes with a limited focus—ensuring that they align with the principles outlined in Cardano's constitution.

5.7. Cardano governance flow

CIP-1694 outlines Cardano's on-chain governance process, but it's also important to consider the supporting off-chain activities.

Off-chain proposal discussions

Before proposals are submitted to the blockchain for official votes, there is a crucial off-chain stage for discussion and refinement. Off-chain debate allows for:

- clearer proposals: proposers can share more details, rationale, and supporting evidence to ensure everyone understands the idea
- community input: reviews, comments, and feedback help improve the proposal and gauge overall sentiment
- informed voters: off-chain discussions generate valuable context, which becomes part of the official proposal (metadata) on-chain, aiding voters in making informed decisions
- reduced burden: filtering and refining proposals off-chain minimizes the number of votes submitted on-chain, reducing stress on the

blockchain.

Without a strong off-chain process, governance could falter, as ideas may not undergo thorough discussion or refinement. On-chain proposals might lack the necessary context, making informed voting difficult.

Intersect recognizes the importance of off-chain discussions and has issued a grant to establish a dedicated proposal discussion forum. More information about the grant can be found in [Intersect's GitBook](#).

Submitting on-chain governance actions

Once a proposal has been thoroughly discussed and refined off-chain, it is ready for the official vote on the blockchain. This is known as on-chain governance action submission. Proposals can be submitted on-chain through the Cardano command-line interface (CLI) or via GovTool's user-friendly interface. The specific content required for an on-chain proposal depends on the type of governance action being submitted. Proposers can optionally add metadata to provide additional context and information alongside the proposal.

5.8. Governance actions

What are governance actions? Imagine them as proposals submitted on the Cardano blockchain for voting. These proposals trigger events on the blockchain through transactions and have a set timeframe for voting before they expire and can't be enacted. Any ada holder can submit a governance action for on-chain voting. Once a proposal is submitted and recorded on the ledger, voters can vote through separate voting transactions. Note a governance action requires a refundable deposit of 100,000 ADA to prevent spam and demonstrate commitment. The deposit is returned after the action is finalized.

CIP-1694 defines seven categories of governance actions:

1. Motion of no-confidence: creates a state of no-confidence in the current

constitutional committee.

2. New constitutional committee or quorum size: proposes a change to the members of the constitutional committee and/or to its signature threshold and/or terms.
3. Updates to the constitution: proposes a change to the off-chain constitution, recorded as an on-chain hash of the text document.
4. Hard fork initiation: triggers a non-backward compatible upgrade of the network.
5. Protocol parameter changes: proposes a change to one or more updatable protocol parameters.
6. Treasury withdrawals: proposals for how to spend funds from the Cardano treasury.
7. Info: simply provide information and don't require enactment.

Governance actions vary in complexity. Info Actions and Treasury Withdrawals are relatively straightforward, but Constitutional updates require metadata, a constitution hash, a URL, and a proposal policy script. Protocol Parameter Changes and Hard Fork Initiation are non-trivial to implement, requiring technical expertise. To demystify some of the processes, the CF published [flow charts](#) to encourage wider participation.

5.9. Registering as a DRep on-chain

DRep registration occurs on the blockchain and can be done through the Cardano CLI or GovTool. During registration, DReps can optionally add details about themselves (metadata) to help ada holders decide who to delegate their votes to.

On-chain DRep delegation

On-chain delegation allows ada holders to give their voting power to a DRep of their choice. These DReps then cast votes on their behalf regarding active governance actions.

To make an informed decision, individuals should review the metadata submitted by DReps during registration. This metadata might include details like their expertise, areas of interest, and even past voting history.

The delegation process happens on the blockchain and can be done through the Cardano CLI or GovTool.

On-chain voting process

On-chain voting is where the three voting groups (DReps, SPOs, and the CC) cast their votes on active governance actions.

For a proposed governance action to be approved and implemented, it needs to meet specific voting thresholds set by Cardano. These thresholds may vary depending on the type of governance action being voted on. In simpler terms, some proposals might require approval from all three voting groups, while others might only need a certain percentage from a specific group.

Governance action type			
	CC	DReps	SPOs
Motion of no-confidence	✗	✓	✓
New constitutional committee or quorum size	✗	✓	✓
Update to the Constitution	✓	✓	✗
Hard-fork initiation	✓	✓	✓
Protocol parameter changes	✓	✓	✗
Treasury withdrawal	✓	✓	✓
Info	✓	✓	✓

Figure 5.6: Voting on governance actions (based on a table from Intersect's [documentation](#))

Following the on-chain voting process, a governance action is considered approved (or ratified) if it meets the specific voting thresholds set for its type. These thresholds determine the level of consensus needed from the different voting bodies.

For DReps, only 'Yes' votes contribute to the passage of a governance action, and not voting counts as 'No'.

DRep voting is based on one lovelace, one vote. The passing threshold for an action is the stake voted ‘Yes’ expressed as a percentage of the total stake. Stake voted ‘Abstain’ or delegated to inactive DReps is excluded from the total stake. Stake delegated but not voted is added to the stake voted ‘No’ and included in the total stake.

A DRep becomes inactive by not voting for a number of epochs of five days. This number is set by the protocol parameter `drepActivity`. A DRep can become active again simply by casting a vote.

Once ratified, a governance action is then enacted on-chain, meaning it’s implemented and becomes part of the Cardano protocol according to a well-defined set of rules.

Proposals categorized as [Info actions](#) are a special case. Since their purpose is solely to provide information, they don’t require enactment and have no impact on the protocol itself. Their ratification simply acknowledges their informational value.

Cardano’s governance process emphasizes open communication. This includes not just discussing proposed governance actions beforehand, but also sharing their outcomes after the on-chain voting is complete.

A complete governance cycle starts with off-chain discussions and should end with the community being informed of the outcome. Sharing results, especially for ratified (approved) proposals that will be implemented, helps **close the loop** and keeps everyone informed.

Ideally, the outcome should be communicated through the same off-chain channels where the original proposal was discussed. This fosters transparency and a sense of connection throughout the entire governance process.

5.10. SanchoNet: testing ground for Cardano’s future

SanchoNet was named after the character Sancho Panza, Don Quixote’s

companion in Miguel de Cervantes' literary classic. SanchoNet is ultimately about transforming an aspirational digital Barataria into an on-chain governance reality on Cardano mainnet. Note that SanchoNet is not another incentivized testnet (ITN), but a testnet where test ada is used to stress test experimental features. SanchoNet was rolled out in six phases, with each Cardano node (cardano-cli) release enabling new governance capabilities.



Figure 5.7: SanchoNet roadmap

SanchoNet goes beyond simple testing. It also serves as a platform for:

- informing the community: keeping the Cardano community updated on the ongoing development of Voltaire
- engaging stakeholders: encouraging community participation and feedback on the evolving governance features
- building a collaborative future: as SanchoNet matures, it aims to become a space where ideas become reality, contributions shape the ecosystem, and fully decentralized decision-making takes root.

SanchoNet has proven itself robust to adversarial behavior. Mike Hornan of Able Pool SPO orchestrated a sustained community-driven stress test on SanchoNet, ensuring the network has the required resilience to handle thousands of governance actions concurrently.

5.11. Governance tools

Cardano's vision is a truly decentralized blockchain fueled by

collaborative decision-making. Effective governance requires more than just principles and processes. It needs the right tools to empower the community and enable consensus across the Cardano ecosystem.

These tools will equip the Cardano community to actively participate in on-chain governance actions. Intersect has already issued grants to develop key components of this toolset. Find out more about these [grants](#) here.

The [GovTool](#) is a central hub for interacting with Cardano's on-chain governance system, and testing upcoming features. GovTool is fully open sourced and maintained by Intersect members WeDeliver, Byron, Lido Nation, and Bloxico. It enables users to connect their wallets to mainnet to participate in governance. They can also connect to SanchoNet, the testnet environment where CIP-1694's ideas are tested.

Intersect initially evaluated multiple tools to support DReps. GovTool was initially prioritized for its open APIs, but [Ekklesia](#) was subsequently implemented as an early signalling tool to “temperature check” the DRep sentiments ahead of on-chain governance actions. Ekklesia does not host a formal vote, but rather a critical indicator to aid in building consensus. Only proposals that surpassed 50% support in Ekklesia, at the snapshot, were eligible for inclusion in the 2025 budget info action. You can review all the proposals on Ekklesia.

Several proposals, including Intersect's, were updated following feedback from DReps but once proposals are submitted as part of a budget info action on-chain, they are immutable. The budget info action is presented to DReps for an on-chain vote. The budget process and tooling caused some confusion and frustration, an obvious area for improvement for the future. There have been calls to abolish the off-chain debates, and move to on-chain info actions followed by treasury withdrawals. Some suggested making immutable metadata for governance actions a requirement in the Cardano Constitution, highlighting a desire to enhance transparency and

permanence in on-chain decisions.

The governance tools working group has begun decentralizing ownership and maintaining the GovTool and constitutional committee portal. Community members are invited to participate in feature development, with discussions ongoing in the [wg-governance-tools](#) Discord channel.

The Cardano Foundation released a [voting tool](#) at voting.cardanofoundation.org where other tools built by the community are also listed.

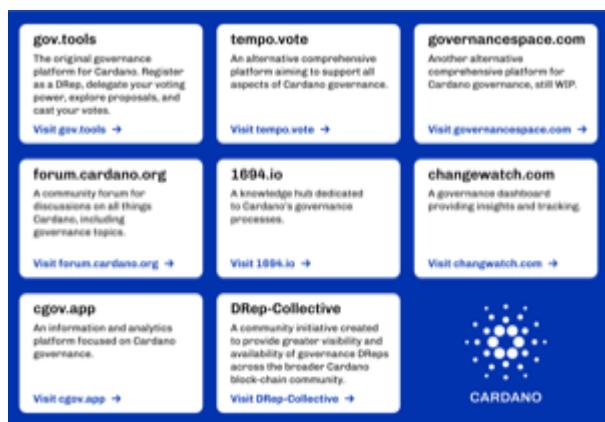


Figure 5.8: Governance tools

5.12. From theory to practice

The recent series of Cardano upgrades is named after Phillip Chang, who passed away in 2022, in honor of his contribution to the early design and concepts described in CIP-1694. The Chang upgrade marked a significant moment for Cardano, representing the culmination of years of dedicated development and community involvement. Extensive testing on SanchoNet and valuable feedback from community workshops paved the way for this critical step.

From Basho to Voltaire: a self-sustaining future

With the Chang upgrade, Cardano transitioned from the Basho development phase to Voltaire. This upgrade series unlocked minimum viable on-chain governance as outlined in CIP-1694, empowering the

community through a self-sustaining blockchain model that sets a new standard for the industry.

The upgrade unfolded in two stages:

- **Chang upgrade:** On September 1, 2024, this initial upgrade introduced core governance functionalities to Cardano, initiating the technical bootstrapping phase as defined in CIP-1694. This took Cardano into the Conway ledger era and officially heralded the start of Voltaire.
- **Plomin upgrade:** Originally named Chang Upgrade 2, the second phase was renamed to the Plomin Upgrade in memory of Matthew Plomin. Matthew was the pioneer and visionary behind Moneta and the USDM stablecoin, who sadly passed away in November 2024. The Plomin upgrade unlocked the full potential of on-chain governance, enabling DRep participation and treasury withdrawal capabilities. This marked the completion of the technical bootstrapping phase.

Cardano's on-chain governance relies on a core document: the ratified constitution. This document, approved through the new governance features, establishes the fundamental rules and principles that guide Cardano's operation.

Technical guardrails for stability The Intersect governance parameters working group shared their [report and recommendations](#) on the initial settings to be included in the technical guardrails as Cardano upgraded to Chang.

To ensure adherence to the constitution, a smart contract acts as the technical guardrail. This contract translates key constitutional provisions into code, wherever possible. For example, it might define acceptable ranges for parameters or treasury withdrawals so the blockchain will automatically reject any governance actions that violate these guardrails, preventing actions deemed unconstitutional. This adds an extra layer of security and stability to Cardano's governance process.

"I see the constitution as a living document, evolving with the Cardano community. Cardano shines as a model of strong blockchain governance" – [Frederik Gregaard, the Cardano Foundation CEO](#)

The Chang upgrade followed a similar deployment strategy to the Vasil upgrade. The final decision to initiate the upgrade was based on three key factors:

- technical stability: no critical issues were identified within core components (i.e., ledger, node, consensus, and CLI)
- performance optimization: benchmarking and analysis ensured acceptable performance and cost implications
- community readiness: sufficient communication and preparation time was provided to SPOs, DApp developers, and the broader Cardano community.

This measured approach, explained in more detail in the [documentation](#), ensured a smooth transition for all stakeholders as Cardano embraces its future of decentralized governance.

5.13. Journey to ratification

The interim constitution, drafted early in 2024 along with the technical guardrails, was made available for the community to read on the [Constitution Committee Portal](#).

The interim Constitutional Committee (ICC), the body that upholds the interim constitution and votes on the first on-chain governance actions, was formed. The community voted for three representatives to sit alongside pioneer entities IOG, EMURGO, and the Cardano Foundation, and Intersect. The [Cardano Atlantic Council](#), [Cardano Japan](#), and [Eastern Cardano Council](#) were duly elected to this responsible position.

The constitution can only claim legitimacy with feedback from the community, and so the first of 63 Constitutional workshops across 50

countries was hosted by Nicolas Cerny, Governance Lead for the Cardano Foundation, in Berlin in July 2024.

The feedback was collated and 128 delegates (64 voting, 64 traveling alternates) were chosen to attend the constitutional convention in Buenos Aires, Argentina and Nairobi, Kenya, in early December 2024.

After two days of debate and speeches, the constitution was approved by 95% of delegates. The proposed [constitution](#) featured a refined preamble, ten guiding tenets, a framework for the ecosystem budget, codes of conduct, compensation guidelines, clarity and safeguards for the Constitution Committee and an amendment process. While passing the constitution was no doubt a milestone, many delegates spotted areas requiring further clarity and scope.

The constitution was officially signed on the third day of the convention, and so the process of ratification by an on-chain vote by the broader community of ada holders began. This interim phase allowed the community to gain practical experience with on-chain governance, stress testing various tools and processes.

The second and final stage of the Chang upgrade, the Plomin hard fork occurred on January 29, 2025. Cardano's move to the Voltaire development phase was now complete and Cardano was henceforth to be fully governed by the community.

The ratified constitution was [enacted](#) on February 23, 2025. It formalized the roles of ada holders, DReps, SPOs, and the Constitutional Committee but importantly, remains a 'living document' not set in stone.

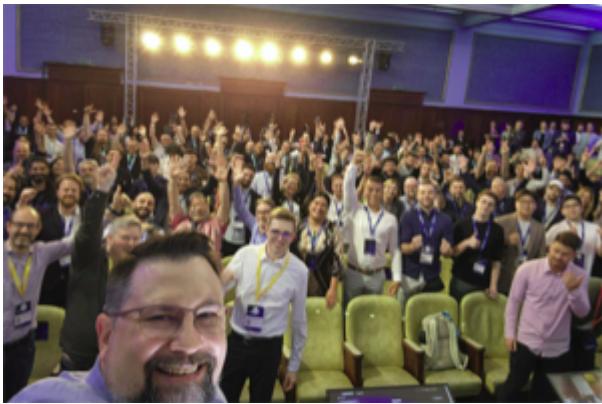


Figure 5.9: Chair of Proceedings in Buenos Aires, Lloyd Duhon, marking the historic moment with a group selfie at the Buenos Aires **Liquid democracy**

As a result of the Plomin hard fork, you can continue accumulating staking rewards but you cannot withdraw them unless you delegate your ada's voting rights to a DRep or a predefined voting option. The community made this decision way back at the first CIP-1694 workshop in Colorado in March 2023. Similar to the "liquid staking" concept, the staked ada never leaves your wallet, "liquid democracy" means you can change your DRep every five days without locking your ada tokens. There is no additional cost or risk. With just a few clicks, you can participate in governance on Cardano.

The first governance action submitted of the new era was a proposal to change the treasury_growth_rate from 20% to 10%. This was the first community-raised parameter change using the new features. Although it didn't pass, the topic sparked lively debate and featured on this episode of [NerdOut - Let's Have a TEA Party](#).

The Net Change Limit

The net-change-limit guards the sustainability and stability of Cardano's treasury. It sets the constitutional and democratic mandate for the amount of ada that can be withdrawn from the Treasury over an annual period. Like most things with Cardano governance, the name itself even prompted questions and [debate](#). Some feel it should be called an 'annual withdrawal

limit' as the treasury replenishes itself constantly. You can read more about the reasoning for such a limit on the [Cardano Forum](#) which hosts discussions like [Budget Info Actions vs. Net Change Limit: Where the Cardano Constitution Draws the Line](#).

You can think of the Net-change-limit as a constitutional mechanism, like a financial safeguard controlling how much ada can be withdrawn from the Treasury within a set period. The community evaluated several proposals, before the 'A350m Net Change Limit' [governance action](#) passed with strong community support.

The *Cardano Constitution* requires that a Net Change Limit be in place before a budget proposal can be passed. With consensus reached, the next phase could commence, garnering much debate as treasury-funded proposals could now be submitted and considered through on-chain Budget Info Actions.

The role of Intersect committees needed to evolve to align with Cardano's new budget process. Some of the key changes introduced were:

- DReps were now the final decision makers for funding, with committees focusing more on roadmap development and governance oversight
- committees no longer submit block budgets
- focus on transparency, enabled by multi-signature smart contracts, so funding decisions and treasury withdrawals can be verified.

Budget process

Cardano's budget process was 'recalibrated' several times throughout late 2024 and early 2025 as the community provided robust feedback. Attempting to meet everyone's expectations and cater for many projects eager for funding three years into a bear market cycle is no small feat. Intersect made the necessary adjustments, eventually achieving consensus on funding priorities via the [product roadmap](#).

Anyone in the community could seek treasury funding. The DReps and the Constitutional Committee would ultimately vote to decide how and when Treasury funds are allocated. Intersect's role was to facilitate, and provide administrative oversight.

Intersect's own budget proposal

On the back of lengthy community feedback, there was a leadership reset, a refined scope and a revised Treasury proposal seeking less ada. The new leadership pledged to return Intersect to its founding mission — to enable decentralized coordination, shepherding, rather than directing, the network. The [full proposal](#) on Govtool outlines deliverables including:

- 2026 product roadmap and budget submission
- Intersect and committee elections
- expansion of regional hubs
- Open Source Fellowship launch
- ongoing support for working groups, events, and governance tooling.

The final temperature check in Ekklesia produced 40 proposals that met the threshold, with 39 of those requested Intersect as their Administrator. You can read all of the proposals on the Cardano Forum, starting with the [Cardano Budget 2025 Overview](#).

On June 14, 2025, an X post by @gomadrep confirmed the [A275M Budget Info Action \(BIA\)](#) had passed. 63.66% of DReps, 93.26% of SPOs and 6/7 of the Cardano Constitution approved.

A non-binding poll was also proposed on Ekklesia asking DReps to signal their preferred approach for submitting Treasury Withdrawals. Ultimately the community voted for Option E: 39 individual Treasury Withdrawals, one per proposal.

Smart Contracts

Smart contracts can leverage the EUTXO model to enforce deterministic behavior from a limited set of actors, with no single entity acting unilaterally. The preference is for a predictable “can’t be evil” model instead of a hopeful “don’t be evil” setup.

The Cardano constitution mandates that treasury withdrawals include independent audits, on-chain oversight, and separate, auditable accounts that delegate to the auto-abstain voting option (not an SPO).

To align with the Cardano Constitution, there needs to be:

- a means to hold Treasury ada securely and transparently, auditable at any time
- a mechanism for releasing funds to vendors after set milestones
- safeguards to ensure no single entity can act alone when moving funds
- independent verification from any oversight body to allay concerns and build trust.

As part of the changes to the budget process, a Smart Contract working group was set up under the Budget committee to oversee the automation of treasury fund management with smart contracts. The solution was delivered through Intersect by Input Output | Global, SundaeLabs, and Xerberus. The implementation of smart contracts to automate treasury fund management satisfies Article 4, section 2 of the Cardano Constitution:

“Development of Cardano Blockchain ecosystem budgets and the administration of such budgets shall utilize, to the extent possible and beneficial, smart contracts and other blockchain based tools to facilitate decision-making and ensure transparency.”

Two types of smart contracts

SundaeLabs's [live demonstration](#), where viewers were encouraged to follow the activity at an [address](#), showcased the two key smart contract types used to manage operations: **Treasury Contracts** act as reserve

accounts for ada withdrawn from the Cardano Treasury. These funds can only move funds after strict conditions are met, such as a multi-signature from different parties on the Oversight Committee. **Vendor Contracts** inherit their permissions from their parent Treasury Contract, releasing funds to Vendors on completion of set milestones.

As Intersect is the only Administrator available at time of writing, it initiates all actions, with an Oversight Committee scrutinizing each step. Intersect applies various best practices to ensure the integrity of governance actions. For example, it published the verification key used to sign all governance metadata it authors. This is an important safeguard which prevents misuse and ensures accountability. This is enabled by governance metadata standards defined in CIP-100 and CIP-108.

Oversight Committee

The Oversight Committee is composed of independent third-parties who provide an additional verification layer within the smart contract framework. Members cross-check the accuracy and consistency of key actions. The inaugural members are:

- Sundae Labs
- Xerberus
- NMKR
- Dquadrant
- The Cardano Foundation

Constitutional Committee

The Constitutional Committee upholds the principles and rules of the Cardano constitution. The interim Constitutional Committee was set up as a temporary body in September 2024. It was structured as a mix of appointed members, with seats held by Intersect, the founding entities, and three elected seats.

With ICC's fixed term expiring in September 2025 so the process to elect a new committee was initiated in November 2024, when a working group was formed. The new CC's term would see the election's top three winners serving two years, and the remaining four serving one year.

Each ICC member agreed not to run for the new committee, allowing for a fresh start. As befits a permissionless, public blockchain, any ada holder could run for the CC. That said, each candidate's credentials were scrutinized in a battery of ask me anythings (AMAs), roundtables and X spaces.

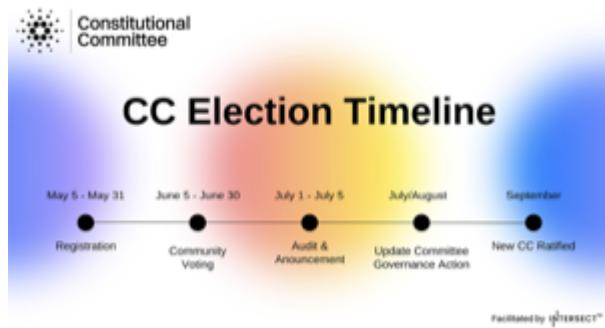


Figure 5.10: Constitutional Committee Election Timeline

The seven members elected are:

- Cardano Atlantic Council
- Tingvard
- Eastern Cardano Council
- KTorZ
- Ace Alliance
- Cardano Japan Council
- Phil_uplc

The successors began a comprehensive training program in preparation for their roles, while all on-chain votes were audited by [DQuadrant](#). The final step is the ratification of the new CC on-chain via an 'update committee' governance action when DReps and SPOs will vote in

September 2025.

This timeline means the new committee was not required to vote on the 39 individual treasury withdrawal proposals. It allowed time for training and a seamless transition before the current Interim Constitutional Committee's (ICC) mandate expires.

5.14. PRAGMA

From the outset of the Voltaire development phase, it was always expected, and some feel necessary, to have multiple MBOs. [PRAGMA](#) was announced on the eve of the inaugural BuidlFest meetup in Toulouse, France. PRAGMA is a member-based, not-for-profit, open-source association for blockchain software projects. Initially, it will be made up of familiar faces to the Cardano developer ecosystem: Blink Labs, Cardano Foundation, dcSpark, SundaeLabs, and TxPipe, but will expand to incorporate more projects and members in future.

PRAGMA will not compete with Intersect, but run as a complementary effort. While the mission of Intersect is broader, PRAGMA is focused solely on open-source software development with two key projects for now: Aiken, the popular programming language for on-chain smart contracts on Cardano, and Amaru, a Rust node client for Cardano. It's important to note that anyone can submit a budget proposal. For example, Amaru's [budget proposal](#) for 2025 was passed and became the first Treasury Withdrawal [governance action](#) to be executed. For Cardano to thrive, PRAGMA and Intersect need to work together to deliver what is best for the ecosystem.

5.15. Looking forward

Decentralized governance requires continuous iteration and community participation, which can pose coordination challenges. Without central leadership, decentralized governance risks deadlock or disputes. Critics argue that decentralized models prohibit fast decision-making and true

accountability due to complex processes. Cardano has sought to mitigate these risks by implementing CIP-1694 and maintaining technical oversight through qualified third parties. However, the long-term viability of this model depends largely on participation from ada holders.

A balance must be struck so rigorous analysis doesn't lead to procrastination. With competitor blockchains spending vast sums at the drop of a hat, the process needs to be more agile, making funds more accessible so Cardano can pivot easily and compete in a volatile industry. This can be achieved while still adopting prudent investment strategies ensuring fiscal responsibility.

Cardano's treasury holds approximately 1.7 billion ADA, so ensuring its long-term sustainability is critical. Some have looked to existing structures, such as Sovereign Wealth Funds, drawing inspiration from the [Santiago Principles](#). However, the Cardano treasury's ada-first policy lacks options to diversify assets. Adding Cardano native tokens, stablecoins, and other assets would mitigate risks of market volatility. Intersect is working on adopting Cardano's native stablecoins as part of the evolving strategy. USDM (@USDMOfficial) and USDA (@AnzensOfficial) are obvious candidates to bootstrap more practical funding mechanisms across the ecosystem.

Staying informed

Given the many moving parts within Intersect and the rapid pace of updates, you can stay informed following the weekly development [newsletter](#). For a broader perspective, the Cardano Foundation regularly [blog](#) outlining their rationale on governance.

[1] Kaidalov, Kovalchuk, Nastenko, Rodinko, Shevtsov, Oliynykov (2017), 'A proposal for an Ethereum Classic Treasury System', iohk.io/en/research/library/papers/a-proposal-for-an-ethereum-classic-treasury-system/

[2] Zhang, Oliynykov and Balogun (2019), 'A Treasury System for Cryptocurrencies: Enabling Better Collaborative Intelligence', eprint.iacr.org/2018/435.pdf

[3] Kiayias, Lazaros (2022), 'SoK: Blockchain Governance', arxiv.org/pdf/2201.07188.pdf

[4] Ciampi, Karayannidis, Kiayias and Zindros (2020), 'Updatable Blockchains', iohk.io/en/research/library/papers/updatable-blockchains/

[5] Charles Hoskinson: Crypto regulations & policy, Importance of stablecoins & the future of Cardano, youtu.be/uEV8tQ6z87k?si=iVazdagl5JWZez3q&t=1983

[6] IO ScotFest Keynote with Charles Hoskinson, youtu.be/tbtkClr3Y3I

Chapter 6. Wallets in the world of Cardano

Chapter 7. 6.1 Fundamentals of crypto wallets

A wallet, also known as a crypto wallet, digital wallet, or cryptocurrency wallet, is like a digital version of a regular wallet. However, instead of holding cash, personal identification, or cards, it stores special digital keys that enable interaction with various blockchains, allowing users to manage digital assets such as cryptocurrencies, tokens, or non-fungible tokens (NFTs).

These keys allow you to access and manage your digital assets. Just like a key opens a safety box lock, you use these digital keys to do the same.

At its core, a wallet serves two primary purposes:

- **Secure storage of keys.** Your wallet securely stores your “keys,” which are unique, sophisticated digital codes known as “public keys” and “private keys.” The public key is like your email address; you share it with others to receive cryptocurrency. The private key, akin to your email password, must be kept secret. It’s what allows you to send or spend your cryptocurrency.
- **Transaction management.** A wallet enables you to send and receive digital currencies, such as ada, Cardano’s native currency. When you send cryptocurrency, you essentially sign off ownership to the receiver’s wallet. Your private key validates this transaction, acting as a digital signature that provides mathematical proof of your authorization.

It’s crucial to understand that a wallet doesn’t physically store your

cryptocurrency. Instead, it holds the cryptographic information required to access and transact with your digital assets on the blockchain, the underlying technology of cryptocurrencies. This makes your wallet a crucial tool for managing your digital assets securely and efficiently. In this section, you will learn the generalities of wallets before going into the details of how wallets work in Cardano.

7.1. 6.1.1 Types of wallets

In the blockchain realm, maintaining the security and accessibility of your digital assets is crucial. This is where the choice of a wallet comes into play. Crypto wallets can be broadly categorized into three types: *hardware*, *software*, and *paper*.

Each type offers a different balance of security and convenience, catering to various needs and preferences. Let's explore these three main types of wallets before diving into the specifics:

- **Hardware wallets.** Imagine a hardware wallet as a secure USB stick that stores your cryptocurrency and other digital assets offline. These physical devices are considered one of the safest options because they keep your private keys completely disconnected from the internet, reducing the risk of hacking. They are ideal for long-term investors who want to keep their assets secure for an extended period.
- **Software wallets.** Software wallets are applications that can be installed on your computer or smartphone. They are more convenient for daily transactions, striking a balance between security and accessibility. Since they are connected to the internet, they are considered “hot wallets.” While they are generally secure, they are more vulnerable to online threats than hardware wallets. Software wallets are ideal for active trading users who require frequent access to cryptocurrencies.
- **Paper wallets.** A paper wallet is a physical document that contains a

public address for receiving cryptocurrencies and a private key for spending or transferring them. It's a piece of paper with public and private keys, printed as QR codes.. Paper wallets offer a highly secure method for storing cryptocurrencies since they are entirely offline. However, they can be easily damaged or lost, and they are not as convenient for quick or frequent transactions.

It's also important to note that paper and hardware wallets are also known as "cold wallets" or "cold storage" wallets, as they always keep private keys offline.

Each type of wallet has unique features, advantages, and considerations. Your choice depends on your specific needs, such as the level of security you require, how frequently you plan to access your cryptocurrency, and your comfort level with technology. The following sections provide an in-depth exploration of each type, helping you determine which wallet best suits your journey into the world of Cardano and the blockchain.

7.2. 6.1.2 Hardware wallets

Hardware wallets are physical devices that store users' private keys securely offline. They connect to a computer via USB ports and are designed to provide the highest level of security for digital assets. By keeping private keys offline, hardware wallets protect cryptocurrencies and digital assets from online hacking attempts, malware, and other cyber threats.

These wallets strike an excellent balance between security and usability, making them particularly well-suited for long-term investors or individuals with substantial digital asset holdings. While considerations such as cost and potential risks of physical loss exist, the security benefits of keeping private keys offline often outweigh these drawbacks.

Choosing the right hardware wallet depends on your specific needs, including the cryptocurrencies you hold, your budget, and how you plan

to use your digital assets.

Let's examine the pros and cons of using hardware wallets and some common models available on the market.

Pros of hardware wallets

- **Enhanced security.** The primary advantage of hardware wallets is their ability to store private keys offline, thereby reducing their vulnerability to online hacking attacks. Transactions are signed within the device, meaning the private keys never leave the device.
- **Portability.** Despite their robust security features, hardware wallets are small and portable, allowing you to manage your cryptocurrencies and digital assets from anywhere with access to a computer or smartphone.
- **Multi-currency support.** Most hardware wallets are designed to support multiple cryptocurrencies and blockchains, making them a versatile choice for users holding a variety of digital assets.
- **User-friendly.** Modern hardware wallets feature user-friendly interfaces, making it straightforward for users, including newcomers, to manage their holdings.

Cons of hardware wallets

- **Cost.** Unlike software or paper wallets, hardware wallets come with a price tag. The cost can be a barrier for people just starting with cryptocurrencies or those with a small amount of digital assets.
- **Physical damage or loss.** While they are less prone to online threats, hardware wallets are physical devices that can be damaged, lost, or stolen. Users must take precautions to protect their devices and back up their recovery phrases.
- **Less convenient for quick transactions.** For users who frequently trade or access their cryptocurrencies or digital assets, hardware

wallets may be less convenient than software wallets due to the need to connect the device to a computer or mobile device for transactions.

Common hardware wallets in the market

This section presents a list of hardware wallets to illustrate some available options. Please note that the authors are not involved in the development of these wallets and do not endorse any specific products.

- **Ledger.** This is one of the most popular hardware wallets, known for its security and support for many cryptocurrencies. You can learn more about it on its [website](#).
- **Trezor.** Trezor is another leading brand in the hardware wallet space, offering high-security levels and an intuitive interface. You can learn more about it on its [website](#).
- **Keystone.** This hardware wallet is open-source and provides an extra layer of security by keeping your private keys entirely offline. You can learn more about it on its [website](#).

For a more extensive list of hardware wallets, you can review [this compilation](#) provided by Interpol for digital forensics detection.

7.3. 6.1.3 Software wallets

Software wallets store cryptocurrency private keys on internet-connected devices like computers or smartphones. They offer convenient access to digital assets, facilitating easy transactions through mobile, desktop applications, or web browser extensions.

Let's explore the pros and cons of software wallets, along with a review of some popular options.

Pros of software wallets

- **Accessible.** Software wallets provide easy access, allowing users to manage cryptocurrencies and other digital assets, perform transactions,

and check balances quickly from any location with internet connectivity.

- **User-friendly.** These wallets typically feature intuitive interfaces, making it straightforward for users to navigate, conduct transactions, and monitor their holdings, catering to beginners and experienced users.
- **Cost-effective.** Unlike hardware wallets, software wallets are generally free to download and use, making them appealing to those new to cryptocurrencies or with smaller holdings.
- **Exchanges integration and other features.** Many software wallets offer additional features such as integrated exchanges, staking, and support for multiple cryptocurrencies, enhancing their functionality beyond mere storage.

Cons of software wallets

- **Security risks.** An internet connection makes software wallets more vulnerable to online threats, including hacking, phishing attacks, and malware. Users must ensure their devices and networks are secure..
- **Device dependence.** If the device hosting the wallet is damaged, lost, or compromised, accessing the wallet can be challenging without proper backup and recovery measures.
- **Vulnerability to malware.** Devices infected with malware pose a significant risk to software wallets, potentially leading to unauthorized access and theft of digital assets.

Common software wallets in the market

Please note that the software wallet examples provided below are for informational purposes only and do not imply any endorsement or recommendation of a specific option.

- **MetaMask.** Primarily known for its integration with the Ethereum

network, MetaMask is a popular software wallet that also supports a variety of Ethereum-compatible tokens. It's available as a browser extension and a mobile app, providing flexibility in how users access their digital assets. You can learn more about it on its [website](#).

- **Exodus.** A multi-currency wallet known for its attractive user interface and ease of use. Exodus supports various cryptocurrencies, including Cardano, and offers features like an integrated exchange and live charts. Exodus is available as a mobile app, a Web3 wallet browser extension, and a desktop application for most popular operating systems. You can learn more about it on its [website](#).
- **Coinomi.** A multi-currency wallet available on desktop and mobile platforms; it supports many cryptocurrencies and allows purchasing cryptocurrency through integration with some partners. It's known for its robust privacy features and its ability to swap coins within the wallet. You can learn more about it on its [website](#).
- **Atomic Wallet.** An open-source, multi-currency wallet that supports over 300 cryptocurrencies. It offers features such as atomic swaps, a built-in exchange, strong encryption, and robust security measures. It's available as a mobile app and a desktop application for major operating systems. You can learn more about it on its [website](#).

These software wallets cater to a broad spectrum of needs and preferences, from casual users to more privacy-focused individuals. When choosing a software wallet, consider factors like supported currencies, security features, ease of use, and any specific functionalities that align with your cryptocurrency usage habits.

7.4. 6.1.4 Paper wallets

Paper wallets are a form of cold storage for cryptocurrencies and digital assets, as they remain offline and are not vulnerable to online hacking attacks. A paper wallet is a physical document containing all the necessary data for accessing and managing digital assets. This document typically

includes a public address (used by others to send funds) and a private key, which allows the owner to access and transfer their funds.

Let's explore the advantages and disadvantages of paper wallets, along with some standard practices in the blockchain realm for effective management.

Pros of paper wallets

- **High security.** Since they are offline, paper wallets are immune to online hacking attacks, malware, and other digital threats, making them one of the most secure methods of storing cryptocurrency.
- **Ownership control.** With a paper wallet, you have complete control over your private keys and, consequently, your cryptocurrency and digital assets. There's no reliance on third-party services or risks associated with online wallet providers.
- **Cost-effectiveness.** Creating a paper wallet is free, requiring only paper and a printer, unlike hardware wallets, which can be expensive.

Cons of paper wallets

- **User-friendliness.** Paper wallets can be less user-friendly, particularly for individuals unfamiliar with blockchain technology. The process of transferring funds to and from a paper wallet is more cumbersome than using software or hardware wallets.
- **Durability and loss risk.** Paper is prone to wear, tear, damage, and loss. If the paper wallet is damaged or lost without a backup, the funds stored on it become irretrievable.
- **Lack of flexibility.** Paper wallets offer a different level of convenience compared to other wallet types. They're best suited for long-term storage rather than for frequent trading or spending.

Next, we'll review some standard methods for generating a paper wallet.

Standard practices to create and manage a paper wallet

The following practices are *general* recommendations intended for informational purposes only.

- **Using a software or hardware wallet.** A common practice for generating a paper wallet is to create a pair of public and private keys using secure software or hardware wallets. After these keys are generated, the public key can be printed on paper for future reference. If needed, a QR code can be created to represent each key.
- **Using a trusted paper wallet generator.** Some websites, such as BitAddress.org, allow the creation of paper wallets. However, some of these websites have been flagged as insecure and potentially malicious. When using this mechanism, ensure you use a secure, trusted website to avoid malicious code.
- **Offline generation.** For enhanced security, the paper wallet should be generated offline on a computer that is not connected to the internet and has never been connected, to prevent exposure to online threats.
- **Secure printing.** The wallet should be printed with an offline printer to avoid any digital traces of your private keys. Using a printer that doesn't retain memory or a brand-new printer is advisable.
- **Safe storage.** Once printed, the paper wallet should be stored in a safe, waterproof, and fireproof location. Some users opt for safes or safety deposit boxes. Creating multiple copies and storing them in different secure locations is also wise.
- **Avoid sharing.** Never share your paper wallet's private key or QR code with anyone. Exposure equals potential theft.
- **Transferring funds.** When you want to move digital assets in a paper wallet, you'll typically need to import your private key into a software wallet to conduct transactions.

When created and stored correctly, paper wallets offer a high-security option for storing cryptocurrencies and digital assets. They are particularly suited for long-term investors who wish to keep their digital

assets securely without the need for regular access or trading. However, the potential risks of physical damage, loss, and the technical challenges of using paper wallets should be carefully considered.

7.5. 6.1.5 Public and private keys in the context of wallets

Understanding the role of public and private keys is fundamental when using a wallet to manage digital assets. Public and private keys are like the username and password to your online bank account, but with some particularities that we will discuss.

Public keys

Think of the public key as your home address. Just like you can share your address with friends so they can send you letters or packages, you can share your public key with others to receive cryptocurrency or any other digital assets. It's a string of letters and numbers derived from your private key, but it's safe to share with others because, while it can receive funds, it can't be used to withdraw any.

Imagine selling lemonade and asking people to leave money in a locked box at your front door (your public key). They can deposit money, but only you can unlock the box to retrieve it because you have the key – your private key..

The wallet private key is like the key to that locked box. Like a public key, it consists of a string of letters and numbers, but should always be kept secret since it allows you to access and control your assets. With your private key, you can send your cryptocurrency and digital assets to others, sign transactions to prove you own the assets, and even access your funds if you switch to a new wallet.

Following the example of selling lemonade, you only hold the key to the padlock, allowing you to open it and retrieve the money left by customers.

If someone else gains access of your key, they can take all the money, just as obtaining your private key enables unauthorized access to your cryptocurrency.

Blockchainkeys facilitate transaction processing and approval. The public key serves as the address for sending or receiving funds, while the private key is the secret that controls access to personal assets. Securing your private key is crucial because anyone with it can access your assets. Just like you wouldn't share the key to your safe with strangers, you should never share your private key with anyone.

7.6. 6.1.6 Wallet addresses

A wallet address is similar to an email address or a home address – it's a unique identifier used to receive funds. When someone sends you cryptocurrency or other digital assets, they send it to your wallet address.

A wallet address is a string of letters and numbers that represents the destination for a blockchain transaction, involving the transfer of assets from one address to another. It's generated based on the public key and, depending on the blockchain, goes through a series of cryptographic transformations to create a shorter, more user-friendly address.

These are examples of what wallet addresses look like for Cardano, Ethereum, and Bitcoin. Each of these addresses is a string of characters that includes numbers and letters, both uppercase and lowercase, specific to the blockchain they belong to:

Cardano (ADA) address:

```
addr1q9d7n2g0s8eqrakj7k65zm4u6gfvg4t5u9yv15cxxxxxxxxsyqgp7ytwx8x5a2hxu2z  
ddm09fjq9usu2kyxj6h2c429sxxfcke
```

Cardano addresses typically start with the *addr1* prefix and are longer than those of Ethereum and Bitcoin, reflecting the network's unique

addressing scheme.

The length of a Cardano address can vary, primarily because it utilizes the *Bech32* address format, which can be adjusted depending on the specific use case and the data it encapsulates. Typically, a Cardano address is about 58 to 103 characters long.

Ethereum (ETH) address:

```
0x4e6ff4719a579De0b461C082eD1D7A1898617A3
```

Ethereum addresses typically begin with *0x*, a common prefix denoting hexadecimal encoding, and are 42 characters long, including the prefix.

Bitcoin (BTC) address:

```
1BoatSLRHtKNngkdXEeobR76b53LETtpyT
```

Bitcoin addresses can start with *1*, *3*, or *bc1* for different address formats; the example shows one of the most common types, beginning with *1*.

Please note that these addresses are provided as examples only and should not be used for actual transactions. Each cryptocurrency wallet generates a unique set of addresses for an individual user, ensuring the security and privacy of the assets.

7.7. 6.1.7 Creating a wallet address

The process of creating a wallet address may vary depending on the blockchain, but the following steps are generally followed.

- 1. Key pair creation.** A private key is first generated using cryptographic algorithms. This private key is a random, long string of numbers and letters that's virtually impossible to guess. Next, a public key is derived from this private key using another set of cryptographic rules.

2. **Hashing.** The public key is then passed through a hashing algorithm – a cryptographic function that converts data into a fixed-size string of characters. This process helps enhance security and privacy.
3. **Formatting.** After hashing, the result undergoes additional transformations, including the addition of a network identifier (which helps differentiate between different cryptocurrency addresses) and a checksum (which helps detect errors in the address). The final output is your wallet address.

Once it's created, a wallet address has a set of common characteristics regardless of the blockchain:

- **Uniqueness.** Each wallet address is unique, ensuring that funds sent to the address reach the correct recipient.
- **Anonymity.** While wallet addresses are publicly visible on the blockchain, they don't reveal the owner's identity directly.
- **Single use.** For enhanced security and privacy, it is often recommended that you use a new address for each transaction. Many modern wallets automatically generate a new address after each transaction.

A wallet address is a critical component of cryptocurrency transactions, serving as a pointer to where funds should be sent on the blockchain. In short, it's derived from the public key through a series of cryptographic processes, ensuring security and facilitating the seamless transfer of digital assets.

The following section explores wallets available in the Cardano ecosystem.

Chapter 8. 6.2 Wallets in the Cardano Ecosystem

Cardano wallets are designed to store, send, and receive ada, the native currency of the Cardano blockchain. These wallets are designed to support Cardano's unique features, offering users a secure way to manage their assets. Let's break down the specifics of Cardano wallets in a simple and understandable way.

8.1. 6.2.1 Cardano wallets are designed for ada

Firstly, it's crucial to understand that Cardano wallets are specifically designed for the ada cryptocurrency. Even though you can have a wallet that allows you to store assets from different blockchains, it's important to highlight that, just like you might have a specific wallet or pocket for coins or cards, Cardano wallets are made to handle the particular requirements and features of ada and the Cardano blockchain.

8.2. 6.2.2 Types of Cardano wallets

The Cardano ecosystem has two types of wallets: full-node and light wallets. Let's examine them briefly.

Full-node wallets

These wallets download the entire Cardano blockchain to your device, offering high security and privacy since they don't rely on a third party to fetch blockchain data. Using a full-node wallet is like having a detailed map of an entire city. You can see every street and building, or, in this case, every transaction on the Cardano network.

A full-node wallet performs several key functions:

- *Transaction verification.* It verifies the validity of each transaction against the blockchain's consensus rules.
- *Blockchain synchronization.* It downloads and synchronizes with the entire history of the Cardano blockchain, ensuring it has the most up-to-date information.
- *Network participation.* It helps the network by relaying transactions and blocks to other nodes.
- *Security and privacy.* By not relying on external sources for transaction verification, it offers its users increased security and privacy.

An example of a full-node wallet is the [Daedalus wallet](#). It allows users to fully participate in the network, including sending and receiving transactions, staking ada for rewards, and interacting with smart contracts, such as those written in Plutus, Aiken, or Marlowe programming languages.

Light wallets

A light wallet, also known as a lightweight wallet, does not download the entire blockchain. Instead, it relies on other nodes in the network to provide the necessary information to manage the user's funds and assets. Light wallets are much faster to set up and require less storage space than full-node wallets.

Using a light wallet is similar to using a map app on your phone. You don't need to download every map detail, but you can still find your way around. Light wallets, such as Lace or Yoroi, allow users to interact with smart contracts without the need to run a full node. They can create transactions, sign them, and broadcast them to the network for execution. These wallets are handy for everyday users who want to engage with blockchain transactions without the technical overhead of maintaining a full copy of the Cardano blockchain.

The wallets in the Cardano ecosystem are constantly evolving, so it's challenging to list them in this book. For a current list of the wallets available in Cardano, you can look at community websites like [BuiltOnCardano](#) or [the Cardano documentation page about wallets](#).

8.3. 6.2.3 Staking and delegation

Cardano wallets offer a unique feature called staking and delegation. By staking your ada within the wallet, you can participate in the network's operation and earn rewards. Think of it like putting your money in a savings account where it earns interest, but in this case, you're helping to secure the network and validate transactions.

Staking and delegation are two key concepts in proof-of-stake (PoS) blockchains, such as Cardano, which utilizes a variation called Ouroboros.

Staking refers to the process of holding funds in a wallet to support the operations of a blockchain network. Essentially, it involves locking cryptocurrencies to receive rewards. Note that Cardano offers its users liquid staking, allowing them to spend their funds at any time. In PoS blockchains, staking contributes to the network's security and governance because the staked coins are used to select validators who confirm transactions and create new blocks.

Delegation, on the other hand, is the act of entrusting your staking power to a stake pool, a server node that maintains the network and processes transactions. Delegation allows wallet holders to participate in the staking process without running a node themselves. By delegating their stake, users can earn rewards proportional to the amount of cryptocurrency they delegate.

Staking and delegation will be discussed in detail in later chapters.

8.4. 6.2.4 Security features

Security is a top priority for Cardano wallets. They are equipped with

various security features to ensure your ada and other assets are safe from unauthorized access. Here are some of the Cardano wallets' vital security features:

- *Encryption.* Wallets use strong encryption to protect private keys and other sensitive data stored on the user's device.
- *Seed phrases.* Wallets generate a recovery seed phrase, typically 12 or 24 words long, which can be used to restore the wallet and its contents on another device if the original device is lost or damaged.
- *Password protection.* Users can set a password to access the wallet application, adding an additional layer of security.
- *Cold storage:* Some wallets offer the ability to store funds offline, which is known as cold storage. This dramatically reduces the risk of online hacking attempts.
- *Multi-signature support.* Specific wallets support multi-signature configurations, which require multiple parties to sign a transaction before it can be executed. This enhances security for larger funds or organizational use.
- *Hardware wallet integration.* Many Cardano wallets can integrate with hardware wallets, such as Ledger or Trezor, which store the user's private keys in a secure hardware device, making them less susceptible to computer viruses and malware.
- *Transaction confirmation.* Wallets require user confirmation for transactions, ensuring that funds are not moved without explicit user permission.
- *Open source.* Many Cardano wallets are open source, allowing the community and security experts to review the code for potential vulnerabilities.
- *Regular updates.* Wallet developers regularly update the software to address any security issues and add new features, keeping the wallet secure against evolving threats.

8.5. 6.2.5 Integration with DApps

Some Cardano wallets allow you to interact with decentralized applications (DApps) built on the Cardano blockchain. This feature allows for more versatile use of your ada, such as participating in decentralized finance (DeFi) platforms or trading non-fungible tokens (NFTs).

8.6. 6.2.6 Cardano wallets vs other blockchain wallets

Cardano wallets are designed specifically for the Cardano blockchain. They are tailored to interact with their unique features, such as the Ouroboros proof-of-stake consensus mechanism, native tokens, and smart contracts written in Plutus, Marlowe, or other supported programming languages.

Next, we present some notable distinctions between Cardano wallets and wallets for other blockchains.

- *Consensus mechanism compatibility.* Cardano wallets support staking and delegation using the Ouroboros protocol, unlike proof-of-work blockchains like Bitcoin.
- *Extended UTXO model.* Cardano employs an extended unspent transaction output (EUTXO) model, a variation of the UTXO model used by Bitcoin. Cardano wallets are designed to handle the additional complexity and capabilities of the EUTXO model, including the ability to process multiple assets in a single transaction.
- *Native token support.* Unlike other wallets that require smart contracts to handle tokens, Cardano wallets support native tokens, which do not require smart contracts, thereby reducing the cost and complexity of token transactions.
- *Smart contract integration.* Cardano wallets can interact with smart contracts on the Cardano blockchain, including those written in Plutus or Marlowe, for example. This integration enables users to participate in complex financial contracts and decentralized applications (DApps).

- *Minimum ada requirement.* Transactions on the Cardano network require a minimum amount of ada to be included in the transaction to be valid. Cardano wallets automatically handle this requirement.
- *Network upgrades.* Cardano undergoes regular network upgrades, known as hard forks, which are seamlessly integrated into the wallets without disrupting the user experience.
- *Security features.* Many wallets share standard security features, such as encryption and seed phrases. Cardano wallets, however, often include additional measures tailored to the Cardano ecosystem, such as leveraging the Ouroboros protocol – the backbone of Cardano’s security and consensus mechanism.
- *Community and governance.* Cardano wallets may include features that allow users to participate in the Cardano community governance, such as voting on Project Catalyst proposals.

It's essential to note that, while there are differences, there are also many similarities between Cardano wallets and other blockchain wallets, including the basic functionality of sending and receiving funds, the use of public and private keys, and a strong emphasis on security.

Finally, not all wallets implement the complete set of features described above, but they provide a robust framework for securely managing ada. Remember, securing your recovery phrases and practicing safe wallet use is essential.

For a comprehensive list of wallets, please refer to [the wallets showcase in the Cardano Developers Portal](#).

8.7. 6.2.8 Exploring Cardano wallets

This section discusses some of the wallets designed explicitly for the Cardano ecosystem, providing an overview of the available options.

8.7.1. 6.2.8.1 Full-node wallets

Remember that when opened, a full-node wallet downloads the entire blockchain, requiring specific software and storage. Let's discuss Daedalus, the full-node Cardano wallet available on the market.

Daedalus is one of Cardano's official wallets developed by IOG. Daedalus is an open-source desktop software wallet for storing ada. As a full-node wallet, Daedalus downloads the entire Cardano blockchain, verifying every transaction to ensure maximum security.

The main features of Daedalus are:

- *Maximum security and completely trustless operation.* With full-node synchronization, you achieve maximum security and completely trustless operation, eliminating the need for centrally hosted third-party servers.
- *Unlimited accounting.* With Dedalus's hierarchical deterministic (HD) wallet implementation, you can manage any number of wallets. You can also have more control over how your funds are organized and use its robust backup and restoration features to recover your funds if necessary.
- *Availability for any desktop operating system.* Daedalus runs on Windows, macOS, and Linux, supporting all major desktop operating systems. It is built with web technologies on top of Electron, a battle-proven open-source platform for writing cross-platform desktop applications.

Please visit the Daedalus official website to download the latest version and verify the recommended hardware requirements. The [Daedalus documentation is available here](#).

Finally, you are encouraged to visit the Daedalus [code repository](#) if you want to inspect the code or contribute to its development.

8.7.2. 6.2.8.2 Light wallets

While full-node wallets keep a copy of the entire Cardano node, a light wallet does not need to download the complete history of blockchain records. Instead, it links to a website where the full blockchain is accessed, making it faster and easier to use. Light wallets are usually distributed as browser plugins or mobile applications.

This section explores some of the most prominent light wallets in the Cardano space. As this is not a tutorial, the section only offers an overview of key features. However, links and learning resources are provided for each light wallet to explore further.

Let's start our journey through the Cardano light wallet space!

Lace

Lace began life as a slim, browser-based wallet for Cardano, designed to make holding and sending ADA as simple as opening a new tab. Fast-forward to its latest release, and Lace has evolved into a full-featured Web3 launchpad: it now supports main-net Bitcoin alongside ADA and Cardano native tokens, integrates NFT management and one-click staking into the same clean interface, and replaces long wallet addresses with short, human-readable Handles. All of this is wrapped in an intuitive extension that installs on Chrome, Brave, Edge, or Firefox in seconds.

But Lace is more than a place to park your coins. Built-in governance tools enable you to cast votes or delegate voting power directly from your wallet. At the same time, a DApp connector drops you straight into DeFi, games, and marketplaces across the Cardano ecosystem. Add a fiat on-ramp, bundled transactions to save on fees, and optional Nami-mode compatibility for older dApps, and Lace becomes a single, lightweight gateway to staking, trading, collecting, and shaping the future of Cardano, as well as Bitcoin and Web3.

Below is a quick tour of what the new Lace can do for you:

- **Move value with ease:** Send and receive ADA, Cardano native tokens, NFTs, and BTC. A built-in gallery shows your collectibles in full colour.
- **Skip address anxiety:** Use short *Handles* (e.g. \1) instead of long cryptographic strings when paying friends or businesses.
- **Buy ADA in seconds:** A fiat on-ramp powered by **Banxa** lets you top up with a debit card or Apple Pay right inside the wallet.
- **Earn rewards automatically:** One-click staking, plus a visual stake-pool explorer, helps you delegate ADA and start earning in minutes.
- **Tap into Cardano dApps:** The *DApp connector* links you to DeFi, games, and marketplaces with one confirmation.
- **Save on fees:** Bundle transactions to send multiple assets to several addresses while paying a single network fee.
- **Stay organised:** Built-in address book and full transaction history keep your portfolio tidy.
- **Secure your keys:** Integrates with the Ledger wallet for cold-storage peace of mind.
- **Vote on the future:** Through [Tempo.vote](#) and [GovTools](#), you can delegate voting power or cast ballots without leaving [walletlace.io](#).
- **Swap to Nami Mode:** Lace now includes an optional Nami compatibility mode. Flip it on in settings, and Lace exposes the same API that older Cardano dApps expect from Nami—perfect for users migrating from the original Nami wallet and for sites that haven’t yet updated their code. It also serves as a beginner-friendly view while the ecosystem transitions to Lace as the default wallet

You can get started with Lace by clicking the “Add to browser” button on [its website](#).

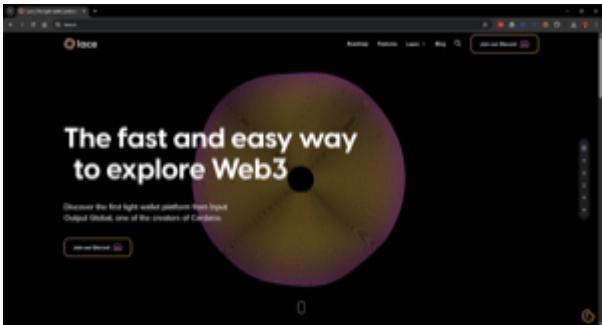


Figure 7. Lace wallet's website.

Lace is open-source software released under the [Apache-2.0 license](#). If you are a software developer and want to contribute to this project and propose new features, you can review the code on the [Lace repository on GitHub](#) and learn more about contributing to Lace's development by using the [Cardano software development kit \(cardano-js-sdk\)](#) as described in [this article](#) on Lace's blog.

Nami

Nami is a browser-based wallet extension designed for the Cardano blockchain. It enables users to interact with decentralized applications (DApps) and smart contracts on the Cardano network. Nami facilitates the management of ada and other Cardano-native tokens and the ability to delegate ada to stake pools for earning rewards.

The wallet is designed to be user-friendly and secure. It stores and transacts with cryptocurrencies on the Cardano blockchain and integrates with various Cardano ecosystem tools and services, making it a convenient choice for users looking to engage with smart contracts and DApps.

Here are some of Nami's main features:

- *Multi-asset support.* It allows users to manage ada and various Cardano native tokens, offering a versatile asset management solution within the Cardano ecosystem.
- *Delegation and staking.* Nami provides features for delegating your ada to stake pools directly from the wallet, enabling users to earn staking

rewards on their holdings.

- *Integrated DApp connector.* Nami offers a built-in connector for interacting with Cardano DApps directly through the wallet.
- *NFT support.* Nami supports Cardano-based non-fungible tokens (NFTs), allowing users to send, receive, and manage NFTs alongside their cryptocurrency assets.
- *Sending and receiving transactions.* Users can easily send and receive ada and other Cardano native tokens.
- *Security features.* As a self-custody wallet, Nami ensures that users have complete control over their private keys, which are stored locally on their devices, enhancing security.
- *Hardware wallet integration.* For added security, Nami integrates with hardware wallets, such as Ledger, allowing users to manage their Cardano assets more securely.
- *Seed phrase backup.* Upon creation, the Nami wallet generates a seed phrase that enables users to recover their wallets in the event of device loss or failure, ensuring asset recoverability.

These features make Nami a comprehensive tool for users looking to interact with the Cardano blockchain, whether dealing with ada transactions, staking, NFTs, or DApps.

Nami is available on Chrome and Brave browsers. To install it, [visit its website](#) and click on the browser icon to navigate to the corresponding web store.



Figure 8. Nami wallet's website.

Nami is open-source software released under the Apache-2.0 license that joined the Input Output Global product family in November 2023\|. If you are a software developer and want to contribute to this project and propose new features, you can review the code on the [Nami repository](#) on GitHub and learn more about contributing to Nami's development.

Eternl

Eternl, previously known as CCVault, is a wallet for the Cardano blockchain. It is designed to be user-friendly and provides features that allow users to manage their ada and other Cardano native tokens. A team of community stake pool developers develops Eternl. Here are some of Eternl main features:

- *Multi-platform support.* Eternl Wallet is accessible on various platforms, including web browsers (as a web application or browser extension) and mobile devices (as a mobile application for iOS or Android). This allows users to manage their Cardano assets across different devices.
- *User-friendly interface.* The wallet's intuitive and clean interface makes it easy for beginners and experienced users to navigate and manage their ada and other Cardano native tokens.
- *Multi-asset management.* Users can store, send, and receive ada and various Cardano native tokens (CNFTs), including non-fungible tokens (NFTs), directly within the wallet.
- *Delegation and staking.* Eternl allows users to delegate their ada to stake pools directly from the wallet, enabling them to earn staking rewards while contributing to the network's security.
- *DApp connector.* The wallet features a DApp connector, enabling users to interact seamlessly with DApps on the Cardano blockchain directly within the wallet.
- *Hardware wallet integration.* Eternl supports integration with hardware wallets like Ledger and Trezor, which provides users with an extra

layer of security by allowing them to keep their private keys offline.

- *Multi-account support.* Users can create and manage multiple wallet accounts, making it easier to organize and separate their funds for different purposes or investments.
- *Transaction history.* The wallet offers a comprehensive transaction history feature, enabling users to track their ada transactions and other token activities.
- *Voting support.* Eternl supports Project Catalyst voting, enabling users to participate in the governance of the Cardano ecosystem by voting on various proposals directly through their wallets.
- *Security features.* Eternl emphasizes security, offering features like seed phrase backup for wallet recovery, encrypted local storage of private keys, and regular security updates to keep users' assets secure.

These features make Eternl a robust and convenient tool for users looking to engage with Cardano, whether they're handling everyday transactions, participating in staking, exploring NFTs, or using Cardano-based DApps.

To get started with Eternl and install it, [visit its website](#), where you will find further information and direct links to the web and mobile stores where you can download and install this wallet.

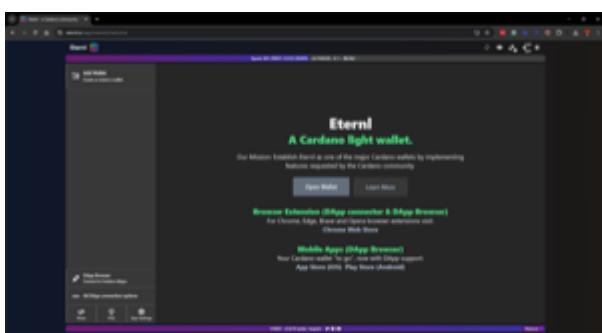


Figure 9. Eternl wallet's website.

Since Eternl is a community wallet, everyone can suggest additional features. Visit their [issues tracker on GitHub](#) to share your ideas with the Eternl development team.

Yoroi

Yoroi is an open-source light wallet for daily use on Cardano, developed by Emurgo, a global blockchain solutions provider focusing on promoting Cardano-based applications. Emurgo is one of the three founding organizations of Cardano, alongside IOG and the Cardano Foundation. Yoroi is designed for Cardano users, offering a simple and efficient way to manage ada and other Cardano-native tokens. Here's a rundown of its main features:

- *Yoroi is available as a browser extension and mobile app* for Chrome, Firefox, and Edge, as well as iOS and Android, offering flexibility in how users access their wallets.
- *User-friendly interface*: Yoroi's clean, straightforward interface is designed with simplicity in mind, making it accessible for beginners while still offering features for advanced users.
- *Secure transactions*. Yoroi emphasizes security by encrypting users' private keys locally on their devices, allowing transactions to be signed without exposing the keys.
- *Staking and delegation*. Users can delegate their ada to stake pools directly from Yoroi, participate in the network's consensus process, and earn rewards.
- *Multi-currency support*. In addition to ada, Yoroi supports a range of Cardano native tokens, allowing users to manage various assets within a single wallet.
- *Instantaneous setup*. Yoroi's setup process is quick and straightforward, requiring no registration or lengthy synchronization processes, allowing users to start transacting almost immediately.
- *Transaction history*. Yoroi provides a detailed transaction history, enabling users to track their transactions over time for better financial management.

- *Hardware wallet integration.* Yoroi can be integrated with hardware wallets like Ledger and Trezor, combining the security of cold storage with the convenience of a hot wallet.
- *Paper wallet import.* Users can import ada from a paper wallet to Yoroi, facilitating a smooth transition from cold storage to a more accessible form of wallet.
- *Catalyst voting.* Yoroi supports Project Catalyst voting, enabling ada holders to participate in Cardano governance.

Yoroi's ease of use, security features, and comprehensive functionality make it a popular choice for Cardano users seeking a reliable and efficient way to manage their ada and engage with the Cardano ecosystem.

To install Yoroi as a browser extension or mobile app, [visit its website](#), where you will find direct links to download each available version.

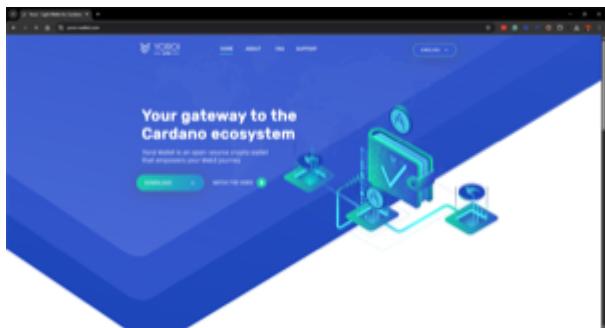


Figure 10. Yoroi wallet's website.

If you are a software developer and want to contribute to Yoroi's development, you can check out the [Yoroi GitHub repository](#).

8.8. 6.2.9 Setting up a Cardano light wallet

Installing a Cardano light wallet as a browser extension, such as Lace, or a mobile app, like Yoroi, is straightforward. While the process may vary between wallets, here are the common steps as a general guide for both methods.

8.8.1. 6.2.9.1 Installing a browser extension wallet

To install a browser extension wallet, for example, Lace, follow these steps:

1. *Choose your browser.* Ensure your browser is compatible with the wallet extension. Popular browsers, such as Chrome, Firefox, Brave, and Edge, typically support such extensions.
2. *Visit the extension marketplace.* Go to the browser's extension store (eg, Chrome Web Store for Chrome, Add-ons for Firefox).
3. *Search for the wallet.* In the search bar of the extension marketplace, type the wallet's name, such as "Lace wallet," and press Enter.
4. *Install the wallet.* Find the desired wallet in the search results and click the "Add to Browser" or "Install" button. Confirm any prompts that appear to proceed with the installation.
5. *Set up the wallet.* Once the wallet is installed, click on the wallet icon in your browser's extension area. You'll likely be guided through an initial setup process, which may include creating a new wallet, importing an existing one, setting a password, and recording a recovery phrase.
6. *Verify and secure.* Ensure your wallet is set up correctly and securely store your recovery phrase offline. This phrase is crucial for accessing your funds if you ever need to recover your wallet.

8.8.2. 6.2.9.2 Installing a mobile wallet app

To install a mobile application wallet, for example, Yoroi, follow these steps:

1. *Choose your platform.* Determine whether you're using an iOS or Android device. Each installation process differs slightly.
2. *Visit the App Store or Google Play.* Open the App Store on iOS devices or Google Play Store on Android devices.
3. *Search for the wallet App.* In the app store's search bar, type the wallet's

name, such as “Yoroi wallet,” and initiate the search.

4. *Install the wallet.* Once you find the wallet app, click the “Install” or “Get” button, depending on your platform. Wait for the app to download and install on your device.
5. *Set up the wallet.* After installation, open the wallet app. Like the browser extension, you’ll go through a setup process that includes creating or importing a wallet, setting up a spending password, and recording your recovery phrase.
6. *Secure your recovery phrase.* It is crucial to write down and store your recovery phrase safely. This is your key to accessing your funds if your device is lost, stolen, or malfunctions.

In both cases, you must follow the setup instructions carefully and ensure that your recovery phrase is stored securely and privately. Whether you use a browser extension or a mobile app, these steps will enable you to manage your Cardano assets conveniently and securely. For detailed installation instructions, please refer to the official website of each wallet.

8.9. 6.2.10 Best practices to secure and back up wallets

No matter your experience with wallets, it’s essential to keep them safe. Here are practices to ensure the security of your digital wallet and assets:

- Never share your private keys or recovery phrase with anyone. These are the only ways to access your funds.
- Create strong and unique passwords. Use a combination of letters, numbers, and special characters. Avoid easily guessable passwords
- Enable two-factor authentication (2FA) where available. This adds an extra layer of security by requiring a second form of verification.
- Keep your wallet software and any related apps up to date. Updates often include security patches that protect against new vulnerabilities.
- Avoid using public Wi-Fi networks when accessing your digital wallet.

Use a secure, private, and encrypted internet connection to protect your data from being intercepted by hackers.

- Access your wallet only from devices you own and trust and that have security software installed. Avoid logging into your wallet from shared or public computers.
- Consider using a hardware wallet for large sums. These devices store your private keys offline, away from online threats.
- Back up your wallet regularly, especially after creating new transactions. Store backups in multiple secure locations. If you're using a physical backup method, such as writing down your recovery phrase, store it in a safe or another secure place.
- Be cautious of phishing attempts. Learn to identify them and be careful of unsolicited communications requesting credentials.
- Stay informed about the best security practices and the latest threats in the cryptocurrency space. Understanding the risks will help you be more prepared to counter them.

Adhering to these best practices can significantly enhance the security of your digital wallets and protect your investments from potential threats.

Chapter 9. 6.3 Common operations

This section explores the functionalities and advanced features of Cardano wallets. Whether you're new to Cardano or seeking a deeper understanding, this guide will help you manage your ada and reveal Cardano's powerful blockchain capabilities.

Navigating day-to-day transactions

First, we'll cover the basics of sending and receiving digital assets like ada – fundamental operations every Cardano wallet user should master. These are the backbone of daily interactions with the Cardano ecosystem.

We will provide step-by-step instructions to ensure that even new users can navigate these processes confidently.

Leveraging advanced wallet features

Beyond basic transactions, Cardano wallets offer advanced features leveraging the unique strengths of the Cardano blockchain. You will learn about:

- Staking ada. Discover how you can participate in staking ada directly from your wallet, securing the network while earning rewards. This section will explain the mechanics of staking, how to choose a stake pool, and the dual benefits of contributing to network security while receiving regular staking rewards.
- Governance participation. Discover how to utilize your wallet to vote on proposals and actively contribute to shaping the future of the Cardano ecosystem.

- Integration with smart contracts and DApps. Discover how your wallet serves as a gateway to advanced features, enabling you to interact with sophisticated applications built on the Cardano blockchain.

By the end of this section, you'll be proficient in managing your transactions and fully equipped to utilize these Cardano wallet features.

9.1. 6.3.1 Sending and receiving digital assets on Cardano

Sending and receiving digital assets is the most common operation in blockchain wallets. On Cardano, you can manage and send various digital assets, including:

- **Ada.** Ada is the native cryptocurrency of the Cardano blockchain. It's used for transactions, staking, and paying transaction fees on the network.
- **Cardano native tokens (CNTs).** With the introduction of the Mary upgrade to the Cardano blockchain, users can create and distribute their custom tokens. These tokens operate on the same blockchain as ada without needing smart contracts to handle the custom token logic. This feature enables the direct management and transaction of various assets through Cardano wallets.
- **Non-fungible tokens (NFTs).** Cardano also supports the creation and exchange of NFTs, unique digital assets verified using blockchain technology that represent ownership of specific items or content, such as art and collectibles.

Cardano's design separates the data layer (where transactions are recorded) from the computation layer (where smart contracts are executed), allowing for efficient handling of different digital assets with reduced costs and improved speed. This makes Cardano wallets versatile tools for managing ada, native tokens, and NFTs within the same ecosystem.

Sending and receiving digital assets on Cardano involves a straightforward process that anyone with a Cardano wallet can perform. Below is the general process for sending and receiving ada or other CNTs, including NFTs, using any Cardano wallet.

Initial requirements

Before you can send or receive digital assets, you need a few essential components:

- A Cardano wallet. You must have a Cardano-compatible wallet, such as Daedalus, Lace, Nami, Yoroi, or another wallet that supports Cardano transactions.
- Wallet setup. Ensure your wallet is appropriately set up. This includes installing and updating your wallet software and completing any necessary setup procedures, such as creating or restoring a wallet.
- Funds in wallet. To send ada or other tokens, you need to have enough ada in your wallet to cover the transaction and any associated fees.
- Recipient's address. To send assets, you need the recipient's wallet address. This should be a valid Cardano address provided by the person or entity to whom you're sending assets.

Once your Cardano wallet is ready, you can send or receive digital assets as follows.

Sending digital assets

Here's the typical process for sending ada or other tokens from a Cardano wallet:

1. Access your wallet. Open your Cardano wallet application on your device.
2. Enter the recipient's address. Navigate to the send section of your wallet. Here, you'll need to input the recipient's Cardano address accurately. Double-check this address to ensure you send assets to the

correct person.

3. Specify the amount. Enter the amount of ada or the specific tokens you wish to send. If you are sending NFTs, you'll select the particular asset from your inventory.
4. Confirm transaction details. Review the transaction details, including the recipient's address, the amount, and the transaction fee.
5. Authorize and send. Confirm the transaction, typically by entering a password or PIN, and, if applicable, approve the transaction using any additional security features, such as 2FA or a hardware wallet confirmation.
6. Wait for confirmation. Once sent, the transaction will be processed on the Cardano blockchain. You can check the status in your wallet or use a Cardano blockchain explorer to see when it has been confirmed.

Receiving digital assets

Receiving ada or other tokens is more straightforward and requires the following steps:

1. Access your wallet. Open your Cardano wallet.
2. Locate your wallet address. Navigate to the receive section of your wallet. You'll find your Cardano wallet address, which you can copy here.
3. Share your address. Provide your Cardano address to the person or service from which you expect to receive assets. You can share this address directly or via a QR code, if your wallet supports it.
4. Check for incoming transactions. Once the sender initiates the transaction, you can monitor your wallet for the incoming funds. The assets will appear in your wallet once the blockchain confirms the transaction.

Following these steps, you can send and receive digital assets on any

Cardano wallet. We encourage you to read the wallet's documentation for detailed information on conducting these actions in a particular wallet.

9.2. 6.3.2 Staking ada

Ada held on the Cardano network represents a stake in the network, with the size of the stake proportional to the amount of ada held.

Staking ada involves participating in transaction validation on the Cardano blockchain. By staking your ada, you support network operations and help verify transactions. In return, you earn rewards in additional ada, incentivizing participants to maintain network integrity.

An ada holder can earn rewards by delegating their stake to a stake pool or by running their stake pool. The amount of delegated stake influences the Ouroboros protocol's selection of who adds the next block and receives rewards.

The more stake is delegated to a stake pool (up to a certain point), the more likely it is to produce the next block and share rewards with its delegators.

9.2.1. 6.3.2.1 Staking ada using a Cardano wallet

Staking ada using a Cardano wallet involves a few straightforward steps that allow you to earn rewards by participating in the network's consensus mechanism. Below is a detailed guide on how to stake your ada.

Step 1: choose a compatible wallet

First, you need a Cardano wallet that supports staking, such as Dedalus, Yoroi, or Lace. The example wallets are provided for informational purposes only and are not endorsed by the authors. Their use is strictly at your responsibility.

Step 2: set up your wallet

- Download and install your chosen wallet
- Create a new wallet or restore an existing one using the recovery phrase
- Secure your wallet by setting a strong spending password and backing up your recovery phrase in a secure location.

Step 3: transfer ada to your wallet

- If your ada is not already in your staking wallet, you should transfer it from where it's currently held (another wallet or an exchange)
- Use the receiving address from your Cardano wallet to transfer ada into it.

Step 4: choose a stake pool

- Within your wallet, navigate to the staking section or delegation center
- Browse or search for stake pools; when selecting a pool, consider factors like:
- Reliability: the pool's uptime and performance history
- Fee structure: the amount that the stake pool charges for its services
- Pool saturation: a measure of how much ada is already staked in the pool; overly saturated pools may offer diminishing returns
- Rewards estimation: some wallets provide estimates of the rewards you might earn from staking with a particular pool.

Step 5: delegate your ada

- Select the stake pool you want to delegate to
- Delegate your ada by confirming the transaction in your wallet; to complete this process, you need to pay a small transaction fee in ada
- Once delegated, your ada remains in your wallet; you simply assign your staking power to the pool.

Step 6: monitor and manage your staking

- After you've delegated your ada, you can monitor the performance of your chosen stake pool directly through your wallet.
- Rewards are usually distributed automatically at the end of each epoch (every 5 days). These rewards will be added to your wallet balance and can be reinvested (re-delegated) to compound your holdings.
- You can change stake pools at any time if you find another pool that offers better returns or aligns better with your preferences.

Additional considerations

- Stay informed. Monitor your stake pool's performance to maximize returns.
- Security. Keep your wallet and recovery information secure. Never share private keys or recovery phrases.

Following these steps, you can effectively participate in staking ada, contributing to the Cardano network's security and decentralization while earning passive rewards. This process not only enhances the stability of the Cardano ecosystem but also offers its participants financial benefits.

9.3. 6.3.3 Governance

Cardano wallets support governance actions within the ecosystem, particularly through the Project Catalyst initiative. This framework enables ada holders to propose, discuss, and vote on development projects that enhance the Cardano network.

Users can participate in this decentralized governance process using wallets like Daedalus and Lace. Wallets provide interfaces for accessing the Catalyst voting system, where users can register and cast votes directly.

This empowers the community to influence the network's future, ensuring

development aligns with global user interests. By participating in governance through their wallets, ada holders help shape Cardano's evolution, fostering an inclusive and democratic blockchain environment.

You can read more about Cardano governance in the Governance chapter.

Chapter 10. Stake Pools and Stake Pool Operation

Chapter 11. Introduction

The strength of the Cardano network lies in its widespread and diverse distribution of stake pools, stake pool operators (SPOs), delegators, builders, and community members at large. The ethos of this well-distributed network and broader ecosystem has found its way into the creation of this chapter.

As you read through this chapter, you may notice varying tones in different sections. This is not by accident. Since the Cardano network is supported by an exceptional global community, it felt appropriate for this chapter to feature contributions from some of the best community members in the Cardano ecosystem.

This chapter aims to deliver a thorough technical and conceptual overview of stake pool operations. For those new to Cardano, this chapter will give insight into what it takes to run and maintain a global, distributed network like Cardano.

It is recommended that prospective SPOs read through this chapter, understand the concepts, and then participate and practice operations on one or more of the publicly available testnets until they gain sufficient expertise before spinning up on mainnet. Seasoned pool operators can use this guide as a refresher, address potential knowledge gaps, and, best of all, contribute where needed.

A list of tools and guides is provided at the end of this chapter to assist with both learning how to operate stake pools and maintaining efficient long-term operations on mainnet.

Chapter 12. What is a Stake Pool?

A Cardano stake pool is a collection of server-based infrastructure that maintains and runs the Cardano blockchain by validating transactions and adding blocks. While a stake pool can run on a single node, it is more common for it to utilize a collection of nodes connected to the broader Cardano network – a distributed network of interconnected stake pools and full nodes. Typically, a stake pool includes a block-producing node and multiple relay nodes (relays).

To become an active block producer, a node requires the following cryptographic files:

- the *verified random function (VRF) signing key* (vrf.skey)
- the *operational certificate* (op.cert)
- the *key evolving signature (KES) signing key* (kes.skey)

When a full node is not started with these files, it is referred to as a passive node or when connected to a block producer a *relay*. A relay sits between the open Cardano network and block producing node(s) of a stake pool. Its purpose is to allow a block producing node to interact with the network securely as the block producing node should only be connected and known to the specific relays the stake pool operator (SPO) controls. This separation of concerns reduces the attack surface of a stake pool and allows the block producing node to focus on minting blocks and propagating these to a small set of well-known relays, which will in turn propagate the blocks to the wider network. Relays are typically publicly known as they are registered with the pool, so other relays can find them in a peer-to-peer manner. Relays may also be run privately with static

topologies listing other relays.

Cardano uses a proof of stake (PoS) consensus mechanism. Network participants holding ada (Cardano's native token) can delegate their stake to a stake pool, effectively pooling their resources and increasing the pool's chances of being selected to validate a block and earn rewards.

Stake pools are run by SPOs, who are responsible for maintaining the pool's infrastructure and ensuring that the pool operates reliably. The rewards earned by the pool are then distributed among the stakeholders who have delegated their ada to the pool, proportional to their stake.

In this way, Cardano's consensus mechanism incentivizes a distributed network of participants to maintain the security and integrity of the network, rather than relying on a single entity to validate transactions.

12.1. Stake Pool Roles

In the Cardano blockchain network, several roles are essential to stake pool operation:

- **SPO.** The individual or organization responsible for running and maintaining the stake pool infrastructure. The SPO determines and sets the pool's operational parameters, such as the fee structure. While the protocol pays rewards directly, the SPO can influence the amount of rewards by adjusting fee parameters. However, they do not control the reward distribution itself.
- **Owner.** The owner is defined by the contribution of pledge to the pool – a type of delegation specific to stake pools. This role can be conducted either by the SPO or a separate entity.
- **Delegator.** A network participant holding ada and delegating their stake to a pool. By doing so, they participate in the validation of transactions and earn rewards.

These roles work together to ensure the reliable and secure operation of

the Cardano network and its consensus mechanism.

12.2. Keys

Cardano cryptographic keys are made up of ed25519 key pairs, which include a public verification key file and a private key file. The public key filename is commonly formatted as `public.vkey`, whereas the private key filename is typically formatted as `private.skey`. The private key file, used to sign transactions, is highly sensitive and must be properly protected. Under all circumstances, this entails limiting third-party access to your private keys. The most effective way to prevent private key exposure is to ensure that it is never stored on any internet-connected device for any length of time. Please note the key pair file names are examples and may be named differently.

12.2.1. Wallet Keys

`root.skey` - the most sensitive private key of any wallet. An attacker can use this key to derive the Cardano wallet payment.skey and stake.skey . In case of a hardware wallet malfunction, the wallet's 12 or 24 word seed phrase can be converted to a root.skey., The payment and stake keys can be recovered by using the CLI and root.skey. A root.skey can thus not only be used to generate private keys for Cardano, but also for other blockchains such as Bitcoin.

`payment.skey` - a highly sensitive Cardano payment address private signing key file. This file provides access to tokens in the payment address and should be kept safe at all times.

`payment.vkey` - the public verification key file for the payment address. It is not sensitive and can be shared publicly.

`stake.skey` - a sensitive Cardano stake address private signing key file. This file gives you access to any rewards held in the stake address, as well as the ability to delegate the wallet to a pool.

stake.vkey - stake address public verification key file, which is not sensitive and can be shared publicly.

12.2.2. VRF Hot Keys

VRF keys control participation in the slot leader selection process:

vrf.skey - a private signing key file for a Cardano stake pool's VRF key. This key is required to start a stake pool's block producing node. It is sensitive but must be placed on a hot node to start a stake pool.

vrf.vkey - a public verification key file for a Cardano stake pool's vrf.skey. It is not sensitive and is not required to start a stake pool's block producing node.

12.2.3. KES Hot Keys

KES keys are operational node keys that authenticate the rightful operator of a pool:

kes.skey - a private signature key file for the stake pool's KES key. It is needed to start the stake pool's block-producing node. This key is sensitive, however, it must be placed on a hot node to start a stake pool and be rotated regularly. KES keys are required to establish a stake pool's operational certificate, which expires 90 days after the certificate's defined KES period has passed. As a result, fresh KES keys must be generated along with a new certificate every 90 days or sooner for a Cardano stake pool to continue minting blocks.

kes.vkey - a public verification key file for the pool's corresponding kes.skey. This key is not sensitive and is not required to start a block-producing node.

12.2.4. Stake pool keys and supporting files

Stake pool keys and some supporting files include:

cold.skey - a private, offline signing key file for a Cardano stake pool,

which is very sensitive. The cold.skey is required to:

- register a stake pool
- update stake pool registration parameters
- rotate stake pool KES keys
- retire a stake pool

cold.vkey - a public verification key file for a stake pool's cold.skey private signing key file. It is not sensitive and can be shared publicly.

cold.counter - a counter file that tracks the number of times an operational certificate has been generated for the relevant stake pool.

Note: Always rotate KES keys using the latest cold.counter and increment by exactly 1.

op.cert - an operational certificate that links the operator's cold.skey and their operational kes.skey. The certificate checks whether the operational key is valid, preventing malicious interference. The certificate identifies the current operational key and is signed by the offline cold.skey. As the operational certificate is sensitive and required to start a stake pool's block-producing node, it must be placed on a hot node to start a stake pool.

12.3. Addresses

Current Cardano wallet addresses are encoded in bech32 format and consist of two parts: a payment address and a staking address. The payment address, along with its associated key pairs, is used to store, receive, and send funds. The staking address and its related keys manage staking-related activities, such as storing and withdrawing rewards, defining the stake pool owner, managing reward accounts, and setting the wallet's target stake pool delegation.

An enterprise address is a specific type of Cardano wallet address without the ability to participate in staking. Enterprise addresses might be used by

exchanges that wish to comply with regulations or in cases where staking is not desired.

payment.addr - a payment address is usually generated using both a payment.vkey and a stake.vkey, associating the resulting address with both keys. Usually the payment.vkey and stake.vkey are derived from the same cryptographic entropy or root.skey, which refers to the same original secret or seed phrase. The first payment address of any wallet is known as the base address. Cardano supports HD wallets, so any number of payment addresses can be derived from the same secret phrase.

stake.addr - a stake address for a Cardano wallet is generated using the stake.vkey file. It controls protocol participation, enables stake pool creation, and facilitates delegation and receiving rewards. This address cannot receive payments but is used to receive rewards from participating in the protocol. Only one stake address can be derived from a single original secret, thus all payment.addr associated with the same secret share the same stake address component.

It is also possible to combine payment.vkey and stake.vkey from two different original secrets, creating what is known as a **mangled** or **Franken Address**. Sending tokens to such an address allows one wallet to spend them, while the other wallet can participate in the protocol and earn rewards using the same tokens.

12.4. Pool Saturation

The protocol parameter **k** defines the saturation point for stake pool rewards. This saturation point is calculated by dividing **Cardano's total supply by k**. As of this writing, with k set at 500, the saturation point is approximately 70 million ada. A stake pool with total stake above this saturation point will receive more slots and should mint more blocks. However, rewards for the pool are capped at the saturation point. This results in diminishing returns as the same rewards pot is distributed

among a larger total delegation. This incentivizes participants to avoid delegating to saturated pools.

Reaching saturation is a positive sign for a pool, as it indicates the pool's popularity and substantial stake. However, it also means that delegators may have to look for alternative pools if they wish to delegate additional stake. The SPO of the saturated pool can also adjust parameters, such as increasing pool minting fees, to manage the saturation level.

The SPO can also open additional pools, leveraging the popularity of the first pool to attract stake to the new ones. This practice, known as pool-splitting or multi-pools, remains a contentious topic in the Cardano community. From a decentralization perspective, an SPO should refrain from pool splitting unless they can fully saturate the existing pool with pledge, effectively turning it into a private pool before opening a new public one. Distributed pool production increases the minimum attack vector (MAV) and protects against Sybil attacks (where a malicious actor creates multiple entities or accounts to gain control over block production).

12.5. Pledge vs Stake

Pledge is a specific type of delegation registered on-chain as part of the pool registration process. This means that the pool's owner/s permanently delegate a certain amount of ada to the pool, registering a pledge commitment. If the pool fails to meet the pledge commitment – meaning the amount pledged on-chain exceeds the total ada in the pool's pledge wallets at the epoch transition snapshot – no rewards will be paid. To resolve this, the pool must either adjust the registration to lower the pledge so it becomes valid again or return the pledge to the designated owner addresses, fulfilling the original commitment.

To prevent Sybil attacks, Cardano uses a well-designed incentive model that encourages stake pool owners to associate as much pledge as possible

with their pools. A higher pledge makes the stake pool more financially attractive to potential delegators. Staking rewards increase linearly with the amount of pledge, reaching maximum rewards when the pool is fully saturated. Such a pool earns nearly 30% more rewards than a pool with no pledge. Saturated pools are often referred to as private stake pools, as the incentive to add delegation diminishes once the saturation threshold is met.

As there is no minimum pledge requirement, a stake pool can operate with zero pledge. However, having a pledge demonstrates the owner's commitment, which can help build trust among delegators. Declaring pledge can also carry potential risks, as this information is publicly available on-chain. The SPO could become a target for criminal activity attempting to steal the pledge. Additionally, regulatory requirements that mandate the disclosure of SPO identities in certain jurisdictions can pose further security risks.

Franken addresses could be useful in these scenarios. An SPO could declare the pledge while managing the pool without direct access to the payment key of the pledge. The SPO would only have access to their wallet with a payment key securing minimal funds for day-to-day operations sufficient to pay for pool maintenance fees. The main pledge is secured with another wallet's payment key. The owner address would consist of one payment address from the pledge wallet and the staking address from the SPO wallet. As long as the pledge resides on this mangled address, the SPO can incorporate the stake as pledge without the risk of physical extortion, as they would have no access to the pledge payment key.

Regular stake refers to the standard stake that participants bring in through delegation. This stake is liquid allowing users to move it in and out of delegation at will. While pledge is not technically locked either, moving it out before pledge commitment changes become active will break the commitment, likely resulting in a loss of rewards.

12.6. Fee Structure

The declared fees of a Cardano stake pool, determined by its operator, are critical for long-term sustainable operation. Fees typically cover costs such as:

- fixed server cost
- time spent on pool and server maintenance
- marketing expenses, regulatory compliance costs, including management and reporting duties

The fee structure determines how much of the rewards generated by the pool are retained by the operator and how much is distributed to delegators. If the pool does not mint blocks in an epoch, no rewards are generated and no fees are paid. Pool fees are always taken from the rewards, meaning they can diminish delegators' rewards. However, fees are never taken from delegators' stake, as Cardano natively uses non-custodial, [liquid staking](#).

Cardano stake pool operators set two types of fees:

- **Fixed fee** (minPoolCost): specified as a set amount in ada, this fee is intended to cover the fixed cost of pool operation. As of writing, the minimum mandatory fee is 170 ada per epoch.
- **Variable fee**: specified as a percentage, this fee can be deducted from the total staking rewards after the fixed fee is applied. Unlike the fixed fee, which remains constant as long as at least one block is minted in an epoch, the variable fee depends on the pool's performance and the number of minted blocks relative to expected blocks. The more blocks minted, the higher the total rewards, the higher the variable fee payout for the operator. Typically, at the time of writing, the variable fee ranges from 0% to 5%.

Note: While a 5% variable fee may seem significant, it should always be

assessed in relation to the expected return on staking (ROS). As of writing, the ROS is approximately 4% per year. A stake pool that sets a 5% variable fee will only reduce the annual ROS by 0.2 percentage points, resulting in a ROS of 3.8% per year.

Chapter 13. SPO Requirements

As an open protocol, Cardano allows anyone to create and operate a stake pool using the free and open-source software (FOSS) – `cardano-node` and `cardano-cli`. However, it is essential for stake pool operators to possess the technical expertise required to operate a pool reliably and securely. While the requirements outlined in this section are not authoritative, they are generally recognized as critical for successful stake pool operation.

It is also important to note that many community-made tools and scripts are available, which significantly lower the technical barrier to entry for operating a stake pool. These tools assist with operational tasks but are not a substitute for the necessary technical knowledge or skills required for effective stake pool management.

13.1. Linux

A strong understanding of and ability to navigate, utilize, and manage the Linux operating system is a must for any stake pool operator. While `cardano-node` binaries are available for Linux, Mac, and Windows with each release, only Linux is supported in the block producer and relay roles. It is possible to build from source and operate `cardano-node` and `cardano-cli` on OpenBSD and potentially other Unix-based operating systems, but this is an advanced and separate topic.

Suggested readings and courses:

- [Introduction to Linux \(LFS101x\) by The Linux Foundation](#)
- [The Linux command line for beginners Tutorial by Canonical](#)

13.2. Networking

A solid understanding of transmission control protocol/internet protocol (TCP/IP) and experience with network maintenance are essential for SPOs managing a distributed network.

Suggested readings and courses:

- [TCP/IP Networking reference guide by Penguin Tutor](#)
- [CompTIA Network + Certification](#)
- [The TCP/IP Guide](#)

13.3. Documentation and Learning

SPOs must keep their nodes updated and stay informed about new node versions and upcoming on-chain changes.

The Cardano node source code is open-source, allowing SPOs to review, clone, fork, compile, and improve it. While Cardano's code is currently hosted on GitHub, it could be available on any version control platform. SPOs should be capable of navigating projects on GitHub, understanding the documentation, and following the steps to compile and run the node.

Suggested reading:

- [Cardano.org guide for installing Cardano node](#)

13.4. Getting Started

Prospective SPOs, whether new to system administration or experienced, should start by using Cardano on public testnets. Currently, IOG supports two public testnet environments: preview and pre-production. Preview allows developers and users to test and provide feedback before updates are released on the mainnet, typically forking four epochs ahead. Pre-production is primarily for SPOs and developers to test major upgrades before deployment on the mainnet, usually forking one epoch ahead.

Other public testnets, such as the [Guild Network](#) are also available for use.

When considering launching a stake pool, it may be tempting to download and run the excellent scripts and tools created by the community for day-to-day operations. However, these tools are designed to assist those who already possess the necessary technical knowledge and skills.

Instead of rushing directly to mainnet operation, SPOs should take the time to learn essential tasks on existing testnets. This includes creating keys, compiling the node and CLI, crafting transactions, and registering signed certificates on the blockchain. Hands-on experience will reduce stress and save time when challenges arise during stake pool operations.

Here are a few useful guides and courses for setting up a stake pool:

- [Cardano Docs: Creating a stake pool](#)
- [Cardano Course: Handbook](#)
- [Cardano Developer Portal: Operate a Stake Pool](#)
- [Concashew's Stake Pool Guide](#)

Note: The recommended reading and courses in this section suggest potential starting points for those interested in learning more about stake pool operations. They do not constitute endorsements.

13.5. Putting it all together, long time maintenance

The stability of the Haskell node has significantly improved, making basic node operation less challenging than during the early days of the Incentivized Testnet (ITN). This is especially true when using community tools mentioned in the ‘Getting started’ section, such as [CnTools](#), [JorManager](#), [StakePool Operator Scripts](#), and others.

Prior to mainnet operations, prospective SPOs should master such key areas:

- **Monitor node health:** set up alerting systems for issues such as missed blocks, node crashes, or stuck block height.
- **Read and interpret logs:** analyze cardano-node logs to troubleshoot potential issues and investigate each missed block. Long-term luck should approximate 100%. Although block collisions (where multiple pools are assigned the same slot) do occur, they are uncommon. More details are covered in the [**Slot Battles, Height Battles, Forkers and Propagation**](#).
- **Collaborate on GitHub:** help other SPOs analyze, contribute to, and create cardano-node issues for testnets or mainnet.
- **Maintain infrastructure:** execute node or infrastructure updates seamlessly, ensuring no service interruption.
- **Harden the pool environment:** strengthen security around the pool infrastructure.
- **Sync to universal time:** keep both the node and server synchronized with universal time to prevent operational issues.
- **Handle pool registration securely:** perform updates without exposing sensitive keys.
- **Expand node resources:** increase resource provisioning for cardano-node without interrupting services.
- **Plan for failover:** prepare and configure systems to handle critical node failures.

Chapter 14. Assigning Leadership Slots to Stake Pools

14.1. Overview

On proof-of-work blockchains like Bitcoin, miners generate blocks by solving cryptographic puzzles, which is highly resource-intensive. In contrast, Cardano, a proof-of-stake blockchain, selects stake pools to create blocks using a stake-weighted lottery system. This system is detailed in the [Ouroboros paper](#), and this section offers a simplified overview.

14.2. Epochs, Blocks, and Slots

The leadership schedule on Cardano is divided into epochs and slots, with epochs being longer than slots. On the Cardano mainnet, each epoch lasts five days and begins at 21:44:51 UTC. Every epoch contains 432,000 one-second slots (five days). While different configurations may exist for other Cardano-based test networks or sidechains, this section focuses on the Cardano mainnet.

There is a chance for block creation in each slot. To ensure a secure process, each stake pool node must determine if the pool is:

- Allowed to create a block in a specific slot
- Able to prove to other nodes that it was permitted to create the block
- Able to hide its selection for future block creation from others

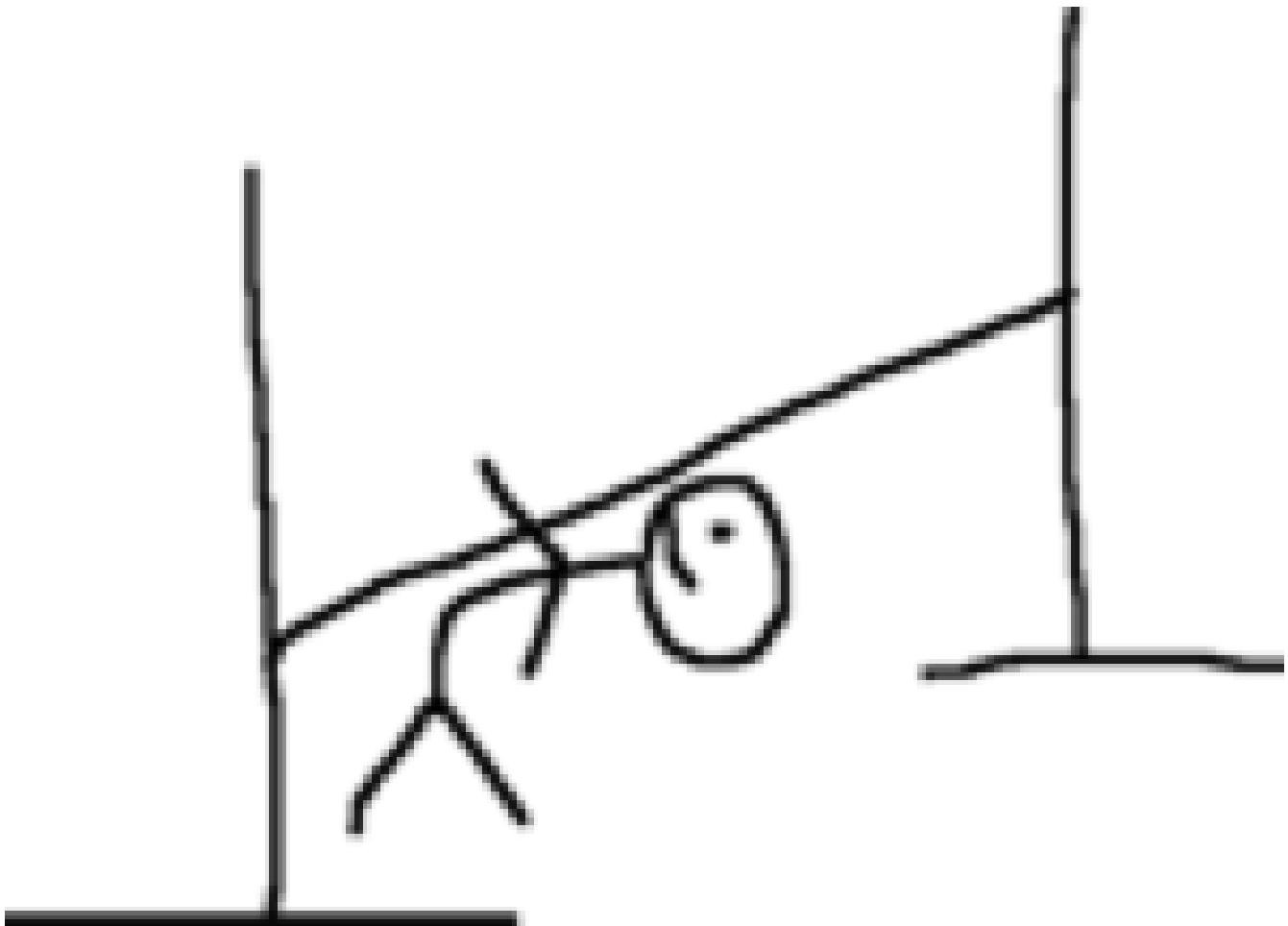


Figure 31. Limbo

14.3. Playing Limbo

To simplify how a pool is chosen to create a block, imagine it as a game of Limbo. To win, a person (the stake pool) must go under the bar (a threshold value). For each slot, the bar is set at a unique height for each stake pool, determined by the pool's stake. Larger pools have a higher bar, making it easier for them to win (create blocks), while smaller pools have a lower bar. The bar's height isn't fixed but is randomly adjusted for each participating pool based on its stake.

To determine if a stake pool can create a block in a given slot, several factors are assessed.

First, the epoch nonce is considered. This is a random number formed from two key components: the rolling nonce (which updates with every block) and a block hash. The rolling nonce is selected from the block just before the stability window of the previous epoch, which lasts 1.5 days. As

a result, leadership information for the next epoch can only be calculated 1.5 days before it begins.

The second value used in the epoch nonce is the hash of the last block from the previous epoch. These values are concatenated and hashed to produce the epoch nonce, which is the same for all pools.

The epoch nonce, the absolute slot number, and the pool's VRF secret key are then combined to generate a random output for each slot. This output is weighted by the pool's stake relative to the total ada staked in the system. If the weighted value is below a certain threshold, the pool is permitted to create a block – winning the game of Limbo for that slot!

14.4. Security

Security is maintained by ensuring only the pool operator knows when they will create a block. The pool's VRF secret key is used to determine leadership selection, while the VRF public key is published on the blockchain. This ensures that only the pool operator is aware of their block-making opportunity, while others can verify the VRF signature after the block is made. This process ensures fairness and prevents anyone else from predicting block creation, making it impossible for attackers to target a stake pool with a DDoS attack.

Chapter 15. Slot Battles, Height Battles, Forkers and Propagation

15.1. Ouroboros leader selection review

Based on the Ouroboros protocol, Cardano stake pools create blocks on behalf of their delegators. This protocol operates a lottery in every slot (one per second, per current parameters), with a pool's chance of being selected as a leader proportional to its stake. If a pool wins the lottery, it becomes the leader for that slot. While the full details of the algorithm are covered in the [Ouroboros paper](#) and other sections of this book, the key point is that any pool with stake can potentially be a leader in any slot. This can result in multiple pools being chosen as leaders for the same or nearly the same slot. However, only one block can be accepted on the chain for a given slot, leading to situations known in the Cardano community as 'battles.'

15.2. Types of battles

To understand the types of battles in Cardano, it is essential to review key aspects of blockchain functionality. Consider the illustration below featuring three blocks, starting with the one on the left at height 8,265,668 in slot 244,252, with a block hash of 'c7b2...8bac.' The next block, with a hash of 'f777...498c' at height 8,265,669, includes the parent hash 'c7b2...8bac,' indicating it is built on the previous block. This block was created 9 slots (9 seconds, per current protocol parameters) later, in slot 244,261. Finally, the third block at height 8,265,670, which includes the parent hash 'f777...498c,' was created in slot 244,309. This illustrates how blocks are

sequentially built upon one another, contributing to the growth of the blockchain.



Figure 32. Basic blockchain illustration

15.2.1. Slot Battles

A slot battle is a situation when two blocks are presented at the same height, have the same parent hash, and occur in the same slot.



Figure 33. Slot Battle

15.2.2. Height Battles

A height battle occurs when two blocks are presented at the same height, share the same parent hash, but have different slots.



Figure 34. Height Battle

15.2.3. Forkers

Forkers refers to a situation where a pool operator runs multiple block-producing nodes simultaneously. This often happens during system upgrades when a second instance is launched before the old one is shut down. Both instances may create and distribute blocks to the network.

While this does not constitute a battle, it can lead to confusion and inefficiencies in the blockchain, making it important for the community to minimize.

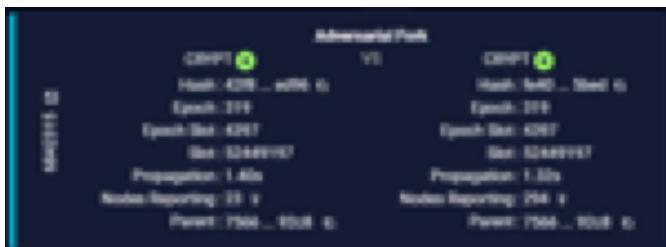


Figure 35. Forker

15.2.4. Summary

The following table summarizes the types of contention we can have on the Cardano blockchain.

Type	Same Slot	Same Parent Hash	Same Height	Same Stake Pool
Slot Battle	Yes	Yes	Yes	No
Height Battle	No	Yes	Yes	No
Forker	Yes	Yes	Yes	Yes

15.3. Resolution of battles and forks

When a battle or fork occurs on the network, all nodes must reach consensus on which block to add to the chain and which one(s) to discard. In the Praos era (post-Vasil hard fork), two primary rules guide this decision:

1. Chain Length, the longest chain length is always preferred.
2. If the chain length is the same, we choose the block with the lowest block vrf.

15.3.1. Block VRF

The block VRF is a value generated from the epoch nonce, the slot number,

and the private VRF key registered by the pool. This value does not depend on the block's contents and cannot be manipulated by the pool operator. Block VRF values can be quite large, represented as a 128-character hexadecimal number.

15.4. Propagation

Propagation refers to the process of distributing a block made by a pool to the network for inclusion in the blockchain. Pool operators often invest considerable effort in optimizing their setups to ensure their blocks reach as many nodes as possible. While the technical details of this process are beyond this section's scope, it is crucial to recognize that blocks do not reach all network participants simultaneously, and there is a time delay in their distribution.

The Cardano ecosystem provides tools to visualize each pool's propagation time. The example below illustrates a pool's measured propagation for the blocks it has created. On average, every node receives the block within approximately 600 milliseconds, though some outliers take significantly longer. This highlights the importance of having an effective distribution strategy for blocks.

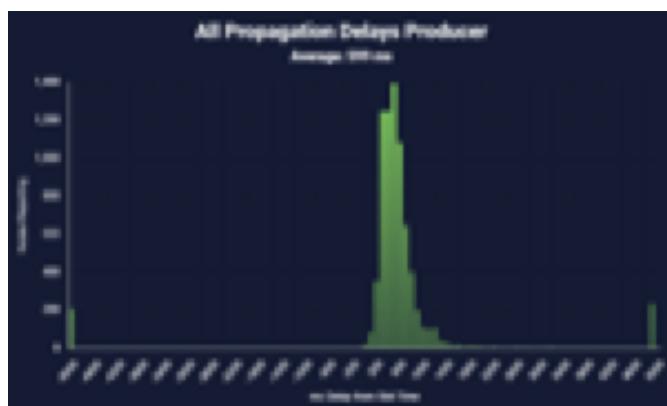


Figure 36. propagation time

15.4.1. Tying it all together: Some real examples

15.4.1.1. The typical situation:

In the typical situation, over 90% of the blocks on the chain are created

smoothly, without issues. The illustration below depicts a normal scenario where one pool creates a block, followed by another pool building a block on top of it. In this example, the first pool creates a block in slot 10, and the second pool creates a block in slot 15. The second pool's block is built on top of the first pool's block, allowing the chain to continue growing.

<need illustration>

1. Pool A creates a block in slot 10
2. Pool A sends a block to all of its peers on the network
3. Pool A's peers forward the block further to all of their peers and so on. Some nodes receive the block in slot 10, some in slot 11, some in slot 12, etc.
4. Pool B creates a block in slot 15 and because it already has seen Pool A's block, it builds on top of it and extends the chain further and the cycle continues.

15.4.1.2. The height battle example:

But what if pool B creates a block in slot 15 and does not have pool A's block yet? This scenario leads to a height battle, as both pool A's block in slot 10 and pool B's block in slot 15 will share the same parent block. The contention is settled by comparing the block VRF values; the block with the lowest VRF value will be added to the chain, while the other block will be discarded.

15.4.1.3. The slot battle example:

The slot battle example illustrates a scenario where both pool A and pool B create a block in slot 10, leading to immediate contention. This situation results in a slot battle, where the block with the lowest VRF value is added to the chain, and the other block is discarded.

Interesting Historical Trivia

Interesting historical trivia: In earlier versions of the Cardano ITN protocol, the first pool to submit a block would win the battle. This approach led to centralization, as nodes with faster internet connections or those located closer to most other nodes consistently prevailed in the battle.

15.4.1.4. The underpowered node problem:

Sometimes, pool operators encounter problems with their nodes that slow down block production. This issue is often due to underpowered hardware or unoptimized node configurations. As a result, other pool operators may lose blocks through no fault of their own.

1. Pool A creates a block in slot 10
2. Pool B is assigned a block in slot 11 but is very slow to generate the block and it takes 5 seconds to create and propagate the block to other nodes.
3. Meanwhile Pool C is assigned a block in slot 14 and since it hasn't seen Pool B's block yet, it builds on top of Pool A's block.
4. Finally in slot 15 Pool B's block is distributed and all nodes need to pick between Pool B's block and Pool C's block when if all pools were operating at full speed, Pool B's block would have been the one that was added to the chain and then Pool C's block would have built on top of that.

Underpowered nodes can cause pools to lose blocks without justification, resulting in lower chain density and reduced decentralization. This highlights the importance of having a robust block distribution strategy for pool operators.

Chapter 16. `cardano-cli`

16.1. Prologue

As its name suggests, the Cardano command line interface (`cardano-cli`) is the low-level CLI component of the `cardano-node` repository. It serves as a perfect companion to `cardano-node`, often built alongside it or provided as a pre-compiled binary. The primary purpose of `cardano-cli` is to interact with the blockchain managed by `cardano-node`. Would you drive a car without a steering wheel?

This section focuses on the subset of `cardano-cli` commands most useful to Cardano stake pool operators. It outlines the steps required to set up a stake pool from the very beginning, including:

- Key generation for addresses and pools
- Certificate generation
- Node queries
- Basic transactions

To walk through these steps, let's assume you have:

- `cardano-cli` installed in your machine
- `cardano-node` installed, running, and synced to a publicly available Cardano network (eg, preview, pre-production, or mainnet)
- the `CARDANO_NODE_SOCKET_PATH` properly set in your environment so that `cardano-cli` can communicate with your running instance of `cardano-node`

Note that the last two points are only necessary when submitting a transaction or querying the ledger. A running node is essential for interacting with or accessing the current state of the blockchain. However, generating keys, addresses, or certificates only requires cardano-cli and should often be done entirely offline for security reasons.

Before starting, check that both cardano-node and cardano-cli are up to date. Paste the following code snippet into your terminal:

```
cardano-node version
```

Your terminal should return a response similar to the following:

```
cardano-node 1.35.6 - linux-x86_64 - ghc-8.10  
git rev 07b0c71d2e6662aec4389ec282a7e91f68c3d85f
```

Ask **cardano-cli** for its version using the command:

```
cardano-cli version
```

```
cardano-cli 1.35.6 - linux-x86_64 - ghc-8.10  
git rev 07b0c71d2e6662aec4389ec282a7e91f68c3d85f
```

This indicates that both tools are running the same version (1.35.6) and originate from the same code branch (git rev 07b0c7...3d85f). Perfect!

Notice how we access the version command of cardano-cli (or cardano-node) using a single space character to navigate through the available commands. The version command takes no extra parameters. As you will see later, parameters are passed using a double hyphen (`--`).

Chapter 17. Keys generation

Public/private key pairs are fundamental in cryptography. In Cardano, the holder of a private key can spend ada from an address, sign a block, or prove to other nodes their authority to sign a specific block.

Note: It is essential to protect all private keys from both unauthorized access and accidental deletion. Since the launch of Shelley, there have been numerous examples of mismanaged keys leading to non-functional stake pools and, in rare instances, stolen funds. Proper key management is crucial for the security and functionality of a stake pool.

17.1. Addresses

Every action recorded on the ledger requires a transaction fee. An address must cover this transaction fee and/or provide funds as a deposit (see the Basic transaction section). This section describes how to generate address key pairs and derive a base address used to create and register a stake pool.

Note that this section is not a complete reference for cardano-cli concerning addresses.

17.1.1. Payment key pair

We will refer to this as the **payment** key pair. The suffix **vk** identifies the **payment public key** (or verification key), while the suffix **sk** identifies the **p*ayment private key*** (or secret key). In Cardano, ada contained in an address is controlled solely by the **payment.sk**.

Let's generate one:

```
cardano-cli address key-gen --verification-key-file ./payment.vk  
--signing-key-file ./payment.sk
```

As mentioned earlier, `address` and `key-gen` are two nested commands of `cardano-cli`, while `--verification-key-file` and `--signing-key-file` are parameters of the entire command.

In the directory where you executed the command, you should have the following files:

```
payment.sk  
payment.vk
```

Both files have the structure shown below:

```
cat ./payment.vk
```

```
{  
  "type": "PaymentVerificationKeyShelley_ed25519",  
  "description": "Payment Verification Key",  
  "cborHex":  
    "5820205d42785c7dc9a46898655ecda8dad8b14e47747dc94ba184edc8ada0b72969"  
}
```

The `payment.sk` has the same structure, with the type `PaymentSigningKeyShelley_ed25519` and of course a different `cborHex` value. You will not learn more, it is a secret after all.

Now, we have what we need to derive an address to receive funds. Use the following command:

```
cardano-cli address build --payment-verification-key-file ./payment.vk  
--mainnet > payment.addr
```

The newly created `payment.addr` file contains an address in the format:

```
addr1v9m8pcfxszyvx7gytqc2s91400aund8z7sazfs2jtfy4h3gnt67k6
```

Three things to note:

- `--payment-verification-key-file` is the sole parameter needed for payment address creation.
- We used the `--mainnet` parameter to create this address. It means that this address won't be of any use on a testnet. Conversely, the `--testnet-magic` parameter would allow us to create an address for a specified testnet. Mishaps avoided!
- Cardano Preview Testnet is on `--testnet-magic 2` and Cardano Preprod Testnet is on `--testnet-magic 1`

17.1.2. Staking key pair

Using the address generated above has one significant drawback: it can receive and send ada but has no staking rights associated with it, making it what we call an enterprise address. To enable staking rights, we need to add a staking key to the address.

In the same folder where the payment key pair is located, let's generate a **staking** key pair. Following convention, we will call them `stake.vk` and `stake.sk`:

```
cardano-cli stake-address key-gen --verification-key-file ./stake.vk  
--signing-key-file ./stake.sk
```

Both files have the following structure:

```
{  
  "type": "StakeVerificationKeyShelley_ed25519",  
  "description": "Stake Verification Key",  
  "cborHex":  
    "5820eaa448543c3f95cbecf5c7ef00e481695388462c7e470b90052920138272a88b"
```

```
}
```

Similarly, we can build the corresponding staking address using the `stake.vk`.

```
cardano-cli stake-address build --stake-verification-key-file ./stake.vk  
--mainnet > stake.addr
```

The newly created `stake.addr` file contains:

```
stake1uy4h1pcmhd026m4ny9y9uxl94rez479g8h0sqalljf9zehguqnhcM
```

17.1.3. Building an address

As noted, both the payment key pair and the staking key pair can be used to generate addresses. While the payment address (or enterprise address) can be used to receive or send ada, the staking address serves a different purpose. A staking address is not functional on its own; it only becomes active when linked to ada residing within a payment address.

```
cardano-cli address build --payment-verification-key-file ./payment.vk  
--stake-verification-key-file ./stake.vk --mainnet > base.addr
```

The newly created `base.addr` file contains an address that enables:

1) sending and receiving ada 2) delegating and receiving rewards

```
addr1q9m8pcfzszyvx7gytqc2s91400aund8z7sazfs2jtfy4h3ft07r3hw6744htxg2gtcd  
7t28j9tu2s0wlqpmllyj29nwssucyxN
```

Note that this address is longer than an `enterprise address`.

17.1.4. File Summary #1

Your working directory should now contain 7 files:

1. `base.addr`

2. payment.addr
3. payment.sk
4. payment.vk
5. stake.addr
6. stake.sk
7. stake.vk

It is now time to back up these key pairs and send some ada to the base.addr to fund the next venture: creating a stake pool.

17.2. Stake Pool related key pairs

The creation and operation of a Cardano stake pool relies on three public/private key pairs.

17.2.1. Stake Pool keys or Cold Keys

Stake pool keys, or cold keys, should be kept in a cold environment, meaning they should reside on a machine that is permanently disconnected from the internet or any other networks.

Note: The pool private cold key governs all pool actions, including pool creation, key rotations (such as KES or VRF key pairs), fee structure, reward and pledge address settings, and pool retirement.

```
cardano-cli node key-gen --cold-verification-key-file ./pool.vk --cold
-signing-key-file ./pool.sk --operational-certificate-issue-counter-file
./counter
```

Notice that in addition to the **cold-verification-key-file** and **cold-signing-key-file**, we also generated a mandatory **counter** file.

```
{
  "type": "NodeOperationalCertificateIssueCounter",
  "description": "Next certificate issue number: 0",
  "cborHex": "820058203e9dff9346dab83c109a9da73aabf4642ebe64e0274b6a0931ee4b8d838ea30
4"
```

```
}
```

This **counter** will be used to create an **operational certificate** for the stake pool. For now, let's keep in mind that the **operational certificate** is generated by using this **counter** and a KES public key defined in the section "KES keys pair".

We have not yet registered the Cardano stake pool, but we can already determine its future on-chain ID.

```
cardano-cli stake-pool id --cold-verification-key-file ./pool.vk
```

and the answer is

```
pool1xhjzslnkyxvj23almagsmzeck0el7989cqz9rlms8a0pvdl0de
```

17.2.2. VRF keys pair

The VRF key pair is used by the node to determine whether to attempt signing a block. At every slot, the node checks this condition.

```
cardano-cli node key-gen-VRF --verification-key-file ./vrf.vk --signing-key-file ./vrf.sk
```

Let's have a look:

```
{
  "type": "VrfVerificationKey_PraosVRF",
  "description": "VRF Verification Key",
  "cborHex":
"5820b49718bee9e359b666950c255f2ff7a3ace260963baeb8e8b618d75575dd8ce7"
}
```

The VRF key will reside on the connected block-producing node, as it is used as a parameter to start **cardano-node**.

Note: While it is possible to modify the VRF key of a stake pool by sending

a new pool registration certificate, doing so will forfeit pool rewards for two epochs. This requirement helps encourage tight security practices.

17.2.3. KES keys pair

The KES key pair is the key used by the node to sign a block.

```
cardano-cli node key-gen-KES --verification-key-file ./kes.vk --signing  
-key-file ./kes.sk
```

Let's have a look:

```
{  
  "type": "KesVerificationKey_ed25519_kes_2^6",  
  "description": "KES Verification Key",  
  "cborHex":  
    "5820f93acee67a1af6529ff02818a18c813d05a71c3cde8a16606133dbbee7f583bc"  
}
```

The KES signing key must also reside on the block-producing node.

A **kes.sk** used by a node has a validity range of 93 days (or 62 KES periods) and needs to be renewed before this period expires. That's where the **counter** and the **operational certificate** come into play.

17.2.4. Operational certificate

To create an **operational certificate** run:

```
cardano-cli node issue-op-cert --kes-verification-key-file ./kes.vk  
--cold-signing-key-file ./pool.sk --operational-certificate-issue  
-counter-file ./counter --kes-period 694 --out-file op.cert
```

Some explanations are in order:

- The **counter** will automatically be incremented by exactly **one** after running the above command. You can check this:

```
{
  "type": "NodeOperationalCertificateIssueCounter",
  "description": "Next certificate issue number: 1",
  "cborHex":
    "820158203e9dff9346dab83c109a9da73aabf4642ebe64e0274b6a0931ee4b8d838ea304"
}
```

The `--kes-period` defines the starting point of a validity range for the `'kes.sk'` referenced in the operational certificate. One way to calculate the `current-kes-period` of the network on Cardano mainnet is to use the formula below. We assume here that `byron_slots`, `byron_end_time` and `slots_per_kes_period` are constant values:

+

```
current-kes-period = (byron_slots+(CurrentTime - byron_end_time))/slots_per_kes_period
current-kes-period = (4492800+(CurrentTime-1596059091))/129600
```

+ `CurrentTime` on your machine can be obtained like this:

+

```
printf '%(%s)T\n' -1
```

+ Some noteworthy Cardano community members have developed fully parameterized methods to calculate the current KES period for any given network, such as [this one](#). Their contributions are invaluable.

- You must generate a new KES key pair and a new operational certificate before the end of the validity period, which lasts exactly 62 KES periods. Your block-producing node will need to be restarted using a new `kes.sk` and a new operational certificate. This process is called ‘KES rotation’. One period corresponds to 1.5 days. That is why KES rotation must occur every 93 days at most. However, you can

perform this rotation earlier if it is more convenient.

In the example above using `--kes-period 694`, the **operational certificate** will certify that the `kes.sk` is valid until the network reaches the **kes-period 756**.

Since the Babbage era (September 2022), it is important to know that an operational certificate **must be rotated using a +1 counter** (previously, it could be any value higher than the last counter) and **only if the pool has produced at least one block during the interval of 93 days**. If the pool has not produced any blocks during this period of 93 days (or 62 kes-periods), the counter must be reset to its previous value before generating a new operational certificate. An example is shown at the end of this section.

+ `cardano-cli` offers a query that recapitulates all that:

+

```
cardano-cli query kes-period-info --op-cert-file ./op.cert --mainnet
```

+ and replies

+

```
Operational certificate's KES period is within the correct KES period
interval
```

```
No blocks minted so far with the operational certificate at: ./op.cert
On disk operational certificate counter: 0
```

```
{
```

```
  "qKesCurrentKesPeriod": 695,
  "qKesEndKesInterval": 756,
  "qKesKesKeyExpiry": null,
  "qKesMaxKESEvolutions": 62,
  "qKesNodeStateOperationalCertificateNumber": null,
  "qKesOnDiskOperationalCertificateNumber": 0,
  "qKesRemainingSlotsInKesPeriod": 7891408,
  "qKesSlotsPerKesPeriod": 129600,
```

```
        "qKesStartKesInterval": 694
    }
```

- + Notice here that by the time we generated our first **operational certificate**, 1 kes-period passed. See the difference between **"qKesCurrentKesPeriod": 695** and **"qKesStartKesInterval": 694** !

Rotation example with a **counter** reset:

Let's assume that 62 KES periods have passed, and the pool has not produced any blocks while using the first operational certificate. At this point, it's necessary to rotate the pool's KES key by generating a new operational certificate. However, since the pool did not produce any blocks, the KES key rotation must be performed without the automatic increment in the counter, which is typically done by the **cardano-cli node issue-op-cert** command. Instead, the counter must remain the same as the previous value before generating the new operational certificate.

To revert the automatic increment of the counter, a new counter can be manually specified using the command below:

```
cardano-cli node new-counter --cold-verification-key-file ./pool.vk
--counter-value 0 --operational-certificate-issue-counter-file
./new.counter
```

We set a **--counter-value** of **0** and created the new counter file **new.counter**.

Let's see how **new.counter** looks:

```
{
  "type": "NodeOperationalCertificateIssueCounter",
  "description": "",
  "cborHex":
"820058203e9dff9346dab83c109a9da73aabf4642ebe64e0274b6a0931ee4b8d838ea30
4"
}
```

The "description" field has unfortunately been destroyed but what matters is the `cborHex` value, reset to "8200...8ea304". Do you notice a difference with the last time we looked at it?

It had a different `cborHex` – 8201…8ea304. That little integer change makes all the difference. Make sure to edit manually the `description` field of the `new.counter` to avoid getting lost later on.

To conclude, the `new.counter` can now be used to generate a fresh operational certificate. Remember to use a new pair of KES keys and ensure that the current KES period of the network is up-to-date.

17.2.5. File Summary #2

Your working directory should now contain 16 files:

1. base.addr
2. counter
3. kes.sk
4. kes.vk
5. new.counter <<< Example file that can safely be destroyed.
6. op.cert
7. payment.addr
8. payment.sk
9. payment.vk
10. pool.sk
11. pool.vk
12. stake.addr
13. stake.sk
14. stake.vk
15. vrf.sk
16. vrf.vk

Chapter 18. Certificates

Certificates are actions performed on the ledger allowing to:

- Register a stake address (and deregister)
- Register a stake pool (and deregister)
- Delegate an address to a stake pool

Address registration (and deregistration)

The stake component within a base address (`base.addr`) must be registered on the ledger before it can be used to delegate ada, receive staking rewards, or declare the pool's pledge or receive pool rewards. This is accomplished by submitting a registration certificate for the corresponding stake address (`stake.addr`) on the blockchain.

For now, let's create the registration certificate with the following command:

```
cardano-cli stake-address registration-certificate --stake-verification  
-key-file ./stake.vk --out-file stake.registration
```

A deregistration certificate for an address can also be generated using the `cardano-cli stake-address deregistration-certificate` command. This is the recommended method for stopping participation in ada staking. To incentivize users, deregistering an address refunds the 2 ada deposit paid during registration. More about this later.

Stake pool registration

Similar to a `base.addr`, a stake pool must register itself on the network

before it can receive delegation and produce blocks. The stake pool registration certificate is more complex than a base address registration because it includes detailed information about the pool's configuration, such as:

- Pool keys
- Owner(s)
- Fee structure
- Pool relays
- Metadata

Let's examine this certificate line by line, to understand its components:

```
cardano-cli stake-pool registration-certificate \  
--cold-verification-key-file ./pool.vk \  
--vrf-verification-key-file ./vrf.vk \  
--pool-reward-account-verification-key-file ./stake.vk \  
--pool-cost 340000000 \  
--pool-margin 0.02 \  
--pool-owner-stake-verification-key-file ./stake.vk \  
--pool-pledge 0 \  
--pool-relay-ipv4 xxx.xxx.xxx.xxx \  
--pool-relay-port xxxx \  
--metadata-url url-to-metadata \  
--metadata-hash hash-of-metadata \  
--mainnet \  
--out-file pool.registration
```

--cold-verification-key-file: ensures the right cold secret key signature is present when sending the certificate on-chain.

--vrf-verification-key-file: other pools will check whether the pool had the right to produce a block for a given slot.

--pool-reward-account-verification-key-file: specifies the stake.vk of the base.addr that will receive the rewards for running the pool. Only one reward account can be assigned to a stake pool.

The reward address will not be required to sign the transaction sending the certificate on-chain.

--pool-cost: the fixed cost the pool will charge before calculating the margin fee. It cannot be set lower than 340 ada or 340000000 lovelaces at this time on mainnet.

--pool-margin: the percentage fee taken by the pool on the remaining rewards after pool cost has been deducted from all block rewards. Its boundaries are 0 (0 %) and 1 (100%). In this example, it is set at 2%

--pool-owner-stake-verification-key-file: specifies the `stake.vk` of the `base.addr` used as the pledge for the pool. While it can be the same as the reward account, a different address may be chosen. Multiple base addresses can be used for pledging.

The transaction sending the certificate will include a signature for each and every address referenced as a pool owner.

--pool-pledge: defines the minimum amount in lovelace that must collectively be present in the owner(s) account(s). If this requirement is not met, the pool will forfeit all rewards, both for delegators and the pool itself. To ensure flexibility or avoid risks in this example, it is set to 0.

The address(es) declared as pool's pledge must all be delegated to the pool being registered.

--pool-relay-ipv4: the IP address of the relay node used to shield the block produced from connections to the wider network. Note that an ipv6 option exists.

--pool-relay-port: specifies the port on which the relay cardano-node will be listening.

NOTE

If more than one relay is used, these parameters can be duplicated thus:

```
--pool-relay-ipv4 IP#1 \
--pool-relay-port xxxx#1\
--pool-relay-ipv4 IP#2 \
--pool-relay-port xxxx#2\
```

Alternatively, the `--single-host-pool-relay` can be used to declare a stake pool relay's DNS name that corresponds to an A or AAAA DNS record.

```
--single-host-pool-relay dns.record \
--pool-relay-port xxxx \
```

`--metadata-url`: specifies a publicly available URL that serves the metadata for the pool. This metadata provides important information about the pool, such as its name, description, and other relevant details. Here is an example of a pool's metadata.json file:

```
{
  "name": "Pool's name",
  "description": "Example pool",
  "ticker": "EXP",
  "homepage": "https://examplepool.com"
}
```

`--metadata-hash`: a hash of the accessible metadata.json file, which ensures the file has not been tampered with. Once the metadata.json file is downloaded from the URL, the hash can be generated using cardano-cli. The command for generating the hash is as follows:

```
cardano-cli stake-pool metadata-hash --pool-metadata-file
./metadata.json --out-file ./metadata.hash
```

Explore the `cardano-cli stake-pool registration-certificate` command to know more about other options available (`ipv6` or `SRV DNS records` for example).

Stake pool deregistration

If you wish to retire a pool, you can easily create a **deregistration certificate**:

```
cardano-cli stake-pool deregistration-certificate --cold-verification  
-key-file ./pool.vk --epoch 410 --out-file pool.deregistration
```

The **--epoch** parameter specifies the desired epoch for the pool to become inactive. This epoch must be in the future, but it cannot exceed 18 months. This limit is defined by the **eMax** value in the **mainnet-shelley-genesis.json** configuration file on the Cardano mainnet.

When a pool operator sends a deregistration certificate on-chain, they will receive a refund of the 500 ada deposit paid for the initial pool registration as an incentive for deregistering the pool.

Only the pool cold keys are necessary to retire a pool. If the owner is not the pool operator, they have no authority in this process.

Delegation certificate

A **base.addr** can be delegated to a stake pool via a **delegation certificate** as follows:

```
cardano-cli stake-address delegation-certificate --stake-verification  
-key-file ./stake.vk --cold-verification-key-file ./pool.vk --out-file  
delegation.certificate
```

As expected, the delegation process requires the **stake.vk** of the address from which you wish to delegate, along with the **pool.vk** of the pool you intend to delegate to. Since the pool is operated by you, obtaining this information should not be an issue.

Do not hesitate to explore the **cardano-cli stake-address delegation-certificate** command to learn how to delegate to another stake pool for which you are not the operator (ie, you do not possess the corresponding

`pool.vk`).

File Summary #3

Your working directory should now contain 21 files:

1. `base.addr`
2. `counter`
3. `delegation.certificate`
4. `kes.sk`
5. `kes.vk`
6. `new.counter` <<< Example file that can safely be destroyed.
7. `metadata.hash`
8. `op.cert`
9. `payment.addr`
10. `payment.sk`
11. `payment.vk`
12. `pool.deregistration` <<< Example file that can safely be destroyed.
13. `pool.registration`
14. `pool.sk`
15. `pool.vk`
16. `stake.addr`
17. `stake.registration`
18. `stake.sk`
19. `stake.vk`
20. `vrf.sk`
21. `vrf.vk`

Congratulations! We are almost done!

Now that all addresses, keys, and certificates are in your possession, you can interact on-chain (see the Transactions section) and announce your presence in the Cardano network.

Chapter 19. Queries

Before building transactions, familiarize yourself with node queries. One cardano-cli query function has already been encountered: checking the KES period information of the network.

All query commands can be listed like this:

```
cardano-cli query
```

Rather than reviewing all available queries, let us focus on some useful ones for the upcoming ‘Basic transactions’ section. Feel free to explore other queries independently.

Protocol parameters

First, retrieve the protocol parameters and save them in a `pparameter.json` file:

```
cardano-cli query protocol-parameters --mainnet > pparameters.json
```

`pparameter.json` includes a detailed list of smart contract-related costing models and essential information for estimating transaction costs. For our purposes, we will focus on the transaction cost estimation details.

UTxOs in an address

To manually build the transaction that will post the previously created certificates, we must identify which UTXO to use. A UTXO is uniquely identified on-chain by the combination of a transaction hash (TxHash) and a transaction index (TxIx). An address can hold multiple UTXOs, making it

essential to determine the specific UTXO for this transaction.

We can access the UTxO(s) of an address as follows:

```
cardano-cli query utxo --address  
addr1q9m8pcfxszyvx7gytqc2s91400aund8z7sazfs2jtfy4h3ft07r3hw6744htxg2gtcd  
7t28j9tu2s0wlqpm1lyj29nwssucyxn --mainnet
```

Amount	TxHash	TxIx
-----	-----	-----
0a0043122fb4913b8694bb0b0af7d0c65130d2787ced56bf61bc6ba2fcf5f211 5000000 lovelace + TxOutDatumNone		0

For demonstration, the address generated in the first section of this tutorial has been funded with five ada, or 5,000,000 lovelaces. While this amount is insufficient for the subsequent steps, it provides a practical example.

Slot height of the network

Cardano transactions have an expiry date, which the user can define. To set this expiry, one must first determine the [Cardano time](#) expressed in [slot](#) height:

```
cardano-cli query tip --mainnet
```

```
{  
  "block": 8668162,  
  "epoch": 406,  
  "era": "Babbage",  
  "hash":  
  "cf5902001ba7024b07c999421804a77b6bf7858c2298e7ead1c5732a6697bcc7",  
  "slot": 90368116,  
  "syncProgress": "100.00"  
}
```

Chapter 20. Basic transaction

In this section we will create a single transaction that will post the `stake.registration` of the `base.addr`, the `pool.registration` and the `delegation.certificate` that were generated in the previous sections. All in one go.

Estimate the transaction fee

We will first create a dummy transaction (`tx.draft`) to estimate the transaction fees.

```
cardano-cli transaction build-raw \
--tx-in
0a0043122fb4913b8694bb0b0af7d0c65130d2787ced56bf61bc6ba2fcf5f211#0 \
--tx-out
addr1q9m8pcfxszyvx7gytqc2s91400aund8z7sazfs2jtfy4h3ft07r3hw6744htxg2gtcd
7t28j9tu2s0wlqpmillyj29nwssucyxn+0 \
--invalid-hereafter 0 \
--fee 0 \
--certificate-file ./stake.registration \
--certificate-file ./pool.registration \
--certificate-file ./delegation.certificate \
--out-file tx.draft
```

Because it is a `tx.draft` all values are set to `0`.

- `--tx-in`: the UTxO that will be consumed in the format `TxHash#TxIx`. Nothing prevents you from consuming more UTxO! Use additional `--tx-in` lines to do so.
- `--tx-out`: the address where ada change will be sent back to. Nothing prevents you from specifying more than one address! Use additional

- tx-out lines to do so.
- --invalid-hereafter: the transaction will be valid until this slot height is reached.
 - --fee: the transaction fee of what we want to calculate!
 - --certificate-file: adding a certificate to the transaction.

Once we have the `tx.draft`, we can calculate the fees.

```
cardano-cli transaction calculate-min-fee \  
--tx-body-file tx.draft \  
--tx-in-count 1 \  
--tx-out-count 1 \  
--witness-count 3 \  
--mainnet \  
--protocol-params-file pparameters.json
```

You may adjust the `--tx-in-count`, `--tx-out-count`, and `--witness-count` values accordingly. Here, we consume one UTxO, have the change sent back in a single address and will sign the transaction with 3 witnesses (a.k.a. secret keys).

The command replies :

```
197313 Lovelace
```

Perfect! Only 0.197313 ada. Not too expensive for a transaction containing three certificates.

Note: Transaction fees are deterministic and as such, have a lower bound. However, users can always choose to pay more than the required amount. It is important to handle fee inputs with care.

Build the final transaction

```
cardano-cli transaction build-raw \  
--tx-in UTx0_TxHash#TxIx \  
--tx-out
```

```
--tx-out $(cat base.addr)+value \
--invalid-hereafter 90369116 \
--fee 197313 \
--certificate-file ./stake.registration \
--certificate-file ./pool.registration \
--certificate-file ./delegation.certificate \
--out-file tx.final
```

- **--tx-in** `UTxO_TxHash#TxIx: TxHash#TxIx` of the UTxO you want to consume!
- **--tx-out** `$(cat base.addr)+value`

IMPORTANT

--tx-out value is expressed in lovelaces. * Registering an address requires a deposit of 2 ada (2000000 lovelaces). * Registering a pool requires a deposit of 500 ada (500000000 lovelaces). * This means that the change to the `base.addr` will be : ***value = input - (502 deposit + transaction fee)*** * Deregistration of an address or a pool will be accounted for by adding 2 ada or 500 ada, respectively, to the ada change value one must calculate to correctly balance a transaction! For example, to deregister simultaneously a base address and a pool: ***value = (input + 502 deposit) - transaction fee***

- **--invalid-hereafter**: The slot tip of the network plus some slots to give your transaction time to get accepted by the network. Here, 1000 slots (seconds on mainnet) or ~ 15 minutes in the future, from the last query we made at "slot": `90368116`.
- **--fee**: the exact value we calculated earlier.
- **--certificate-file**: adding a certificate to the transaction.

Note: The order in which the certificates are declared will matter in the

final transaction. You cannot delegate to a pool that does not exist yet. You cannot register a pool with an owner's address that is not registered yet. Hence, we register the `base.addr` (its staking part) first, then the pool and finally delegate to it.

Sign and send a transaction

```
cardano-cli transaction sign \
--tx-body-file ./tx.final \
--signing-key-file ./payment.skey \
--signing-key-file ./stake.skey \
--signing-key-file ./pool.skey \
--mainnet \
--out-file tx.final.signed
```

The transaction must be signed by three private keys in this case.

- `payment.skey` will authorize spending funds from the `base.addr`.
- `stake.skey` will authorize the use of `base.addr` (its staking part) as a pool owner and authorize delegation to the pool at the same time.
- `pool.skey` will authorize the registration of the pool

Note: The address used for collecting pool rewards does not need to sign this transaction. In this case, the same address is used for both reward collection and pledge functions. The pool owner (pledge) is not required to sign a `de-registration certificate`. Updating pool parameters is done by sending a new `pool-registration certificate`, and the deposit is only required during the initial registration.

```
cardano-cli transaction submit \
--tx-file tx.final.signed \
--mainnet
```

Congratulations ! It is all done, the pool is now registered! With some delegated stake, it will sign blocks and pay rewards to the `base.addr` (once the block-producing node is started with the appropriate keys).

20.1. Rewards withdrawal

As the pool grows in saturation, it will eventually mint blocks and start accumulating ada rewards. These rewards, however, reside in the stake account associated with the base.addr and are not immediately available as UTXOs.

To access these rewards, let's query a random address that has accumulated ada:

```
cardano-cli query stake-address-info --address  
stake1u97v0sjx96u5lydjfe2g5qdwkj6plm87h80q5vc0ma6wjpq22mh4c --mainnet
```

```
[  
 {  
   "address":  
 "stake1u97v0sjx96u5lydjfe2g5qdwkj6plm87h80q5vc0ma6wjpq22mh4c",  
   "delegation":  
 "pool1kchver88u3kygsak8wgll7htr8uxn5v35lfrsyy842nkscrzyvj",  
   "rewardAccountBalance": 370751053  
 }  
 ]
```

This address contains **370751053** lovelaces or **370.751053 ada** rewards.

A withdrawal transaction can be created to convert these ada rewards into spendable UTXOs.

First, let's estimate first the transaction fee for this:

```
cardano-cli transaction build-raw \  
--tx-in  
0a0043122fb4913b8694bb0b0af7d0c65130d2787ced56bf61bc6ba2fcf5f211#0 \  
--tx-out $(cat base.addr)+0 \  
--withdrawal $(cat stake.addr)+0 \  
--invalid-hereafter 0 \  
--fee 0 \  
--out-file withdraw.draft
```

--withdrawal: specifies from which `stake.addr` rewards will be withdrawn from; `+` separates the address from the value withdrawn in lovelace.

Since we only create a fake transaction in order to calculate transaction fees, we set the value withdrawn at 0.

Next, run:

```
cardano-cli transaction calculate-min-fee \  
--tx-body-file withdraw.draft \  
--tx-in-count 1 \  
--tx-out-count 1 \  
--witness-count 2 \  
--mainnet \  
--protocol-params-file pparameters.json
```

178525 Lovelace

Almost done. Let's craft the real withdrawal transaction now.

```
cardano-cli transaction build-raw \  
--tx-in  
0a0043122fb4913b8694bb0b0af7d0c65130d2787ced56bf61bc6ba2fcf5f211#0 \  
--tx-out $(cat base.addr)+375572528 \  
--withdrawal $(cat stake.addr)+370751053 \  
--invalid-hereafter 90455278 \  
--fee 178525 \  
--out-file withdraw.draft
```

All values are expressed in lovelaces.

- The `base.addr` will receive as change : `UTx0_value + rewards_withdrawn - transaction_fees`
- Rewards are withdrawn in full. Partial withdrawals are not allowed.
- Additional `--tx-out` fields can be added, provided the transaction remains balanced (total input -

IMPORTANT

transaction fees = total output).

We can now sign the transaction:

```
cardano-cli transaction sign \
--tx-body-file ./withdraw.draft \
--signing-key-file ./payment.skey \
--signing-key-file ./stake.skey \
--mainnet \
--out-file withdraw.signed
```

Two witnesses are required:

- The **payment.sk** of the **base.addr** that pays for the transaction fee.
- The **stake.sk** of the **stake.addr**, to allow the withdrawal of ada rewards.

Send the transaction:

```
cardano-cli transaction submit \
--tx-file tx.final.signed \
--mainnet
```

Chapter 21. Epilogue

This guide details all the necessary `cardano-cli` actions to become an autonomous stake pool operator on Cardano. Hopefully, this document has clarified the numerous components involved.

Community tools, like [Guild-Operators](#) or the [Stake Pool Operators Scripts](#) repositories, abstract most of the steps described. Additionally, operations requiring a running instance of `cardano-node` (to query or post on the ledger) can be performed using a copycat of `cardano-cli` called `blockfrost-cardano-cli`. It can be sometimes faster than querying your local `cardano-node` instance!

Given the importance of handling private keys securely, it's recommended to use tools that integrate hardware wallets. One such tool is `cardano-hw-cli`, a version of `cardano-cli` designed to manage both addresses and pool private keys via hardware wallets [here](#).

Note: *It is essential to have a clear understanding and hands-on experience with these tools before use. For first-time users, practicing on testnets is highly recommended to avoid any mistakes.*

Chapter 22. Keeping Time

Ouroboros is the protocol powering Cardano, symbolized by the eternal serpent consuming its own tail. Until [Ouroboros Chronos](#) is implemented, the ‘timeless’ Ouroboros relies on stake pool operators to help maintain accurate time.

Clock synchronization is essential in distributed networks. Nodes must prevent clock drift, which occurs when they measure time at slightly different rates. In Ubuntu and other Linux distributions, various network time protocol (NTP) programs help reduce time synchronization offsets to a few milliseconds. One commonly used program is Chrony, which should be installed and running on the node or relay for Cardano. On Ubuntu, Chrony can be installed using `apt install chrony`.

After installing Chrony, it is recommended to configure the default configuration file (`/etc/chrony/chrony.conf`) with nearby, high-performance NTP servers. These servers should be polled frequently to minimize drift from the global clock. NTP servers are categorized by stratum levels, where:

- **Stratum 0:** reserved for atomic clocks or other highly accurate time sources
- **Stratum 1:** suitable for systems that can deviate only within a few milliseconds of stratum 0
- Higher strata indicate further deviations from precise time.

To check the stratum level, use the command `chrony ntpdata`. Below is an example of a typical `chrony.conf` configuration:

```
pool 192.168.2.100 minpoll 1 maxpoll 2 maxsources 1
pool time.cloudflare.com minpoll 3 maxpoll 4 maxsources 1
pool time.google.com minpoll 3 maxpoll 4 maxsources 1
maxupdateskew 5.0
makestep 0.1 -1
rtsync
leapsectz right/UTC
```

22.1. Comments on the example configuration :

- To add NTP servers, include their IP address (eg, 192.168.2.100) or DNS name (eg, time.google.com) in the chrony.conf file.
- Setting lower values for minpoll and maxpoll reduces time drift; maxsources sets the maximum number of sources used from the pool.
- Public timesync servers provided by companies like Cloudflare (time.cloudflare.com), Google (time.google.com), and Facebook (time.facebook.com) are generally reliable and high-performing. The [NTP pool project](#) also lists NTP servers worldwide for broader options. Running a local timesync server is another option to encourage decentralization. [Guides are available](#) for setting one up using a Raspberry Pi.
- In the example configuration, a local Stratum 1 GPS source (192.168.2.100) on a LAN is used with a fast polling rate. Additionally, two public NTP servers (Stratum 3 Cloudflare and Stratum 1 Google) are included with slower polling rates. Be cautious with overly-aggressive polling rates to avoid being blacklisted by public timesync servers.
- On modern fiber connections near the timesync server, it is recommended to use a lower value for Chrony's internal estimate of how fast or slow the computer clock runs relative to the timesync server. Setting maxupdateskew to 5.0 (down from the default 1000 ppm) helps maintain higher precision in time synchronization.
- Chrony can adjust system clocks more frequently by setting a lower

`makestep` value. For example, `makestep 0.1 -1` adjusts the clock whenever a drift of 0.1 seconds occurs. Network Interface Cards (NICs) with an internal clock for timestamps (which can be identified using `ethtool -T`) can also be enabled for hardware timestamping with `hwtimestamp` and the interface name or `*` for all interfaces.

- Drift may be checked with `chronyc tracking` and `chronyc sourcestats` to view offsets and skew.
- Rtsync can be set to enable the kernel to periodically synchronize the system clock with the real-time clock, typically every 11 minutes on Linux. If the connected timesync server does not provide leap second updates in advance, the `leapsectz` option (e.g., `leapsectz right/UTC`) may be configured. In general, it is recommended to use Coordinated Universal Time (UTC) for server time, as this is the global standard and the format most GPS-based timesync servers report in.

With that out of the way in a fairly short time (pun intended), a stake pool operator will be able to keep Cardano nodes collaborating well with the rest of the network in a timely manner!

Chapter 23. Server Security and Hardening

Security is a state characterized by the absence of fear or anxiety regarding one's physical, economic, technological, or social well-being, or that of those under one's care. In information technology (IT), achieving this requires implementing preventive measures, enforcing security policies and procedures, and conducting regular security assessments to identify and address vulnerabilities. IT security is critical today, with the widespread use of electronic communication, by protecting systems, networks, and the data they handle, and preventing data breaches.

This section addresses the security considerations for becoming an SPO. We will begin with general system administration security principles, followed by a focus on Cardano node security, containerized environments, secure shell (SSH) configuration, the advantages of using a virtual private network (VPN), and conclude with final thoughts on stake pool security.

23.1. System Administration security

Outlined here are common best practices in IT server administration for any project.

Note: All examples listed here are suggestions for commands you can use on the Unix operating system (OS). Please conduct your own research (DYOR) or refer to the command-related manual (RTFM) for verification.

23.1.1. Best practice:

- Use a secure operating system with long-term support to ensure access to the latest security updates and patches.
- Start with a clean, minimal setup for the operating system hosting your Cardano environment.
- Ensure that your day-to-day user account and cloud provider accounts utilize secure authentication methods, such as strong passwords and two-factor authentication (2FA).
- Whenever possible, configure your server with two public IP addresses: one for administration (SSH and/or VPN) and another for the services you want to expose. This approach minimizes targeted attacks on your administration pipelines and restricts public exposure to only those services you choose (commonly referred to as ‘minimizing the attack surface area’).
- Exercise caution during system administration. Always have a plan, document it, and test it before implementing changes in the production environment. An automated reporting system, such as Prometheus/Grafana, is advisable. Additionally, develop a recovery plan for potential incidents.
- Avoid running services as root whenever possible, as this increases the risk of unauthorized access and data breaches.
- Monitor exposed services, as they can be targets for cyberattacks. Be aware of the services running on your server and take steps to secure them. It is advisable not to leave any unused ports open, as they can provide unauthorized access. Disable any unnecessary open ports or services, or filter them using appropriate firewall rules. You can list open ports and their associated services using commands like: `sudo netstat -latupen`, `sudo lsof -i`, `sudo ss -antlp`. Some of these commands will also list services listening on 127.0.0.1, which is on the loopback interface (lo).

- Implement Fail2Ban to block unauthorized access attempts and denial-of-service (DoS) attempts. This tool helps block or delay persistent and/or malicious IP addresses, such as port scanners, crawlers, and those using brute-force techniques, from accessing your server.
- Change your server DNS to use known secure DNS (ie, 1.1.1.1 and 1.0.0.1)
- Use a hardware or software firewall on your server to restrict ports and allow only necessary traffic. Firewall rules can mitigate DoS attacks and port flooding by implementing connection limit rules; refer to iptables-extensions(8) for examples. Consider the firewall as a backup mechanism after assessing the open services. While a firewall can help prevent further administrative incidents, it may also increase complexity and require additional attention.
- Keep your server's operating system and installed software up-to-date to ensure that you have the latest security patches and fixes.
- Disable IPv6 on your server if it is not in use, as it can serve as a potential attack vector. Use the following command to edit the configuration file: `nano /etc/sysctl.conf`. Then, add the line: `net.ipv6.conf.all.disable_ipv6 = 1`.
- Limit the programs with suid or sgid bit set. You may list them all with:
`$ sudo find / -type f \(| -perm -4000 -o -perm -2000 |) -exec ls -l {} \\;`
- Conduct a vulnerability scan using a tool like Nmap to protect against known vulnerabilities. While it may not offer the precision or comprehensiveness of a commercial scanner, it can still provide valuable insights.
- For those using systemd services, check security with `systemd-analyze` security. You can harden any UNSAFE or EXPOSED service by creating an `override.conf` file in the `/etc/systemd/system/unit.service.d/` directory (where unit is the name of the systemd service). Directives

such as ProtectHome, ProtectSystem, and ProtectHostname are detailed in `systemd.exec(5)`. Use the command `systemctl edit unit.service` to create the override.

- A quick and easy way to monitor the overall health, load, and security of your system is to use a terminal multiplexer like tmux, combined with the tmuxp session manager to save and reproduce tmux sessions from YAML or JSON files. This lightweight and efficient method provides secure remote access via SSH and serves as a text-based alternative or complement to Grafana. Useful commands for this setup include `dmesg`, `journalctl`, `iptables`, `tload`, `htop`, and `systemctl`, as well as combinations of `watch`, `tail`, and `grep`.

23.2. Node Security:

- It is highly recommended not to operate both the block producer and a public relay on the same host. Since the relay is publicly exposed, if it becomes compromised or is under attack (eg, DoS), your block producer may also go offline, leading to missed block production opportunities, which can negatively impact your ranking and associated rewards. To mitigate this risk, secure the block producer behind a firewall, allowing it to connect only to the relay nodes. The relay nodes can then connect to the block producer and other peer relays.
- It is recommended to only store the essential files (such as `kes.skey`, `vrf.skey`, etc) on your server for running the Cardano node. It is also strongly recommended to keep other files related to your wallet/pool, such as private keys, on a separate, secure, and preferably air-gapped device.
- Additionally, it is important to avoid running any programs, including `cardano-node` and `db-sync`, with root privileges. Instead, create a non-privileged account and use it for these tasks, and this applies to both non-containerized and containerized environments.
- It is always advisable to use a hardware wallet for your pool as it will

keep your private keys safe and secure.

- Proper server security is crucial to ensure the availability, integrity, and confidentiality of your blockchain network. Segregating different functionalities and roles across separate servers, along with using unprivileged accounts, can help reduce the risk of unauthorized access and data breaches.
-

23.3. Containerized Environments

Another option is to use containerized environments, which allow Cardano's software to run in logical separation from the hosting server, thereby containing potential breaches. Utilizing such software to ensure proper configuration also increases the level of IT knowledge.

Containerized environments provide several advantages from a security point of view:

- **Isolation:** containers are isolated from each other and from the host system, meaning that a security vulnerability or attack in one container will not impact other containers or the host.
- **Least privilege:** each container runs with a specific set of permissions, reducing the risk of privilege escalation. This means that a malicious container will only have access to the resources it needs to function.
- **Segmentation:** allows the creation of multiple isolated networks for different containers, enhancing the separation of services and reducing the attack surface.
- **Patching:** allows for quick and easy patching of vulnerabilities in an application, without the need to patch the entire system.
- **Auditing:** provides detailed information about the container environment, including system calls, network connections, and file access. This information can be used to identify potential security issues and track down the source of a security incident.

- **Security scanning:** provides a security scanning feature; several third-party security scanning tools can also be used to check images for vulnerabilities.
- **Sandboxing:** containers run in a sandboxed environment – any malicious activity is restricted and the host system is not affected.
- **Control over the environment:** enables consistent and predictable application execution.

Please note that while containerized environments enhance security, they are not a complete solution. Proper configuration and security measures within the containers are essential to ensure system safety.

A plethora of options exist that provide similar containerization functionality:

- LXC (LinuX Containers), Docker, rkt (Rocket), OpenVZ, LXD (built on top of LXC), Kubernetes, Mesos.

Note that while these alternatives provide similar functionality, they may have different architectural designs and may require different configurations and management processes. It is essential to evaluate the needs of your organization and compare each solution before making a decision.

23.3.1. Securing SSH

SSH is a fundamental service for remote server access. Proper configuration and security are essential to prevent unauthorized access. It is also advisable to implement a plan for monitoring and auditing SSH access (eg, using Grafana or Fail2Ban) and to apply regular security updates and patches.

Key security considerations for securing SSH include:

1. Use a different port than the default 22/tcp, such as a port over 10000/tcp for added security. Example: `nano /etc/ssh/sshd_config` and change the line `Port 22` to `Port 10022`
2. Disable access to root login on the SSH config and only use unprivileged account(s) with SUDO access. Example: `sudo nano /etc/ssh/sshd_config` and change the `PermitRootLogin` to `no`
3. Use port knocking (`knockd`) to add an additional layer of security to your SSH connections.
4. Disable (or fake) banners to prevent displaying unnecessary information that could potentially be used by attackers. Example: `nano /etc/ssh/sshd_config` and change the line `Banner /etc/issue.net` to `#Banner /etc/issue.net`
5. Define a number of max concurrent sessions to limit the number of simultaneous connections to your server. Example: `nano /etc/ssh/sshd_config` and add the line `MaxSessions 10`
6. Add it to your fail2ban instance to delay any attempt at brute-forcing or password guessing. Example: `nano /etc/fail2ban/jail.local` and add the section `[sshd]` `enabled = true`
7. Disable password login in SSH. It is recommended not to use username/password authentication for SSH, instead use certificate-based (PKI) authentication for added security. Don't forget to password protect your private key when you generate it. Example: `ssh-keygen -t rsa`
8. When applicable, restrict access to the only IP address you'll be using to connect to your server. Example: `nano /etc/ssh/sshd_config` and add the line `AllowUsers user@xxx.xxx.xxx.xxx`
9. Use SFTP or SCP to transfer files (over SSH) to and from your server, as it provides an added layer of security compared to FTP.
10. You can use SSH tunneling to encrypt and redirect traffic within remote nodes, in order to access backend infrastructure.

23.3.2. Use a VPN (as alternative to ssh tunneling)

It is beneficial for SPOs to use a virtual private network (VPN) (ie, WireGuard) to access a server backend (such as Grafana or Prometheus) due to enhanced security and accessibility. A VPN establishes a secure and encrypted connection between the client device and the server backend, protecting transmitted data from potential eavesdropping or tampering.

This protection is especially important when accessing sensitive information like financial data, personal details, or confidential blockchain data. Additionally, a VPN enables remote access to the server backend, allowing SPOs to connect from anywhere with an internet connection. This capability is particularly useful for SPOs managing nodes remotely or across different locations. Furthermore, using a VPN can help bypass geographical restrictions or censorship. Overall, employing a VPN to access a server backend effectively secures and facilitates access to sensitive information and resources for SPOs.

Chapter 24. Monitoring

Monitoring is a crucial part of stake pool operation. It helps minimize the chance for lost blocks and can allow the operator to maximize rewards for delegators along with contributing to strong network performance on the Cardano blockchain.

- Reliability: Monitoring helps ensure the full-time availability of a stake pool to mint and propagate assigned blocks in a reliable manner. By diligent monitoring of key metrics such as blockheight, uptime, timesync, KES expiry period, unadopted blocks, missed slots, etc. patterns of Stake Pool service disruption can be detected and prevented.
- Capacity Planning: By analyzing historical resource usage data, future resource requirements can be forecast to anticipate capacity constraints and allow the operator to scale infrastructure proactively to meet evolving Cardano Stake Pool operation needs, thereby avoiding performance degradation or service outages due to insufficient resources.

24.1. Prometheus

Prometheus is an open-source monitoring and alerting toolkit. It is designed for reliability, scalability, and simplicity in monitoring complex IT environments. Prometheus is often recommended for scraping Cardano Stake Pools metrics for several reasons:

- Powerful Data Model and Query Language: Prometheus employs a multi-dimensional data model and PromQL (Prometheus Query

Language) for querying and analyzing time-series data. This model allows for flexible and efficient querying of metrics based on various dimensions such as labels, enabling operators to gain deep insights into the performance and health of an operator's stake pool infrastructure. With PromQL, operators can perform complex aggregations, transformations, and statistical analysis on time-series data, making it easier to identify trends, anomalies, and patterns in monitored metrics.

- **Scalability and Reliability:** Prometheus is designed to be highly scalable and reliable, capable of handling large-scale deployments with thousands of servers, containers, or microservices. It employs a pull-based model, where Prometheus servers periodically scrape metrics from instrumented targets, providing a scalable approach to monitoring dynamic environments. Additionally, Prometheus supports federation, allowing multiple Prometheus servers to aggregate and federate metrics from different sources, enabling horizontal scalability and distributed monitoring setups. Its robust architecture and proven reliability make it suitable for mission-critical IT environments where uptime and performance are paramount.
- **Rich Ecosystem and Integration:** Prometheus has a vibrant ecosystem with a wide range of integrations and exporters for collecting metrics from various systems, applications, and services. It supports integrations with popular technologies such as Kubernetes, Docker, AWS, and more, making it easy to monitor modern cloud-native environments and microservices architectures. Additionally, Prometheus integrates seamlessly with other tools in the monitoring ecosystem, such as Grafana for visualization, Alertmanager for alerting, and third-party storage solutions like Thanos for long-term storage and high availability. This rich ecosystem of integrations and complementary tools enhances the flexibility and extensibility of Prometheus, making it a versatile choice for monitoring diverse IT environments.

For Cardano Stake Pools these are typical targets to scrape for Prometheus:

- The Cardano-node built-in EKG Prometheus exporter with Cardano node metrics on blockheight, missed-slots, KES expiry and more.
- The open source [node_exporter](#), an open source Prometheus exporter for hardware and OS metrics exposed by *NIX kernels for metrics on CPU, memory, disk usage, timesync and others.
- [Cardano Guild Operators Koios SPO Tools](#) to expose [CNCLI](#) Prometheus metrics on block.db metrics such as number of sequentially missed blocks, adopted blocks, epoch luck, etc.

The Prometheus scrape targets are defined via a `.yml` configuration file (typically named `prometheus.yml`). To double check syntax, you can run `./promtool check config prometheus.yml`. Before adding a visualization layer to Prometheus like Grafana it is recommended to connect to the Prometheus web portal (by default on localhost:9090) to double check that scraping targets are reachable. Check that the status is green and `up` on all configured endpoints in [Status → Targets](#). This status page is very helpful to debug scrape endpoint connections.

To scrape remote servers securely various techniques exist, see also [Server Hardening](#). For example, one could use a web server such as Nginx and configure it as a Reverse Proxy for the endpoint metrics to be scraped by Prometheus with Transport Layer Security (TLS) encryption in place. You can use TLS for IPs. You can also scrape Prometheus over a WireGuard VPN or other VPN service. Firewall rules should ensure that only the Prometheus monitoring server may access the remote endpoint.

Note: Prometheus and Grafana server should be run separately from the block producing node, to avoid competition of compute resources, and disruption of block production. Ideally, a separate monitoring host should be set up. If a separate host is not available, a passive relay host may be used.

24.2. Grafana

Grafana is an open-source platform for monitoring and observability, specializing in data visualization and analytics. It allows operators to create dashboards, charts, and graphs to visualize and analyze metrics from various data sources. It is an ideal companion to display Prometheus metrics. Stake pool operators generally opt to use Grafana alongside Prometheus for the following reasons:

- **Flexible Visualization and Dashboards:** Grafana provides a highly flexible and customizable platform for visualizing metrics and building dashboards. It offers a wide variety of visualization options, including graphs, gauges, tables, heatmaps, and more, allowing operators to create rich, interactive dashboards tailored to their specific monitoring needs. Grafana supports multiple data sources, including Prometheus, Graphite, InfluxDB, Elasticsearch, and many others, enabling operators to consolidate metrics from different sources into a single unified dashboard for comprehensive monitoring and analysis.
- **Extensibility and Integration:** Grafana is highly extensible and supports integration with a vast ecosystem of data sources, plugins, and extensions. It offers a plugin architecture that allows developers to create custom data source plugins, panel visualizations, and integrations with third-party services. This extensibility enables Grafana to adapt to diverse monitoring environments and integrate seamlessly with existing tools and systems. Additionally, Grafana supports features such as annotations, templating, and alerts, enhancing its functionality and making it a versatile platform for monitoring IT services.
- **Community and Adoption:** Grafana has a large and active community of users, developers, and contributors, driving innovation and adoption in the monitoring space. The Grafana community has developed a wide range of plugins, dashboards, and integrations, which are freely available through the Grafana Plugin Repository and community

forums. This vibrant ecosystem of community-contributed content provides operators with access to a wealth of resources and pre-built solutions for monitoring various technologies, applications, and infrastructure components. Additionally, Grafana's popularity and widespread adoption make it a de facto standard for visualization and monitoring in many organizations, ensuring long-term support, stability, and continued development of the platform.

Some Prometheus exporters like `node_exporter` come with their own pre-configured Grafana dashboard. `Cardano-node` does not have a default Grafana dashboard yet, but the community has created various dashboards and shared those online to copy and adapt or for inspiration.

Grafana dashboards can be easily configured with a graphical drag & drop interface or editing the JSON configuration file.

24.3. Alerting with Prometheus and Grafana

Effective alerting options exist within Prometheus and Grafana, and alerts may be configured with either service:

- **Prometheus Alertmanager:** Alertmanager is a component of the Prometheus monitoring system responsible for handling alerts generated by Prometheus servers. It receives alerts from Prometheus via its alerting rules and then performs actions based on those alerts, such as sending notifications to various alerting channels (e.g., email, Slack, PagerDuty, Telegram, xMatters). Alertmanager focuses on managing and routing alerts efficiently, ensuring that the right notifications reach the appropriate recipients according to defined alert routing and suppression rules. Alertmanager is tightly integrated with Prometheus and is primarily designed to work with Prometheus-generated alerts. It provides native integration with Prometheus's alerting rules and relies on Prometheus's pull-based model for collecting metrics. While Alertmanager can integrate with other

monitoring systems and services via webhooks and APIs, its primary focus is on handling alerts generated by Prometheus. The Prometheus Alertmanager is configured via YAML-files.

- **Grafana Alerting:** Grafana Alerting is a feature built into the Grafana platform that enables operators to create and manage alerts directly within the graphical Grafana dashboards. It allows operators to define alert conditions based on query results from data sources and configure alert notifications to be sent via various channels (e.g., email, Slack, PagerDuty, Telegram, xMatters) Grafana Alerting is tightly integrated with Grafana's visualization and dashboarding capabilities, enabling operators to create rich, interactive dashboards with embedded alerts and seamlessly transition between monitoring and alerting workflows within the Grafana interface. Grafana Alerting offers integration capabilities with various data sources and external systems. Grafana Alerting allows operators to define alert conditions based on data queries from diverse data sources, enabling flexible and customizable alerting workflows tailored to specific monitoring environments. Additionally, Grafana Alerting can integrate with external notification services and platforms, providing users with a wide range of options for alert notification delivery.

Thresholds can be defined for each relevant monitoring metric to determine when alerts are triggered. Useful alerts include:

- Last blockheight: Did the block producer lag behind the anticipated blockheight?
- Disc usage: Is disk space running out?
- Missed or ghosted blocks: Did the block producer miss minting or propagating a sequential number of assigned blocks?
- Missed slot leadership checks: Did the block producer miss checking leadership eligibility for a large number of slots?
- Available memory: Is memory usage too high and potentially affecting

node performance?

- Endpoint availability: Are all relevant endpoints for scraping available?
- KES expiry: How much time is left before KES keys expire?
- Time synchronization: Is the server time out of sync?

For advanced alert routing and communication, cloud platforms such as xMatters or PagerDuty can be added as another layer between the generated alerts from either Prometheus or Grafana with the potential benefits:

- Reducing MTTR (mean time to respond) by suppressing redundant alerts and only relay the most critical insights to on-call resolvers.
- Customizing alert data and response actions to eliminate manual work.
- Allowing resolvers to pause and resume. An example being Grafana alerts directly from xMatters notifications as resolution is reached.

Note: xMatters has a free tier that works for both Prometheus Alertmanager and Grafana Alerting.

24.4. Zabbix

[Zabbix](#) is another integrated, all-in-one monitoring solution with out-of-the-box capabilities for monitoring diverse IT components and can be configured to monitor Cardano Nodes. Prometheus and Grafana do offer more scalability, flexibility, and customization options, but Zabbix is an alternative.

24.5. RTView

[RTView](#) is a real-time monitoring program that provides visibility on the state of running Cardano nodes. It supports multiple node monitoring, even if the nodes work on different machines.

The main benefit of RTView is simplicity. It is simple to use; technically

there is no installation, you just unpack an archive and run an executable. It is also simple to configure through an interactive dialog and shows particular changes the user should make in the node configuration files.

RTView does render a webpage dashboard and offers a less-complex but robust alternative to a well-configured Prometheus and Grafana monitoring setup.

24.6. Koios gLiveview

[Koios gLiveView](#) is a local monitoring tool to use in addition to remote monitoring tools like Prometheus and Grafana, Zabbix or RTView. This is especially useful for systemd deployments as it provides a terminal UI to monitor real time node status for Stake Pool operators.

Note: If Koios gLiveView is launched on a block producer with an up-to-date block.db the tool will show block minting metrics and real-time left until the next block.

24.7. Manual Cardano Log Review

For Stake pool operators it is important to be able to read and interpret logs from cardano-node to troubleshoot and find potential issues. It is best practice to investigate each missed block and determine what went wrong. Long-term luck should be near 100% and while block collisions within the same slot occur, they are uncommon. About 5% of all blocks result in slot battles, as the outcome of a slot battle is random, about 2.5% of blocks are expected to be lost over time.

Ideally, an operator will investigate issues that affect block production within a stake pool and make adjustments to ensure smooth and reliable operation moving forward.

This is a typical debugging sequence:

- Determine the slot number missed from the calculated leaderlog.

- Was more than one block missed in sequence? If so, this is likely a serious issue such as an improperly started node, expired KES keys, or hardware or server failure.
- If a single block was lost, it is time to debug.
- Search the node log for the slot number it missed.
- Did the block producer recognize it was its turn to mint a block (`nodeIsLeader`)?
- Did the block producer mint the block? What was the `blockheight` and `blockhash`?
- Did the block producer add this block to its local chain or did it run into an internal fork?
- Did the relays see this `blockhash` and add it to their local chains?
- `Blockheight` can also be investigated in any of the public Cardano blockchain explorers.
- Did another pool mint this `blockheight` with the same slot number? If so, this was the result of a slot collision and is normal.
- Did another pool mint this `blockheight` with a different slot number? This is considered a Height battle and is usually the result of a misconfigured pool.
 - If the block was propagated within its slot time (one second) it was very likely a race condition and the other pool was awarded the block based on a lower VRF hash. Not much can be done about these conditions at this time outside of working with the pool operators with misconfigured stake pools.
 - If block propagation took longer than the awarded slot, it needs to be investigated. It is possible that a bottleneck or misconfiguration exists, preventing speedy propagation such as poor upload bandwidth.

Pool operators may choose to assist in diagnosing network-wide issues

while, although rare, benefit from as many data-points as possible. Operators can consider contributing information about nodes to the following community services and keep informed about new and upcoming services:

- Operators may send the current chain tip of their block producer to Pooltool via [cncli-SendTip](#). This will show a green badge on Pooltool with current tip height and helps Pooltool to capture orphan blocks. While this doesn't guarantee every orphan block made will be seen by Pooltool, a large number of reported orphaned blocks across pools can help diagnose wider network issues.
 - Operators may also share the number of blocks assigned for an epoch and validate the correctness of past epochs via [cncli-SendSlots](#). This helps to debug issues on Leaderlog calculation and can increase pool performance visibility for delegators.
 - Block propagation metrics may also be sent to monitor network propagation of new blocks as seen by the local cardano-node with the [blockPerf.sh](#) script; a public dashboard is in the works to display these metrics. Pooltool may also be checked regularly and operators can compare their own block propagation times with other operators.
-

24.8. Final Thoughts

Keep the setup simple and minimize the attack surface. Being overly cautious can increase the complexity of monitoring and may isolate you from your system. Carefully manage risks until confident in your security, and always have a contingency plan.

Your laptop, air-gapped system, relay node servers, and block producer server will have different security levels based on their operational and economic value, as well as the context of use (cloud, dedicated, or bare metal, and geographical location). Develop a security plan after

considering various scenarios and making informed decisions.

Take the time to design your infrastructure before implementation. This will save time in maintenance and improvement later.

Chapter 25. Writing smart contracts

25.1. Prefaces

25.1.1. Plutus preface

Prof Philip Wadler, FRS

University of Edinburgh and IO

Smart contracts for the Cardano blockchain run in a language called Plutus. Roughly speaking, Plutus plays the role for Cardano that the Ethereum Virtual Machine (EVM) does for Ethereum. But where the EVM requires a dense forty-two-page yellow paper to define, the key concepts of Plutus can be spelled out in less than one page.

The reason for this is that Plutus is based on the lambda calculus, first defined by the logician Alonzo Church in the 1930s, and an elaboration of the lambda calculus called System F, first defined by the logician Jean-Yves Girard in the 1970s. To be fair, that claim is a bit of an oversimplification. While the core of Plutus is indeed based on those systems and can be described quite concisely, Plutus also contains a collection of built-in operators and rules for costing, which takes longer to spell out.

The reason for designing a language with such a tiny core is tied closely to the needs of the Cardano blockchain. Any significant updating of Cardano, including a change to the definition of Plutus, requires a hard fork, and it is highly desirable to avoid these. How do you design a system that can last unchanged for the next fifty years? Base your design on a system that is fifty years old!

Because this fifty-year-old theory would now be executed directly on the Cardano blockchain, carrying billions of dollars of value, there was a business case for formally validating that theory. Input | Output (IO) hired James Chapman to validate our variant of System F in the Agda proof assistant—James, having written his dissertation on a related subject, is one of the foremost experts in the field. The completed proof validated the theory. It also gave us an executable version of Plutus. Continuous Integration ensures the production Plutus implementation is regularly checked against the executable formal specification. Further, the formal specification serves as the foundation for a new effort to write a validator that issues checkable certificates to ensure that phases of the compiler optimize code correctly. James is now head of formal methods for IO.

Everything should be made as simple as possible, but not simpler. Arguably, our initial simple design is too simple. It requires representing data types as higher-order functions, but we had reason to believe the extra cost was negligible. Always check your assumptions! Once the system was up and running, performance measurements showed the cost of implementing data structures as functions was greater than expected. So, we modified the design to support data structures directly in Plutus Core, and the feature is available as of version 1.1.0.

The on-chain production evaluator for Plutus is written in Haskell, one of the most widely used functional programming languages. Plutus is designed to represent the output of a compiler, and no one should write it directly. Haskell can also be used as a source language for creating Plutus scripts, using the Plinth language embedded into Haskell. On Ethereum, one must learn the special-purpose language Solidity to write smart contracts that compile to the EVM. In contrast, on Cardano, one can use the Haskell language, designed by an expert team of researchers thirty years ago. It has a mature and stable core, even though it is still an actively growing research language. The brilliant idea to apply Haskell in this way is due to Manuel Chakravarty, a contributor to both the design and

implementation of Haskell.

In addition to Plinth (embedded in Haskell), a wide range of other languages can also be used to generate smart contracts in Plutus to run on Cardano. These include: Aiken (a stand-alone language), Plutarch (also embedded in Haskell), Plu-ts (embedded in TypeScript), OpShin (embedded in Python), Scalus (embedded in Scala), Helios (which runs on top of JavaScript and TypeScript), and Pluto (which closely resembles Plutus with added syntactic sugar for convenience). Marlowe, which is not a general-purpose language but aims to make it easy to write certain kinds of financial contracts, is also available.

Many people have contributed to the designs of Plutus and Plinth. I won't attempt to name them all here because with my absent-minded professor's memory I fear I would leave someone out. I am grateful for the opportunity I've had to work with them and with everyone at IO. I'm particularly grateful to Charles Hoskinson for the opportunity to collaborate with IO. It's been an amazing experience – it's not often that you can suggest a practical application of a fifty-year-old theory and have people take you seriously!

25.1.2. Marlowe preface

Prof Simon Thompson

University of Kent

Eötvös Loránd University

simon.thompson@marlowe-lang.org

Marlowe is a high-level smart contract language designed to describe asset flows on the blockchain. Marlowe embodies a local, account-based model and runs equally well on UTXO and account-based chains like Ethereum. Marlowe describes fully distributed interactions between contract participants, mediated by the underlying blockchain.

Contracts written in Marlowe automatically provide safety guarantees,

ensuring that no assets remain locked in a script indefinitely and that no scripts run forever. Marlowe contracts can also be fully analyzed for contract faults before execution. Marlowe on-chain validators have been audited, and Marlowe supports contracts on both the Cardano mainnet and pre-production testing environment.

Decentralized application (DApp) developers, small and medium-sized enterprises, and other entities can use the Marlowe platform to develop Cardano-based DApps. The platform provides everything needed to build and run complete DApps on Cardano, leveraging the guarantees of Marlowe smart contracts and integrating with Web3 workflows through a TypeScript SDK. The platform includes:

- **The Marlowe Runtime** supports all the necessary on- and off-chain contract actions, including the tedious work of transaction construction and tracking contract activity by other participants.
- **The Marlowe TypeScript SDK** fully integrates Marlowe as a component within a complete DApp, including contract creation and initialization, as well as execution.
- **The Marlowe Playground** provides the environment for learning Marlowe through its Blockly representation, as well as simulation and analysis of contracts as they are developed.
- **End-to-end DApp examples** were developed in collaboration with Gimbalabs with Catalyst Fund12 support, and are designed to make access to the Marlowe Platform as straightforward as possible.
- **Other components** support free-standing execution of Marlowe smart contracts, including the Marlowe Runner and the Marlowe CLI.

The platform simplifies development for the developer community and provides key components, such as oracles, used in many DApps. Individual end users can also execute Marlowe contracts directly on the Cardano blockchain, using the facilities that the Marlowe Runner and the CLI provide. These features directly support decentralized finance (DeFi)

interactions, including peer-to-peer loans and escrow facilities.

The Marlowe platform offers a complementary route to DApp development alongside Aiken or Plinth as long as the DApp's core functionality can be expressed through Marlowe contracts. Each Marlowe contract has a predetermined, finite set of roles and behavior that is guaranteed to terminate. However, it is crucial to realize that a Marlowe DApp can dynamically create and execute multiple Marlowe contracts for particular use cases. For example, an escrow service DApp can generate a new Marlowe contract for each particular use of the service, and that contract will provide a guarantee of termination and no asset retention for each such use. Using the TypeScript SDK for Marlowe, developers can build complete DApps using familiar technology.

Marlowe originated as a research project funded by IO at the University of Kent in 2017 and became part of IO in 2019. As of autumn 2024, Marlowe is an independent open-source project owned and managed by Marlowe Language Community Interest Company (CIC), a non-profit organization (UK company number 16010921). This CIC supports Marlowe users and developers by overseeing development, maintenance, education, and training. It also hosts Marlowe open-source software repositories and maintains the ecosystem's online presence.

With Tomasz Rybczyk, I have been funded under Catalyst Fund13, [project 1300131](#), to formalize, develop, and implement the Marlowe Oracle protocol based on a novel pull-based model, where oracles respond dynamically to on-chain data requests, ensuring better adaptability and data coverage for a wider array of contracts. The project will also provide validator and runtime adaptations to support the protocol and integrate it with the Marlowe TypeScript SDK, Aiken, and the Marlowe DApp starter kit.

This is just the first step in Marlowe's new development phase, with plans to enhance the developer experience on the Marlowe platform and refine

the core language evolution to better support its aim of providing a high-level, chain-agnostic language for blockchain financial smart contracts. Please join us on this journey!

25.2. Smart contract programming languages

Cardano smart contract languages can be grouped into three categories:

- Native languages that run on the Cardano node
- Compiled languages that can be compiled to a native language
- Interpreted languages that are interpreted by a compiled language.

Native languages

The Cardano node can only process native languages. Currently, there are two available: simple scripts and Plutus.

The Shelley era introduced script addresses and [simple scripts](#), also known as native scripts. They can be used for multi-signature addresses, requiring multiple keys to sign a transaction to spend funds. The Allegra era extended simple scripts by adding conditional time-based functionality. This allows the creation of addresses with 'time locks', where funds can only be withdrawn before or after a specified time. A script can also be written to allow one group of keys to spend funds before a certain time, and another group after.

The Alonzo era introduced the Plutus scripting language, also referred to as Plutus Core. Plutus smart contracts validate transactions and because of this they are also called validation scripts or validators. In practice, the Cardano node executes an untyped version called Untyped Plutus Core (UPLC). The compilation pipeline from a compiled language to UPLC is covered in section [Plutus security](#). Plutus is a simple, functional language that enables general-purpose Turing-complete smart contracts on Cardano. It implements Cardano's extended UTXO (EUTXO) model, which is as powerful in expressing smart contract logic as Ethereum's account-

based model, allowing for arbitrary logic in smart contracts. A comparison of Plutus with Bitcoin Script and Solidity languages is presented in section [Plutus in comparison to Bitcoin Script and Solidity](#). The security advantages of Cardano's EUTXO model over Ethereum's account-based model are discussed in section [Cardano security](#).

Compiled languages

Smart contract developers do not write code directly in Plutus. Instead, they use compiled or interpreted languages that are compiled into Plutus. The language developed by IO that compiles to Plutus is called Plinth, previously known as PlutusTx. Plinth is a Turing-complete subset of the Haskell programming language ([Krijnen et al, 2023](#)). It draws from modern language research to provide a safe, full-stack programming environment based on Haskell, the leading purely functional programming language. It is a general-purpose smart contract language that focuses on security. The basics of coding smart contracts in Plinth are explained in section [Plutus smart contracts](#).

There are also other compiled smart contract languages developed by companies within the Cardano ecosystem. They are all domain-specific languages ([DSL](#)) as they target the smart contract domain. Some of them, including Plinth, are embedded DSLs ([eDSL](#)) because they are implemented as libraries in a general-purpose programming language. Some examples include:

- [Aiken](#): a unique, typed, purely functional DSL; IO supports its development
- [Plu-ts](#): a typed eDSL in TypeScript
- [OpShin](#): a typed eDSL in Python
- [Helios](#): a JavaScript/TypeScript SDK and typed DSL
- [Plutarch](#): a typed eDSL in Haskell for writing efficient Plutus validators
- [Scalus](#): a typed eDSL in Scala

- [Purus](#): a PureScript to Plutus Core compiler
- [Pluto](#): a DSL resembling UPLC with some syntactic sugar, written in a Haskell-like syntax.

One can read more about languages that compile to UPLC at the official [Plinth docs](#). The logic in all compiled Cardano smart contract languages follows the same rules defined by Plutus. Learning one compiled language also helps a developer to understand other compiled languages that use different syntax. You can refer to the [State of the Cardano Developer Ecosystem report – 2024](#) to see how much these languages are used in practice. As of 2024, the most commonly used ones are Aiken and Plinth. A comparison of these languages is discussed in section [Plinth in comparison to Aiken](#). A brief comparison of some of the listed languages can also be found in the following blogs:

- The [Emurgo\(Emurgo](#) blog about the programming languages behind Cardano on-chain code
- The [Essential Cardano](#) blog about programming languages.

Languages that compile to Plutus generate scripts that have the same logic, but might be optimized differently for factors like size or performance. This data is presented in the [UPLC benchmark](#) comparison. Community guidelines and tools can help optimize Plutus scripts for size, CPU, and memory consumption, reducing transaction fees. These tools can also help analyze the fees users will encounter while interacting with a Plutus script. The official documentation provides more information on the [Cardano fee structure](#). Links to Cardano developer tools are provided at the end of this section.

Some examples of projects that can be built using Plinth or other compiled Cardano smart contract languages include:

- NFT marketplaces and platforms ([NMKR](#), [CardaHub](#), [JPGStore](#), [JamOnBread](#))

- Decentralized exchanges (DEX) ([MuesliSwap](#), [MinSwap](#), [SundaeSwap](#), [GeniusYield](#))
- Automated lending and borrowing platforms ([Liqwid](#), [Lenfi](#), [FluidTokens](#))
- Digital identity management platforms ([Identus](#), [IAMX](#), [Profila](#), [Veridian](#))
- Decentralized, blockchain-powered mobile network ([WorldMobile](#))
- Decentralized artificial intelligence systems ([SingularityNET](#), [CardanoGPT](#))
- Decentralized autonomous organizations (DAO) ([IndigoDAO](#), [Charity DAO](#), [eLearning DAO](#))
- Decentralized synthetic assets protocol ([IndigoProtocol](#))
- Decentralized prediction markets ([Foreon](#))
- Decentralized cloud storage systems ([Iagon](#)).

You can also explore active projects built on Cardano on the pages below. These pages categorize projects into areas like DEX, identity and data, lending and borrowing, developer tools, education, artificial intelligence (AI), decentralized finance (DeFi), infrastructure, marketplaces, and more:

- [CardanoCube interactive map](#): presents projects in a visually engaging, interactive format. For each selected project, the page provides a description and links to an official webpage, white paper, social media pages, and GitHub repository.
- [CardanoSpot project library](#): offers a category filter to list projects by certain categories. For each selected project, a description is provided along with links to the official page and a white paper.
- [Cardano developers showcase](#) page: tags projects by groups and adds a short description to each.

Interpreted languages

The third category of smart contract languages in Cardano consists of interpreted languages that are interpreted by a compiled language. [Marlowe](#), initially developed by IO, is an interpreted smart contract language that is not Turing-complete. It is well-suited for designing financial contracts, such as those defined in Algorithmic Contract Types Unified Standards ([Actus](#)), for example. The Marlowe interpreter is written as a Plinth smart contract. Besides the programming language, the Marlowe project provides open-source tools to easily create, verify, and deploy secure financial smart contracts on Cardano. You can write smart contracts in JavaScript and Haskell or use Blockly, a visual coding solution. All language options are available in the online development environment – [Marlowe playground](#). The Marlowe language, its tools, architecture, and contract examples are presented in section [Marlowe smart contracts](#).

With Marlowe, it is possible to design a diversity of contracts for the following domains:

- Bonds, forwards, options, futures, swaps, etc
- Structured financial products
- Escrows
- Auctions
- Peer-to-peer loans
- Token swaps
- Airdrops.

Section [Marlowe security and best practices](#) explores the security and best practices of Marlowe in more detail.

On-chain and off-chain code

Sometimes, smart contract code is referred to as on-chain code because it runs in the node during the inclusion of new transactions that aim to spend a UTXO at a script address. Off-chain code, in contrast, runs on the

user's or a service provider's device and queries the blockchain, builds, signs, and submits transactions. A web application that connects with a wallet and interacts with one or more smart contracts is called a decentralized application (DApp). Chapter [Decentralized applications](#) covers DApps in more detail. Every DApp contains some off-chain code and interacts with one or more smart contracts that represent the on-chain code. Off-chain code tasks can be performed with a command line tool, such as the [Cardano CLI](#), or with the help of libraries and builder tools that are embedded in popular programming languages. Some of them include:

- [Blockfrost SDK](#): enables access to the Blockfrost API layer for Cardano. The SDK is provided in various programming languages such as Arduino, .NET, Crystal, Elixir, Go, Haskell, Java, JavaScript, Kotlin, PHP, Python, Ruby, Rust, Scala, and Swift.
- [MeshJS](#): a NodeJS-based open-source library providing numerous tools to easily build DApps on Cardano. It also integrates the popular [React](#) library.
- [Lucid](#): a popular JavaScript/TypeScript library for off-chain code, which is further developed by the [Evolution-SDK](#) project. The Evolution-SDK project was temporary called Lucid Evolution and was funded by [Catalyst Fund11](#).
- [Atlas](#): an all-in-one, Haskell-native application backend for writing off-chain code for Plutus smart contracts.

Explore Cardano tools that can be used for building DApps at:

- The [Builder Tools](#) page on the Cardano Developer portal. You can filter the tools by language/technology or by domain. Every tool contains a short description.
- The Cardano community-built [developer tools](#) list hosted on Essential Cardano.

You can also look at the State of the Cardano Developer Ecosystem report – 2024, listing the most commonly used [Cardano libraries](#).

Section [Plutus smart contracts](#) presents the MeshJS tool and showcases how to write off-chain code for smart contracts in subsections [Off-chain code with MeshJS](#) and [Minting policies and native tokens](#). We provide a link to a repository that contains the presented MeshJS code examples and also contains Lucid Evolution code examples.

25.3. Smart contract case studies

World Mobile Token smart contracts

The [World Mobile](#) company offers an interesting case study of a solution that can change the current state of internet networks. With the help of the Cardano blockchain and smart contracts, the company provides a sharing economy model to deliver network infrastructure and enable connectivity in a more distributed and decentralized manner.

The establishment of a sharing economy leads to reduced operational costs and more efficient resource allocation. Additionally, the token-based, decentralized nature of this sharing economy makes the model highly scalable in terms of deployment. Instead of depending on a centralized network operator to continuously assess the network's capacity and demand, which is always changing, the network's expansion is driven by the communities that require access to the internet. Smart contracts play a key role here: they remove intermediaries and incentivize network expansion through an automated reward system, whereby operators are rewarded for providing good-quality services.

Network overview

The World Mobile network consists of three layers of nodes, each with different responsibilities:

- **Earth nodes** contain the core business logic of the World Mobile Chain.

They provide an authentication layer (decentralized identity module), manage all blockchain transactions (blockchain module), and include a telecommunications layer (telecommunications layer).

- **Air nodes** are located in areas where connectivity is needed. They serve as the first point of contact with the network for individual users or entire communities.
- **Aether nodes** interface with legacy telecommunications networks. They handle protocol translations, media transcoding, and the routing of traffic to these networks.

Earth nodes core logic

To simplify the complexity of business logic within the network, Earth nodes are responsible for calling the appropriate smart contracts. For example, Earth nodes handle the processing of rewards for other nodes, ensuring automated payments are made once the conditions of the smart contracts are met. Additionally, Earth nodes process and verify identities provided by Air nodes, responding with the user's available account balance and a list of available services.

Earth node operators must stake a certain amount of tokens to join the network. The minimum number of tokens required to operate an Earth node is set at 100,000 tokens, as specified in the initial blockchain parameters. Each Earth node earns rewards based on several factors. Firstly, rewards are given for producing and committing blocks to the blockchain, which includes financial settlements and metadata such as the hash reference to call details records (CDRs). Secondly, node operators are rewarded for providing services to users, such as routing communication traffic (voice, SMS, etc).

Earth nodes can operate from any location globally; however, traffic routing within the network is biased towards nearer nodes to enhance performance and service quality.

World Mobile token and Earth node non-fungible tokens (NFTs)

Utilizing Cardano's native token capabilities, World Mobile introduced the [World Mobile Token](#) (WMT) as the primary currency for transactions and reward distribution within its ecosystem. The primary purpose of a WMT is to incentivize both token holders and node operators. Token holders support network operations by delegating their WMTs to node operators (stakers) who manage nodes to support the network. There is a finite supply of two billion WMTs, with only a portion initially circulating.

Additionally, there exists another currency within the World Mobile ecosystem – Earth node non-fungible token (ENNFT). ENNFTs are created using Cardano's native token functionality and are issued to Earth node owners who locked 100,000 WMT to a smart contract before January 4, 2023. These NFTs provide monthly rewards; each month, Earth node owners receive 1,300 WMT (1.3%) for maintaining their node.

Cardano within the WMT sharing economy model

In contrast to traditional network models, the operation and maintenance of nodes within the WMT sharing economy model is shared with communities and local businesses. This approach enhances scalability and reduces costs by allocating resources to areas where they are most needed. Leveraging blockchain technology and smart contracts offers numerous advantages that align seamlessly with this distributed model:

- **Transparency:** Cardano records information that can be easily accessible to different stakeholders to make more informed decisions
- **Privacy:** user information is stored using private/public encryption provided by Cardano
- **Immutability:** Cardano's EUTXO model ensures transaction immutability and restricts spending to only unspent transaction outputs, making the reward system more deterministic.

25.4. Cardano addresses

A blockchain address serves as a communication link between the blockchain and the user. With the introduction of stake pools in the Shelley era, a Cardano address consists of two parts: the *payment* part and the *staking* part.

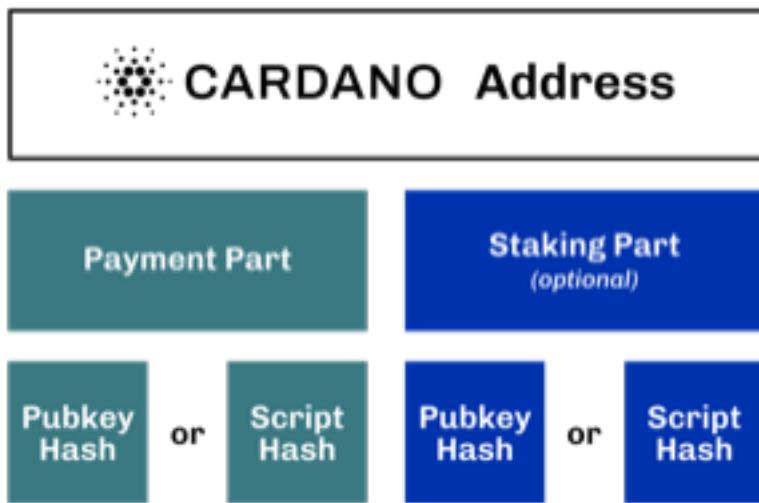


Figure 37. Cardano address

Both parts of a Cardano address are cryptographically derived from the private key, containing the same owner information. The payment part, which is always present, determines the conditions under which a UTXO at the address can be spent. It is either defined by the hash of a public key or a Plutus script. If it contains a public key hash, UTXOs can only be spent if the transaction is signed with the corresponding private (signing) key. If it contains a script hash, the script executes during validation to decide whether UTXOs at the address can be spent.

The optional staking part controls delegation and staking rewards. If defined with a public key hash, the corresponding private key owner can spend the rewards. If defined with a script hash, the script determines the conditions under which staking rewards can be spent.

Cardano Shelley addresses can be divided into four categories:

- Base addresses

- Pointer addresses
- Enterprise addresses
- Reward account addresses.

Only *base* and *pointer* addresses carry staking rights. The *base* address directly specifies the staking key controlling the stake, while a *pointer* address indirectly specifies it. The advantage of the *pointer* address is that it can be considerably shorter than the hash used in base addresses. *Enterprise* addresses, which carry no staking rights, are also shorter and can be used for sending and receiving funds. *Reward account addresses*, used to distribute proof-of-stake rewards (either directly or via delegation), are cryptographic hashes of the public staking key. They follow the account-based model, unlike the UTXO model. Rewards are reflected in accounts, and UTXOs are created only when rewards are withdrawn.

The Shelley era continued to support Byron-era *bootstrap addresses* and *script addresses*. The [Cardano addresses](#) documentation page provides more information about address categories.

25.4.1. Binary format

Under the hood, a Cardano address is a sequence of bytes that conforms to a particular format. Users will typically interact with addresses only after they have been encoded into sequences of human-readable characters. [Bech32](#) and [Base58](#) are encodings used in Cardano, as opposed to standard hexadecimal notation ([Base16](#)). These encodings represent the addresses users perceive, though they are distinct from the underlying byte sequences. Shelley addresses, which include staking addresses, use Bech32 encoding without a character length limit. In contrast, Byron addresses are encoded in Base58, allowing for easy differentiation from Shelley-era addresses. Below are examples of the different address types:

Figure 38. Address types, source: CIP-19

In Cardano addresses, the sequence of bytes (decoded from *Bech32* or *Base58*) consists of two parts – a one-byte header and a payload of several bytes. Depending on the header, the interpretation and length of the payload vary. In the header byte, the bits from 7 to 4 indicate the type of addresses being used; we'll call these four bits the header type. The remaining four bits from 3 to 0 are either unused or refer to what we call the network tag. You can see a graphical representation below:

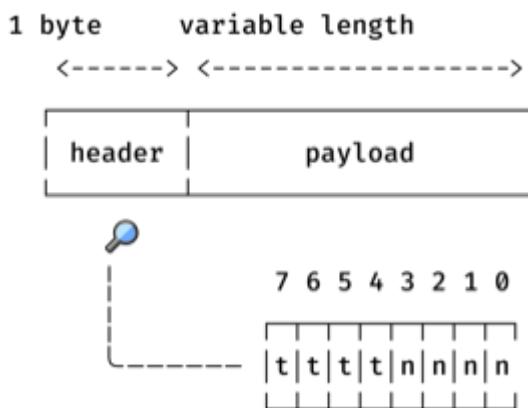


Figure 39. Address structure, source: CIP-19

Depending on the various header types and address formats, there are currently 11 types of addresses in Cardano, which are either Shelley or Byron addresses, including the addresses used for staking. You can see the various address structures below:

	TYPE	TAG	PAYMENT	DELEGATION	
ADDRESS = %b0000		NETWORK-TAG	KEY-HASH	KEY-HASH	; type 00,
Base address					Shelley
	\ %b0001	NETWORK-TAG	SCRIPT-HASH	KEY-HASH	; type 01,
Base address					Shelley

```

        \ %b0010 | NETWORK-TAG | KEY-HASH      | SCRIPT-HASH      ; type 02,
Base                                         Shelley

address
        \ %b0011 | NETWORK-TAG | SCRIPT-HASH | SCRIPT-HASH      ; type 03,
Base                                         Shelley

address
        \ %b0100 | NETWORK-TAG | KEY-HASH      | POINTER       ; type 04,
Pointer                                         Shelley

address
        \ %b0101 | NETWORK-TAG | SCRIPT-HASH | POINTER       ; type 05,
Pointer                                         Shelley

address
        \ %b0110 | NETWORK-TAG | KEY-HASH      ; type 06,
Enterprise                                         Shelley

address
        \ %b0111 | NETWORK-TAG | SCRIPT-HASH      ; type 07,
Enterprise                                         Shelley

address
        \ %b1000 | BYRON-PAYOUT      ; type 08,
Byron /                                         Shelley

Bootstrap address
        \ %b1110 | NETWORK-TAG | KEY-HASH      ; type 14,
Stake                                         Shelley

address
        \ %b1111 | NETWORK-TAG | SCRIPT-HASH      ; type 15,
Stake                                         Shelley

address

NETWORK-TAG = %b0000 ; Testnet
                    \ %b0001 ; Mainnet

```

For *Bech32*-encoded addresses (used for all Shelly addresses), the last six characters of the data part form a checksum of the previous address data

and contain no information. This allows for quick offline validity checks and provides an additional safety measure for wallets. For an additional explanation of address type structures, refer to [Cardano Improvement Proposal 19](#) (CIP-19), which defines the technical details of Cardano addresses.

25.5. Marlowe smart contracts

25.5.1. About Marlowe

Marlowe is a domain-specific language (DSL) for building financial smart contracts. One can think of Marlowe as a robust, open-source technology that provides a special-purpose language describing asset flows on the blockchain. As a special-purpose DSL, it offers a higher-level model of contracts in a more restricted domain than other Cardano languages compiling directly to Plutus. This means that Marlowe can provide safety guarantees, such as ensuring that no assets are held in a script indefinitely, by the design of the language. Additionally, it offers tools for a full analysis of potential contract faults before contract execution.

The implementation of Marlowe on Cardano is carried out using Plinth. Marlowe smart contracts are interpreted by a Plinth smart contract under the hood. Marlowe enables the implementation of specific domain expertise to write and manage contracts conveniently, without the steep learning curve associated with software development, blockchain, or smart contracts. Marlowe's core technology has been audited, and it supports contracts on mainnet and pre-production testing environments. Its Runtime enables all the necessary on- and off-chain contract activity, including the tedious work of transaction construction. The TypeScript SDK supports Marlowe as a component within a complete DApp. This makes it a smart contract technology that is complementary to Aiken, Plinth, or any other Cardano smart contract language. It abstracts away the complexities of Cardano and provides a local, account-based model like Ethereum.

Beyond the notable benefit of usability by non-programmers, the Marlowe language offers many other advantages:

- Easily checks that programs have the intended properties
- Ensures high assurance that the contract consistently fulfills its payment obligations
- Helps people write programs in the language using special-purpose tools
- Emulates contract behavior before execution on the blockchain, ensuring intended performance through static analysis
- Provides valuable diagnostics to potential participants before they commit to a contract
- Formally proves properties of Marlowe contracts, delivering the highest level of assurance regarding intended behavior through logic tools
- Prevents certain flawed programs from being written by the design of the language
- Mitigates some unanticipated exploits that have affected existing blockchains.

Marlowe is modeled after special-purpose financial contract languages popularized over the past decade by academics and enterprises, such as LexiFi, which provides contract software for the financial sector. In developing Marlowe, these languages have been adapted to work on blockchain. Although it is implemented on the Cardano blockchain, Marlowe could also be implemented on Ethereum or other blockchain platforms, making it 'platform-agnostic', similar to modern programming languages like Java and C++. Designed as an industry-scale solution, Marlowe embodies examples from the [Actus](#) taxonomy for financial contracts. It can also interact with real-world data providers through oracles, enabling contract participants to make choices within the contract flow that determine on-chain and off-chain outcomes, such as in a wallet.

Marlowe is based on original, peer-reviewed research conducted by the Marlowe team, initially at the University of Kent with support from a research grant from IO, and later as an internal engineering team within IO. The Marlowe team at IO was also collaborating with the [Wyoming Center for Blockchain and Digital Innovation](#) (CBDI) at the University of Wyoming. More information about the research conducted for Marlowe can be found on the official documentation page, which also lists [published research papers](#) related to Marlowe.

In the future, Marlowe will be administered by an independent vehicle, a not-for-profit organization, which will ensure community representation and stewardship. This will allow the community to actively contribute to its roadmap and propose updates and enhancements. To further support Marlowe's vision, a new [Special Interest Group](#) (SIG) has been established that is active on Discord, focusing on Marlowe's continued innovation and enhancement, with builders at the heart.

In summer 2024, IO transferred the intellectual property rights for Marlowe to the Marlowe Foundation – a non-profit organization established to oversee the continued development of Marlowe and its ecosystem as a community-based project. The Marlowe repositories will be transferred to the [marlowe-lang](#) GitHub, and continued development will take place there.

25.5.2. Developer tools and services

Marlowe provides a set of open-source tools that help create, test, and deploy secure smart contracts on Cardano. It offers intuitive solutions to create, utilize, and monetize smart contracts with ease, catering to developers of all expertise levels. The following developer tools and features are available:

- **Marlowe language** – a DSL that includes a web-based platform to build and run smart contracts

- **Marlowe Playground** – a simulator that allows testing Marlowe smart contracts before deployment to ensure intended code behavior
- **Marlowe Runner** – an easy-to-use DApp that can be used to deploy, execute, and interact with Marlowe smart contracts
- **Marlowe Scan** – a tool for visualizing Marlowe contracts on the Cardano blockchain
- **Marlowe Runtime** – the application backend for managing Marlowe contracts on Cardano, which includes easy-to-use, higher-level APIs for developers to build and deploy enterprise and Web3 DApp solutions
- **Marlowe CLI** – provides capabilities to work with Marlowe's Plutus validators and run Marlowe contracts manually
- **Marlowe starter kit** – provides tutorials for developers to learn and run simple Marlowe contracts on Cardano
- **Marlowe TypeScript SDK** (currently in beta) – a suite of TypeScript/JavaScript libraries for developing web DApps on Cardano using Marlowe technologies
- **Demeter.Run integration** – a web service that allows building Marlowe projects without installing any software
- **Documentation website** – significantly expanded, updated, and integrated into the updated Marlowe website.

IMPORTANT: In the transition phase of Marlowe, where some GitHub repositories of the above-mentioned tools may not be actively maintained by the Marlowe foundation, some of these tools might have issues when using them with the latest test or main network due to Cardano updates.

The [Marlowe language](#) enables users to build contracts by combining a small number of constructs, which can describe many different financial contracts. Contract participants can engage in various actions: they can be asked to deposit money or make choices between various alternatives [source: Marlowe: implementing and analyzing financial contracts on

blockchain, Lamela et al. 2020]. Marlowe contract examples are presented in section [Contract examples](#).

The [Marlowe Playground](#) is the main entry point for learning and developing Marlowe smart contracts. It is an online simulation that allows users to experiment with, develop, simulate, and analyze Marlowe contracts in a web browser without installing any software. Supported programming languages include Marlowe itself, Haskell, JavaScript, and TypeScript. The playground also includes Blockly – an editor for visual programming. Together, these languages form a plug-and-play building and simulation smart contract environment that is simple to use, visual, and modular. The playground also allows downloading contracts as a JSON file for further use. For more details on how to use the playground, see this [video tutorial](#).

[Marlowe Runner](#) is an online tool that facilitates the deployment and execution of Marlowe contracts on the blockchain, eliminating the need for command-line expertise. With Marlowe Runner, users can deploy contracts created in the Marlowe playground, test them, and interact with them in a simulated environment before mainnet deployment. For this, users need to connect to the Runner using a Cardano wallet such as [Lace](#) or [Eternl](#). Contracts can be uploaded to the Runner as a JSON file, or one can manually paste the JSON structure into an editor window. One can look at the source graph before creating a contract, which is also available when interacting with the contract. If a Marlowe contract uses role tokens, the funds cannot be retrieved from the role-token contract with the Runner. In such a case, one can use the [Payout DApp prototype](#). A [video tutorial](#) about using Marlowe Runner can be found on the IO YouTube channel.

[Marlowe Scan](#) is a tool that allows users to query information about Marlowe contracts and view the current contract state. It can be used for the preview and pre-production test networks and the mainnet. Users can also view the contract code, download it, and view a list of transaction IDs.

[Marlowe Runtime](#) is the application backend for managing Marlowe contracts on Cardano. It provides easy-to-use, higher-level APIs and complete backend services that enable developers to build and deploy enterprise and Web3 DApp solutions using Marlowe. Users don't need to assemble the "plumbing" that manually orchestrates a backend workflow for a Marlowe-based application. Runtime takes commands relevant to the Marlowe ledger and maps them to the Cardano ledger. It consists of a series of services that can be divided into frontend and backend components. Marlowe Runtime backend services are off-chain components largely responsible for interfacing with a Cardano node. They offer abstractions to hide many implementation details of Plutus and the Cardano node directly. There are two ways to interface with Marlowe Runtime:

- Using Marlowe Runtime web REST API
- Using `marlowe-runtime-cli` command line tool.

The role of Runtime is to facilitate the mapping between the Marlowe conceptual model and the Cardano ledger model in both directions. Users can primarily perform two types of tasks: discovering and querying on-chain Marlowe contracts, as well as creating Marlowe transactions. More specifically, the tasks include the following:

- Creating contracts
- Building transactions
- Submitting transactions
- Querying contract information and history
- Listing contracts
- Subscribing to live contract updates.

There are two main use cases for using Marlowe as a layer for smart contract developers. Depending on the complexity of the smart contract and the DApp, higher-level operations provide a simplified interface,

allowing developers to focus mainly on smart contract logic rather than implementation details. However, more complex workflows might require lower-level control, necessitating a deeper understanding of Plutus. For more information, refer to the list of high- and low-level operations on the [developer tools](#) documentation page.

[Marlowe CLI](#) is a command-line tool that provides access to Marlowe capabilities on testnet and mainnet. It is specifically built to run Marlowe contracts directly without needing a web browser or a mobile app. Just as the [cardano-cli](#) tool enables plain transactions, simple scripts, and Plutus scripts, the Marlowe CLI tool provides the ability to interact with and develop Marlowe contracts. Users can measure transaction size, submit transactions, test wallet integration, and debug validators. The tool provides a concrete representation of Marlowe contracts that are quite close to what is occurring on-chain. Users can also create their own workflows that operate with Marlowe or develop custom tool sets. This allows them to wrap the Marlowe CLI tool similarly to how developers have wrapped the cardano-cli to create services such as libraries, faucets, and marketplaces.

The image below offers an overview and description of the Marlowe CLI and Marlowe Runtime tools for running and querying Marlowe contracts:

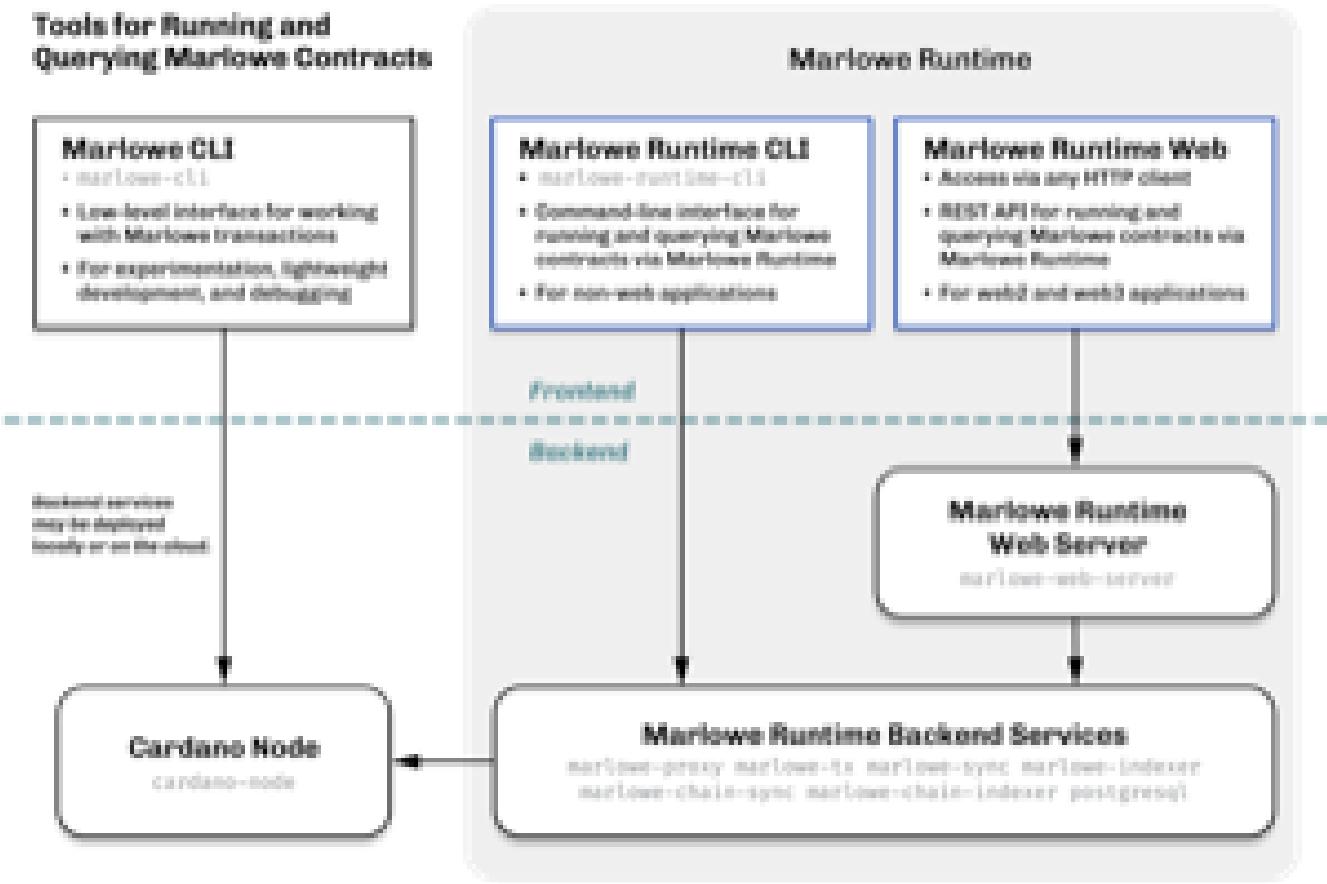


Figure 40. Marlowe tools, source: [Marlowe docs](#)

The [Marlowe starter kit](#) is a GitHub repository that contains Jupyter notebook lessons demonstrating the use of the Marlowe CLI and Marlowe Runtime tools through concrete examples. It can be used with Demeter Run, a Docker deployment of Marlowe Runtime, and Nix to run the Jupyter notebooks. The repository provides instructions for setting up various environments. Additionally, the Jupyter notebooks offer guidance on interacting with Marlowe smart contracts, covering different approaches and tools.

The [Marlowe TypeScript SDK](#) (TS-SDK) consists of JavaScript and TypeScript libraries, available as npm packages, designed to support DApp developers with the necessary tools to build and integrate with the Marlowe smart contract ecosystem. There are [short video tutorials](#) on the IO YouTube channel that demonstrate how to use the TS-SDK to build an example DApp. Since the tutorials were created during the beta stage, function names may change in the official release. The TS-SDK offers the

following features:

- Smart contract toolkit
- Integration with Marlowe Playground
- Wallet connectivity
- Integration with Runtime
- Coordination between wallets and Runtime
- Prototype DApp examples.

You can read more about these features in the official TS-SDK documentation. To interact with Marlowe contracts, the TS-SDK needs a Runtime instance. TS-SDK GitHub repository provides the [following table](#) showing the compatibility between the SDK and the Runtime versions. The SDK also provides a wrapper around the [Lucid Library](#). This allows using the SDK in a Node.js environment.

The [Demeter.Run](#) platform, developed and maintained by [TxPipe](#), offers a variety of tools and development environments targeting the Cardano ecosystem. Their price model depends on the usage of their services, and users also have the option to get some working time for free. You can read more about the platform in section [Setting up a Plinth development environment](#).

The [Marlowe documentation](#) provides extensive explanations, links to learning resources, and access to tutorials and community resources from the top bar.

25.5.3. Marlowe Runtime architecture

Below is the Marlowe Runtime architecture:

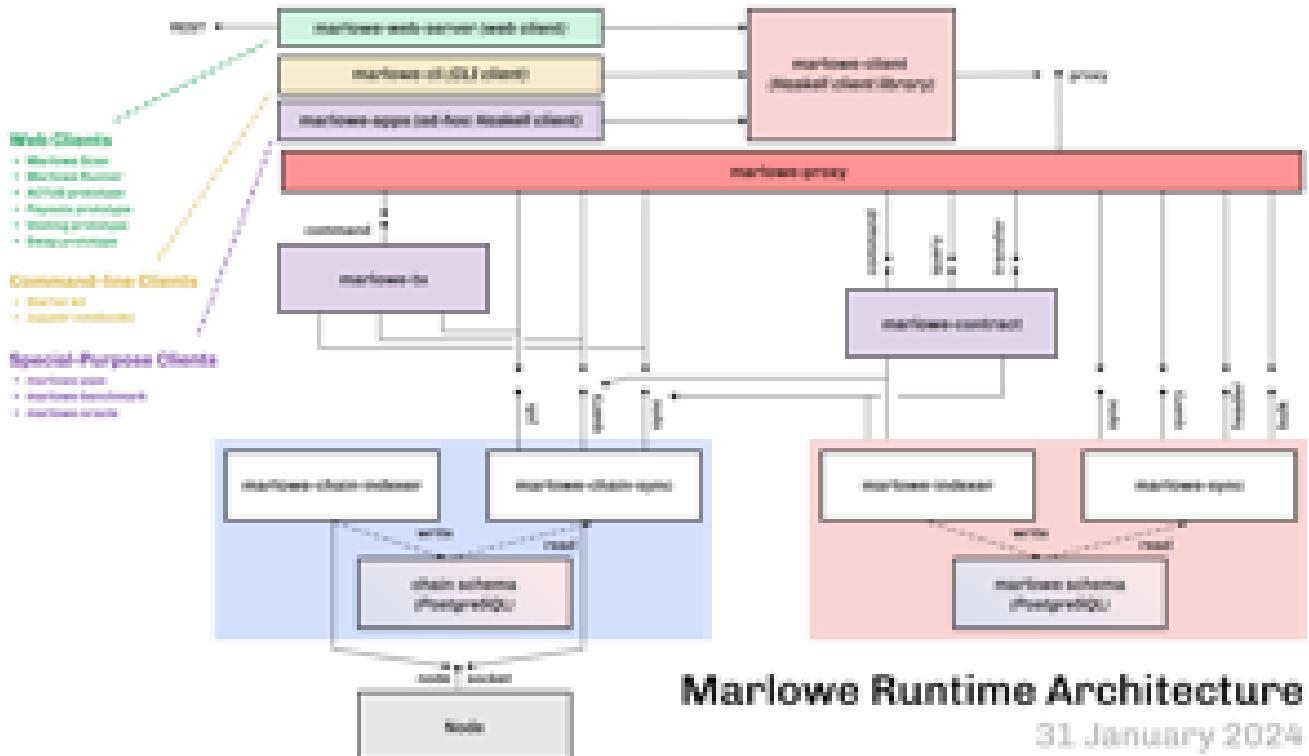


Figure 41. Marlowe runtime architecture, source: [Marlowe GitHub](#)

The Marlowe Runtime backend consists of a chain-indexing and query service (`marlowe-chain-indexer` / `marlowe-chain-sync`), a contract-indexing and query service for Marlowe contracts (`marlowe-indexer` / `marlowe-sync`), and a transaction-creation service for Marlowe contracts (`marlowe-tx`). These backend services work together, relying upon `cardano-node` for blockchain connectivity and PostgreSQL for persistent storage. Access to the backend services is provided via a command-line client (`marlowe-runtime-cli`), or a REST/WebSocket server (`web-server`) that uses JSON payloads. Web applications can integrate with a [CIP-30 light wallet](#) for transaction signing, whereas enterprise applications can integrate with `cardano-wallet`, `cardano-cli`, or `cardano-hw-cli` for the same purpose.

The backend services use typed protocols over TCP sockets, with separate ports for control, query, and synchronization. Each service handles rollbacks using intersection points that reference specific slots/blocks on the blockchain. Most of the data flow is stream-oriented, and the services prioritize statelessness. The information flow within the backend

maximizes the node as the single source of truth, minimizing the danger of downstream components receiving inconsistent information. The Haskell types in the client API for Runtime Clients are independent of various Cardano packages for ledger, node, and Plutus, resulting in a Haskell client for Runtime having minimal dependencies in its `.cabal` file.

Please note that the Marlowe Runtime architecture may evolve. Refer to the [Marlowe documentation](#) for the latest version.

25.5.4. Contract examples

Marlowe is designed to create the following building blocks of financial contracts:

- Payments to and deposits from participants
- Choices by participants
- Real-world information.

It is a small language with a handful of different constructs that, for each contract, describe behavior involving a fixed, finite set of roles or accounts. When a contract is run, the roles it involves are fulfilled by participants, who are identities on the blockchain. An on-chain token represents each role. Roles can be transferred during contract execution, meaning they can be traded. Users can also use external accounts represented by their Cardano addresses instead of roles. In this case, role tokens are not created. In a Marlowe contract, internal accounts correspond to roles or external accounts, with internal accounts controlled by the smart contract. Typically, all participants should make a deposit, sending funds from their wallets to the Marlowe contract, which retains the funds and associates them with the relevant internal accounts. Depending on the contract's terms, funds can be transferred between these internal accounts and back to external accounts. No funds are permanently locked in a Marlowe contract; when the contract concludes, any remaining funds in internal accounts can be withdrawn by their

owners.

Contracts are built by putting together a small number of constructs that, in combination, describe and model many different kinds of financial contracts. Some examples include:

- A running contract that can make a payment to a role or a public key
- A contract that can wait for an action by one of the roles, such as a deposit of currency
- A choice from a set of options.

Crucially, a contract cannot wait indefinitely for an action: if no action has been initiated by a set time (the timeout), the contract will continue with an alternative behavior. For example, it may refund any funds in the contract as a remedial action. Marlowe contracts can branch based on alternatives and have a finite lifetime, after which any remaining funds are returned to the participants. This feature means that funds cannot be locked forever in a contract. Depending on the contract's current state, it can choose between two alternative future courses of action, which are also contracts. When no further actions are required, the contract closes, and any remaining funds are refunded. Marlowe is embedded in Haskell and is modeled as a collection of algebraic data types, with contracts defined by the *Contract* type:

```
data Contract = Close
  | Pay Party Payee Token Value Contract
  | If Observation Contract Contract
  | When [Case] Timeout Contract
  | Let ValueId Value Contract
  | Assert Observation Contract
```

Marlowe has six ways of building contracts. Five of these methods – **Pay**, **Let**, **If**, **When**, and **Assert** – build a complex contract from simpler contracts, and the sixth method, **Close**, is a simple contract. At each step of execution, besides returning a new state and continuation contract, it is possible that

effects – payments – and warnings can also be generated. A description of each of the methods that the Contract data type defines can be found in the [Marlowe language guide](#) hosted on the official documentation page.

The Haskell source code for the data types that Marlowe code uses can be found in the [marlowe-cardano](#) GitHub repository. If you are writing Marlowe version 1 scripts, the module you need to import in a Haskell project to be able to write Marlowe code is [Language.Marlowe.Extended.V1](#) ([source code](#)). Some important Haskell data types that this module exports are contained in the [Language.Marlowe.Core.V1.Semantics.Types](#) module ([source code](#)). You can look up these modules if you view the documentation for Marlowe dependencies. The [marlowe-dependency-docs](#) GitHub repository contains instructions for setting up your own documentation server.

Let us now look at an example of a Marlowe contract involving three parties – Alice, Bob, and Charlie. In this contract, Alice and Bob deposit 10 lovelaces. Then, Charlie decides whether Alice or Bob receives the total amount. If any of the three parties fails to participate, the contract ensures that all deposited funds are reimbursed. You can see a demonstration of this contract’s design using Blockly in the Marlowe Playground in the following [Plutus Pioneer program](#) video.

Below, you can see the contract code in the Marlowe language:

```
When
  [Case
    (Deposit
      (Role "Alice")
      (Role "Alice")
      (Token " " " ")
      (Constant 10)
    )
    (When
      [Case
        (Deposit
          (Role "Bob"))
```

```

        (Role "Bob")
        (Token "" "")
        (Constant 10)
    )
    (When
        [Case
            (Choice
                (ChoiceId
                    "Winner"
                    (Role "Charlie")
                )
                [Bound 1 2]
            )
            (If
                (ValueEQ
                    (ChoiceValue
                        (ChoiceId
                            "Winner"
                            (Role "Charlie")
                        ))
                    (Constant 1)
                )
                (Pay
                    (Role "Bob")
                    (Account (Role "Alice"))
                    (Token "" "")
                    (Constant 10)
                    Close
                )
                (Pay
                    (Role "Alice")
                    (Account (Role "Bob"))
                    (Token "" "")
                    (Constant 10)
                    Close
                )
            )
        )
    )
    1682551111000 Close
)
)
1682552111000 Close
)
1682553111000 Close
)
```

Next is a flowchart generated with the Marlowe Runner that indicates possible actions and outcomes of the above contract. The highlighted block is the start of the contract, and the greyed-out blocks show possible execution paths:

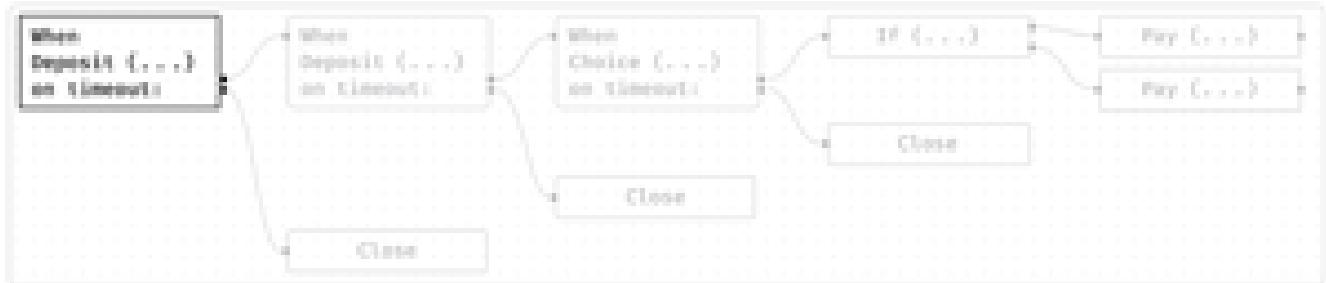


Figure 42. Marlowe contract tree

Below, you can see the image of the contract implemented in the Blockly tool:

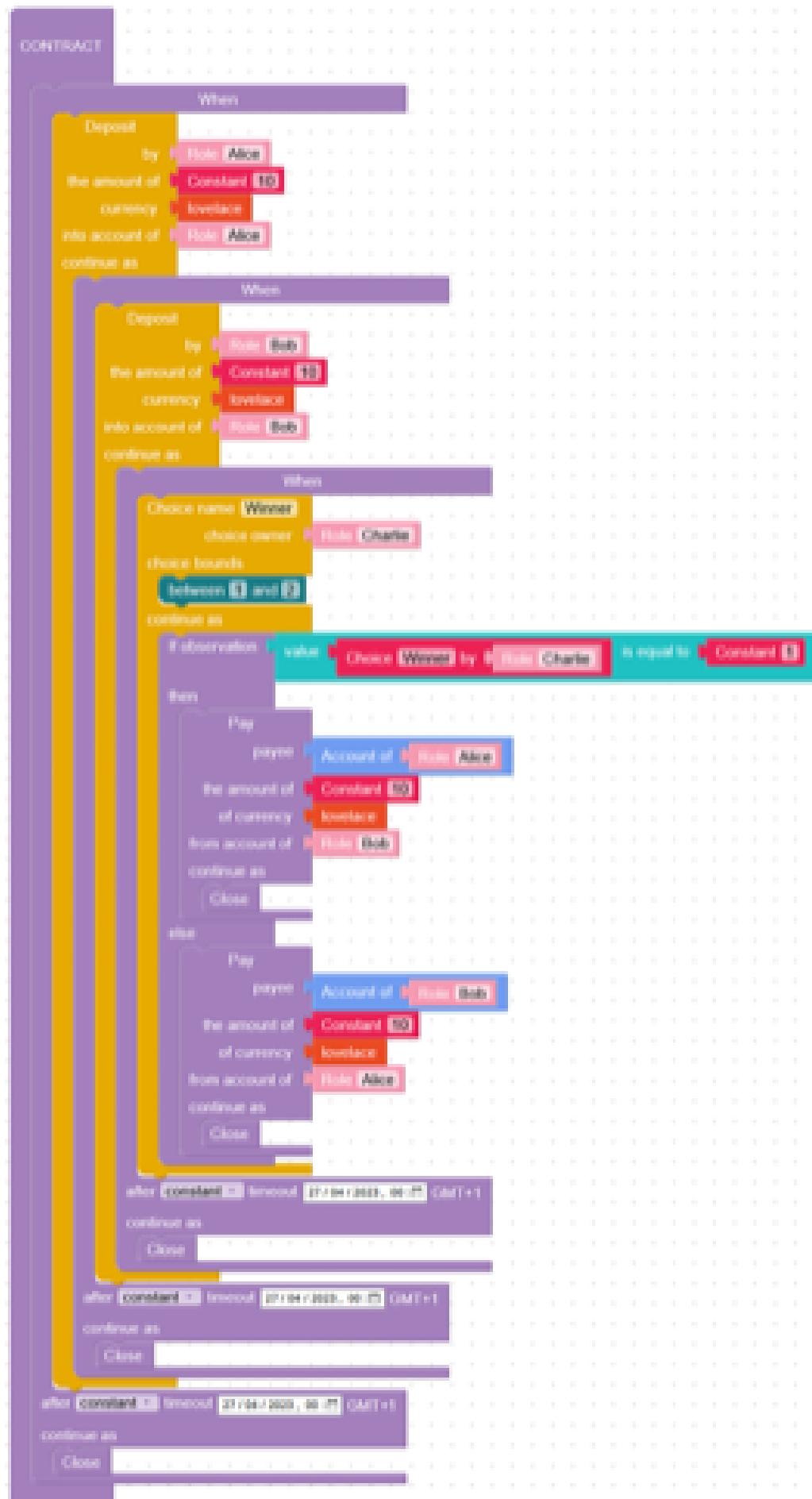


Figure 43. Marlowe choice contract

The **When** constructor, in the beginning, waits for an external action that we specify in square brackets. If this external action does not happen, we will close the contract. We do this by specifying a timeout in POSIX time (measured as milliseconds after UNIX epoch – 00:00:00 UTC on January, 1, 1970) after the closing square bracket of the **When** constructor and add the **Close** constructor at the end. If the external action does not happen before this timeout, the contract terminates. Our external action will be a deposit by Alice, and we say that the deposit should happen until a certain time. We use the **Case** constructor for that, which contains two pairs of regular brackets. The first one defines the case of what has to happen for the contract to progress, and the second one defines the logic that follows if the case action is met. In the first bracket, we specify that Alice should deposit 10 lovelaces. We do this by using the **Deposit** constructor. First, we specify to which account the deposit should go. We can specify this with the **Role** constructor followed by the role name, which, in our case, is Alice. Second, we specify who is depositing into the previously mentioned account. We can again use the **Role** constructor and Alice's name. We could also use the **Address** constructor, where we would have to input Alice's Cardano address. Then, we have to specify what kind of currency we are depositing. For this, we use the **Token** constructor and add the currency symbol and the token name. For the ada currency, we can use two empty strings. At the end, we specify the amount (in lovelaces) of the currency we want to deposit. For this, we can use the **Constant** constructor followed by a number.

After defining the first part of the **Case** statement, the next step is to outline the logic for when the condition is met. This begins with a new **When** statement, in which Bob will deposit 10 lovelaces, and a new timeout is set. The code for Bob's deposit mirrors Alice's, but the role name changes from Alice to Bob. After that, we again define the logic that follows if Bob makes his deposit. We use a **When** statement that says Charlie has to make a choice. We set a timeout for the statement in case Charlie does not make his choice and then the contract gets terminated. We use a

`Case` statement for Charlie to make a choice, and in the code, we apply the `Choice` constructor for that decision. This constructor takes two arguments. First, it takes the information about the choice name and the person making the choice. The `ChoiceId` constructor defines this, specifying the person with the `Role` statement. The second argument is a list of integers that defines Charlie's possible choices. Since Charlie decides whether the funds go to Alice or Bob, the list contains only two numbers.

The logic that follows after that is a conditional statement, which uses the `If` constructor. First, it takes the condition statement and then the two possible cases depending on the condition. For the condition statement, we use the `ValueEQ` constructor. It takes the value that Charlie chose and compares it to a number. We retrieve the value from Charlie's choice by using the `ChoiceValue` constructor and input the `ChoiceId` statement that we previously used. The value to which we compare it is 1, for which we again use the `Constant` constructor. The first case that follows the conditional statement is when the choice equals 1, which means that the funds from Bob will go to Alice. To transfer the funds from Bob's internal account to Alice's internal account, we use the `Pay` constructor. We first specify the party that sends the money, for which we can use the `Role` statement. After that, we have to specify the account with the `Account` constructor, which can again take a `Role` statement. Because we chose the `Account` constructor, the transfer occurs between internal accounts. There is also a `Party` constructor that can be used in this field, which takes an `Address` constructor and sends the money to an external address. Next, we need to specify the currency and the amount we want to send. At the end, we write the `Close` constructor, which means that the funds from the internal accounts will be sent back to the external account of those parties. The second `Pay` statement is the same, but the roles of Alice and Bob are reversed, as Charlie will send the funds to Bob if he makes choice 2.

Let us look at another, more complex example that uses the same Marlowe constructors. The contract will be an escrow that regulates a funds

transfer between a buyer and a seller. If there is a disagreement between them, a mediator will decide whether the funds are refunded or paid to the seller. Below, you can see the Marlowe contract code:

```

When [
  (Case
    (Deposit (Role "Seller") (Role "Buyer")
      (Token "" ""))
      (ConstantParam "Price")))
  (When [
    (Case
      (Choice
        (ChoiceId "Everything is alright" (Role "Buyer")) [
          (Bound 0 0)]) Close)
    ,
    (Case
      (Choice
        (ChoiceId "Report problem" (Role "Buyer")) [
          (Bound 1 1)])
      (Pay (Role "Seller")
        (Account (Role "Buyer")))
        (Token "" "")
        (ConstantParam "Price"))
      (When [
        (Case
          (Choice
            (ChoiceId "Confirm problem" (Role "Seller"))
[           (Bound 1 1)]) Close)
        ,
        (Case
          (Choice
            (ChoiceId "Dispute problem" (Role "Seller"))
[             (Bound 0 0)])
        (When [
          (Case
            (Choice
              (ChoiceId "Dismiss claim" (Role
"Mediator")) [
                (Bound 0 0)])
              (Pay (Role "Buyer"))
            )
          )
        )
      )
    )
  )
]

```

```

        (Party (Role "Seller"))
        (Token "" ""))
        (ConstantParam "Price") Close))
        ,
        (Case
        (Choice
            (ChoiceId "Confirm problem" (Role
"Mediator")))
                [(Bound 1 1)]) Close])
        (TimeParam "Mediation deadline")
        Close)
    )]
    (TimeParam "Complaint response deadline")
    Close))
)
]
(TimeParam "Complaint deadline")
Close)
)
]
(TimeParam "Payment deadline")
Close

```

The next image shows the contract implemented using the Blockly tool:

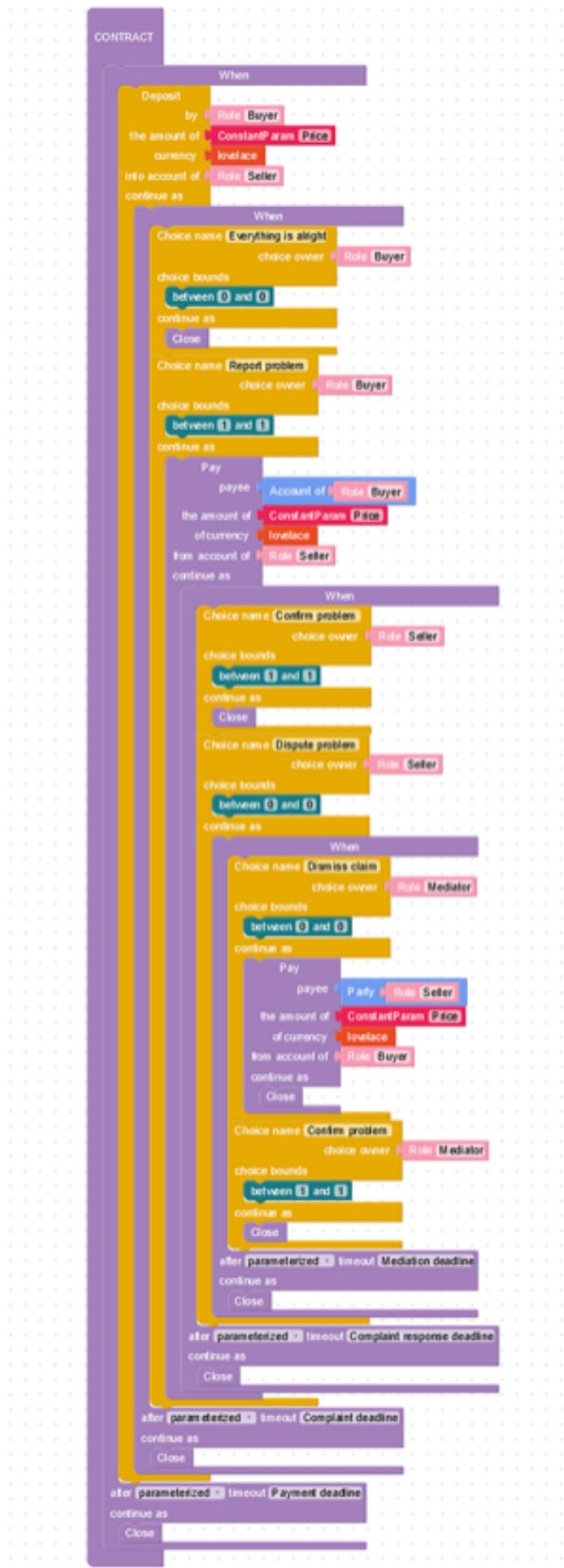


Figure 44. Marlowe escrow contract

First, the buyer must deposit the funds into the seller's internal account, with the amount defined by the *Price* parameter. The next **When** statement presents two options. In the first case, the buyer confirms everything is fine, and the contract closes. This is handled using a **Choice** statement, where the buyer has only one option. In the second case, the buyer reports a problem, and the funds are returned to the buyer's internal account. Two further possibilities arise: if the seller agrees with the problem, the contract closes, and the buyer's funds are transferred back to their external account. If the seller disputes the issue, and no action is taken by the timeout, the contract closes, and the buyer is refunded. If disputed, another **When** statement allows the mediator to either dismiss the claim, transferring the funds to the seller's internal account before closing the contract, or confirm the issue, in which case the buyer is refunded, and the contract closes. If the mediator does not act by their deadline, the contract closes, and the buyer is refunded. All deadlines are set as time parameters before the contract is submitted on-chain.

Another feature Marlowe contracts enable is peer-to-peer trustless lending. A borrower can obtain loans directly from individual lenders, bypassing traditional financial institutions. An example of a zero-coupon bond contract, which functions as a peer-to-peer lending agreement, can be found on the [Marlowe starter kit GitHub page](#). In this case, the loan is not collateralized, meaning the lender risks losing the funds if the borrower fails to repay, as the smart contract can not enforce repayment. There are a couple of options to mitigate this risk:

- Back the smart contract with a traditional legal contract
- In some B2B environments, bilateral or multilateral umbrella legal agreements cover instruments like this
- Combine with a reputation system, as is commonly done in micro-lending
- Add a guarantor to the contract

- Link the contract to a margin account
- Bundle the contract with other instruments to create a structured product that mitigates the default risk.

Regardless of the Marlowe contract a user interacts with, if role tokens are involved, they can be traded, effectively transferring the token ownership to another person. A Marlowe contract can facilitate these token trades, and ownership of role tokens can also be traded for certain time periods. More contract examples can be found in the Marlowe Playground or the [Real world Marlowe](#) GitHub repository, which also showcases the off-chain code for interacting with contracts. Additionally, a [Marlowe Actus implementation](#) is available for [Actus](#).

Marlowe DApps can be discovered by searching the keyword "Marlowe" in various [Project Catalyst](#) proposals. The Marlowe team at IO has also developed the following Marlowe DApps:

- [Marlowe Payouts](#) – helps users discover available payouts from Marlowe contracts on the Cardano blockchain, simplifying the tracking and withdrawal process
- [Token Plan Prototype](#) – allows token providers to create token plans, where they deposit ada, and release funds over time to a claimer, based on a time-based scheme
- [Order Book Swap Prototype](#) – a decentralized platform for users to list tokens for swap, specifying the desired return. Interested parties can accept offers, resulting in a token swap.

25.5.5. Integrating with Plinth

Marlowe contracts can be integrated with Plinth smart contracts or other compiled languages, such as Aiken, for example. This section focuses on integration with Plinth. One example is the open roles Plinth smart contract, which enables interaction with a Marlowe contract where participants' Cardano addresses are unknown at deployment. When using

open roles, the Marlowe contract sends role tokens to the [open roles](#) Plinth validator script that holds them until an address is specified later. This enables the contract to be verified on-chain before users interact with it. When the user initiates an action, like a deposit or choice, the smart contract assigns them the appropriate role and distributes the role token from the validator script. The developer simply needs to specify the [OpenRoles](#) type when setting contract participants, while Marlowe Runtime manages the rest. The [Open roles in Marlowe](#) GitHub documentation page explains this process in more detail.

Marlowe and Plinth validators can interact as follows:

- PlutusTx validators can run in the same transaction as Marlowe transactions
- The script context contains sufficient information for a Plinth validator to inspect the Marlowe transaction's redeemer, incoming, and outgoing datum
- The Marlowe validator will allow the Plinth transaction to run, as long as the Marlowe validator is not making a payment
- The presence of a datum in the UTXO holding the role token by the Plinth script does not interfere with Marlowe validation
- The Marlowe validators do not need to be modified to run alongside a Plinth script that holds the role token.

25.5.6. Future of Marlowe

The Marlowe Foundation will continue supporting Marlowe by consolidating and extending it for Cardano DApp builders. This includes providing oracles and micropayments, optimizing execution, and supporting runtime monetization. Marlowe offers a lower barrier to entry for DApp development. The new Marlowe DApp starter kit (DSK), which will be maintained by the Marlowe Foundation, will highlight these advancements and streamline onboarding for small and medium-sized

businesses, developers, infrastructure providers, and stake pool operators.

Once the intellectual property rights for Marlowe are moved from IO to the Marlowe foundation, the Marlowe repositories will be moved to an independent GitHub organization, and community activities will be coordinated through the Marlowe special interest group, supported by a new, members-based, non-profit organization. The Marlowe Foundation created the [Marlowe 2025](#) Catalyst application and plans to seek additional funding from Catalyst and other sources.

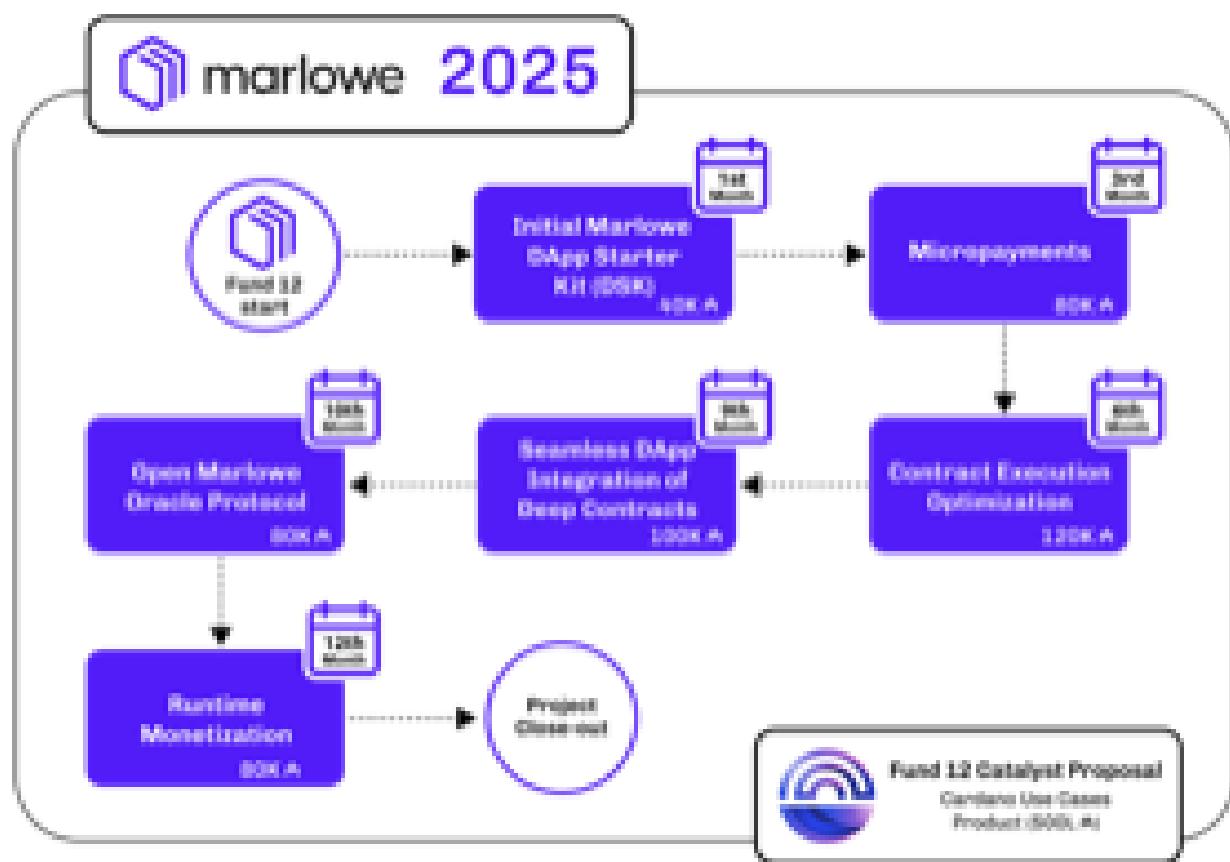


Figure 45. Marlowe 2025 plan

The Marlowe 2025 application targets DApp developers and aims to further Marlowe's development to help businesses leverage emerging market opportunities, making advanced features more accessible and relevant to a broader audience. Marlowe can be particularly attractive to businesses looking to use Cardano because it reduces the risks and costs of initial blockchain onboarding. The application aims to unlock Marlowe's potential by implementing critical updates to its validator and tech stack,

transforming the Marlowe Runtime, protocols, and contracts into competitive products. These updates will create new business opportunities by making Marlowe tools essential for small and medium businesses, developers, infrastructure providers, and stake pool operators (SPOs). The Marlowe 2025 application targets the following areas:

Marlowe DApp starter kit

The Marlowe DApp starter kit (DSK) will consolidate the documentation, examples, and tutorials into an end-to-end guide for designing, implementing, deploying, and maintaining DApps that leverage Marlowe tools and capabilities. The DSK will include comprehensive support materials, such as tutorials, readily available software packages, and pre-built binaries to ensure ease of implementation. The initial DSK will be delivered early in the project, with incremental enhancements added as new technologies are developed.

Validator enhancements

Targeted changes will be implemented to the current validator to significantly reduce on-chain execution costs for specific contracts and Marlowe idioms. By addressing inefficiencies in the existing execution flow, the changes aim to optimize a set of key contracts with business value potential. This process will not only improve efficiency but also expand capabilities, including the integration of off-chain micropayment channels into the framework. These changes will also be incorporated across the whole Marlowe toolset. The selected contracts, chosen for their real-world application potential, will be demonstrated through DApp prototypes, fully documented and accessible via the marlowe-ts-sdk for developers to build upon.

Configurable runtime fee mechanism

To encourage infrastructure providers and SPOs to adopt Marlowe, a configurable fee mechanism will be introduced within the Marlowe

Runtime. This backend enhancement will support the safe execution of Marlowe contracts via web applications, supported by a new transaction validation layer in the TypeScript client library. This layer will ensure secure interactions, even with untrusted backends, facilitating a wider distribution and adoption of Marlowe technology.

Marlowe oracle protocol

The Marlowe oracle protocol will be formalized and expanded with a detailed CIP specification, building on the oracle scanner MVP. This protocol employs a unique on-chain request-response mechanism, offering distinct advantages over traditional feed-based systems by enabling extensive data set coverage. The Marlowe Foundation will collaborate with existing oracle providers on the Cardano network to ensure seamless integration. The protocol will also be made interoperable with other languages such as Aiken or Plinth, allowing scripts to efficiently manage oracle data. This enhancement will not only ensure compatibility with current technologies but also open up numerous new applications on Cardano.

Outreach

The Marlowe 2025 project aims to enhance community engagement through a series of live online presentations, leveraging the previous experience from participating in local and international Cardano summits, meetups, and workshops. These efforts aim not only to educate, but also to foster community collaboration to further drive the technology evolution and refinement.

The Marlowe 2025 proposal marks the first step in launching Marlowe as an independent project. The Marlowe Foundation will make further bids to Catalyst and other funding bodies, engaging with the community through the special Interest group and potentially collaborating with other partners. A key focus will be the development of Marlowe V2 – a separate conceptual track aimed at fundamental language modifications. This track

will be organized through the creation of Marlowe improvement proposals (MIPs), in collaboration with the wider community.

25.5.7. Impact of Marlowe

Marlowe has the potential to evolve into a smart contract technology complementary to Plinth and Aiken.

Marlowe significantly reduces the barriers to entry for new businesses and developers interested in exploring blockchain solutions by simplifying the onboarding process. The technology not only mitigates the complexities of the UTXO model through a user-friendly yet robust programming language, but also drastically lowers costs, reduces risks, and shortens prototyping time for traditional Web2 businesses looking to integrate blockchain functionality. The introduction of marlowe-ts-sdk and the Marlowe Runtime, which integrates with familiar REST APIs, make it straightforward to build end-to-end DApps incorporating Marlowe on- and off-chain together with traditional web frameworks.

The Marlowe 2025 proposal is designed to sustain and enhance this project. To accelerate adoption, Marlowe will be enriched with efficient real-world functionality and innovative features such as off-chain micropayment channels. The technology's success relies on a thriving ecosystem around it; therefore, Marlowe improvements in oracle data availability and infrastructure robustness are critical. By simplifying the initial onboarding process to a single npm install command, a surge in Marlowe's usage is anticipated. In doing this, it will benefit a range of different stakeholders in different ways.

Stakeholder beneficiaries

Marlowe is essential for the Cardano infrastructure. The addition of oracles and micro-payments in the Marlowe product will benefit infrastructure providers by simplifying or expanding their services. Cardano DApp developers will similarly gain the ability to incorporate

Marlowe features into their development solutions.

The Marlowe DSK is designed for small and medium-sized enterprises and developers, enabling quick and effective onboarding. By focusing on developers and lowering barriers throughout the development cycle – from design to deployment – the DSK will streamline access to Marlowe. The inclusion of features like micropayments and oracles will also attract potential users in the decentralized finance (DeFi) space. The simplifications and end-to-end support provided by the DSK will help users quickly learn about Marlowe’s applications and facilitate smoother onboarding.

The proposal is designed to foster a positive feedback loop within the Cardano ecosystem, enhancing the symbiotic relationship between Marlowe technology and its infrastructure. The planned enhancements will not only benefit developers and businesses, but will also create incentives for infrastructure providers (possibly SPOs) and oracle services to engage more deeply with Cardano. This collaborative growth, supported by the Marlowe special interest group and existing forums, will help forge a vibrant and sustainable ecosystem, positioning Marlowe as a cornerstone of the Cardano economy.

25.6. Plutus smart contracts

25.6.1. Overview and learning resources

The *Plutus smart contracts* section presents the rules of the Plutus language by demonstrating how to write and interact with Plinth smart contracts (formerly known as PlutusTx). Plinth is a high-level smart contract language embedded in Haskell that compiles to Untyped Plutus Core (UPLC). Because these rules also apply to other compiled languages such as Aiken, Plu-Ts, and OpShin, learning the basics of Plinth helps developers read and write smart contracts in other languages that target Plutus.

The [2024 Cardano ecosystem report](#) shows that Aiken is the most used smart contract language in the Cardano community. It is a user-friendly standalone language with easy-to-follow and comprehensive documentation. It provides a simple setup of the development environment needed to write, test, and compile smart contracts for Cardano. A more detailed comparison of Aiken and Plinth can be found in the [Plinth in comparison to Aiken](#) section, which also lists Aiken learning resources. Plinth, as a high assurance language embedded in Haskell, works better with the [Agda](#) proof assistants that can be used for the formal verification of smart contract code. The advantages of the Haskell programming language are presented in section [Features and benefits of Haskell](#). IO offers a free self-paced and beginner-friendly [Haskell course](#) that is hosted on GitHub and teaches the necessary skills for programmers to read and write Plinth scripts. The official [Haskell documentation](#) site provides more learning resources.

Before reading section *Plutus smart contracts*, one should have a basic understanding of the extended UTXO (EUTXO) accounting model that is presented in [The EUTXO model](#) section. One should also be familiar with the types of smart contract languages offered by Cardano, covered in the [Smart contract programming languages](#) section, which lists each type and provides resources for development tools. Additionally, understanding Cardano addresses and their binary format, as explained in section [Cardano addresses](#), is important. Plinth code examples presented in this section are based on PlutusV3, which became available after the Chang hard fork. From the [Cardano docs](#) page, we get a short description of what PlutusV3 brings to Cardano:

With the introduction of PlutusV3, Plutus scripts are available for use as part of the governance system, enabling sophisticated voting possibilities like supporting DAOs, for example. Chang also brings advanced Plutus cryptographic primitives, other new primitives, and performance enhancements to the platform. These additions provide developers with a

richer smart contract creation toolkit, enhancing both developer and user experiences, and unlocking new possibilities for decentralized applications (DApps) on Cardano.

Section [PlutusV3 features](#) explains some of the PlutusV3 features that are not covered in the coding sections, and highlights the advantages they bring. The *Plutus smart contracts* section covers the following topics:

- Plutus in comparison to Bitcoin Script and Solidity
- Plinth in comparison to Aiken
- Setting up a Plinth development environment
- Simple validation scripts
- Script context explained
- Time-dependent and parameterized validators
- Off-chain code with MeshJS
- Minting policies and native tokens
- PlutusV3 features.

Plinth validator code and [MeshJS](#) off-chain code presented in the *Plutus smart contracts* section can be found at the [plinth-plutusV3](#) branch of the Plutus Pioneer program (PPP) GitHub repository. This branch contains validator examples of the [4th PPP iteration](#), translated from PlutusV2 to PlutusV3. In addition to MeshJS off-chain code, the *plinth-plutusV3* branch also contains Lucid Evolution off-chain code examples. Further Cardano learning resources and smart contract examples in Plinth, Aiken, and other smart contract languages can be found at:

- [Cardano's smart contract marketplace](#) – hosted by Maestro. The platform provides powerful APIs and advanced software tools to build and scale DApps with ease.
- Gimbalabs [Plutus project-based learning](#) – provides *Course Repositories* that redirect to Plutus example projects. More resources are available

on their [YouTube channel](#).

- Project pages listed in the education sections of [Cardano's interactive map](#). One can also explore projects in other groups and view their GitHub repositories.
- [Cardano for the Masses](#) online book written by John Greene that is available under an open-source license.
- Cardano foundation's free online [Cardano academy](#) course, that teaches the basics of blockchain and Cardano.

Cardano's hard fork combinator technology, briefly introduced in section [Cardano node layers](#), subsection **Consensus and storage layer**, enables the Cardano node to continue processing scripts written in earlier versions of Plutus even after a hard fork introduces a new Plutus version. If you encounter smart contract code examples written in PlutusV1 or PlutusV2, you can check out videos and learning examples of the previous Plutus Pioneer program iterations:

- [third-iteration](#) - PlutusV1 examples
- [fourth-iteration](#) - PlutusV2 examples.

25.6.2. Plutus in comparison to Bitcoin Script and Solidity

This section compares the basic characteristics of smart contracts in Bitcoin, Ethereum, and Cardano, highlighting the new concepts each introduced in the cryptocurrency space.

A Bitcoin Script is a simple stack-based smart contract language, whose most complex control structure is a conditional. It is written in a Forth-like non-Turing-complete language and is essentially linear, which means it can branch, but the language does not offer looping constructs or recursion. All Bitcoin scripts terminate, and it is possible to give an accurate estimate of the time taken to execute a script. Bitcoin scripts have the following developer limitations (taken from [Functional Blockchain Contracts](#), Chakravarty et al. 2019):

1. The Bitcoin Script language constrains programs to be of a limited size and provides barely any control structures (essentially only conditional statements). The primitive operations that can be used in Bitcoin Script are also very limited (for example, the division operation was originally included but was subsequently disabled).
2. The computational context available to a Bitcoin Script program is very constrained. For example, it cannot inspect a transaction currently being validated, but it can access the transaction's hash.

A compilation pipeline is being developed at IO that builds a connection from Cardano smart contracts to Bitcoin. This is done by combining Untyped Plutus Core (UPLC), the CEK machine - a clever interpreter architecture, and RISC-V that is a widely supported open-source reduced instruction set architecture. BitVMX plays a key role, that makes Bitcoin programmable. It runs computations off-chain, but gives Bitcoin users a way to successfully challenge suspicious transactions. One can read more about this topic in the following [IO blog](#).

Ethereum provides a Turing-complete language for the Ethereum virtual machine (EVM), which is the runtime environment for transaction execution in Ethereum. It also provides Solidity, a custom higher-level programming language that compiles into EVM code. Solidity is an object-oriented, statically typed programming language designed for developing smart contracts. It supports inheritance, libraries, and complex user-defined types. Gas fees must be paid for every smart contract transaction on Ethereum. A Solidity smart contract is able to see and access information from the entire global state of the blockchain. That is the opposite extreme of Bitcoin, where the script has very little context. This enables Ethereum smart contracts to be more powerful, but it can also make it difficult to predict their behavior, leading to potential security issues. During the interval between a user constructing a transaction and its incorporation into the blockchain, concurrent events may also occur. In Ethereum, it is possible that a user has to pay gas fees for a transaction

that interacts with a smart contract, although it can eventually fail with an error. Ethereum employs an account-based model, where each user has an account with a balance. When funds are transferred between accounts, the balances are updated accordingly.

Cardano uses the EUTXO model and Plutus as its native smart contract language. Developers can write smart contracts in Plinth or in other high-level languages that compile to Plutus (see section [Smart contract programming languages](#)). Section [Plutus security](#) covers the security aspect of the Plutus language in more detail. A Plutus smart contract, also called a Plutus script or validator, provides a more flexible view than a Bitcoin script, but does not have a global view as a Solidity smart contract. Plutus scripts cannot see the whole state of the blockchain, but can see the whole transaction that is being validated. This means a Plutus script can see the script context that carries information about the transaction being validated. This includes all transaction inputs and outputs and other transaction data.

A UTXO can have an arbitrary piece of data attached to it, the datum, that can define the state of the UTXO. When a transaction intends to spend a UTXO sitting at a script address, the script code can access the information contained in the datum that is potentially present in the script context. Not every UTXO necessarily needs to contain a datum. The information in the datum can be used for:

- Indicating who can consume a UTXO, when, and under what conditions
- Representing the current state of the UTXO
- Defining metadata and/or configurations.

The script also sees a piece of arbitrary input data provided by the user submitting the transaction. This data is the redeemer, which is also contained in the script context of a transaction. The information in the redeemer can be used for:

- Indicating the purpose of interacting with a script, eg, placing a bet, paying a fee, or claiming a reward
- Providing information known only to a specific party, which can be used to unlock funds held at a script address
- Supplying a value that modifies the current datum data.

Section [Script context explained](#) details the information contained in the script context that a transaction interacting with a script can access. A Plutus script can use all this information to decide whether it is ok to consume a UTXO or not. It is also possible to use a tool like the Cardano CLI or an off-chain code library to check whether a transaction will be validated before on-chain execution. If the transaction is valid, it will be processed on the network, given that all UTXO inputs are present and the processing time falls within the transaction's validity interval. In case these conditions are not met, the transaction will fail without charging the user any fees, unlike in Ethereum, where users must pay gas fees for transactions that can ultimately fail. We say that transactions on Cardano are deterministic. This does not mean we know for sure if a transaction will be processed because a UTXO input might be consumed by another transaction just before our transaction gets processed. It means that if the transaction gets processed, the effect of the transaction on the ledger is deterministic and can be computed off-chain.

Ethereum's accounting and Cardano's EUTXO models are equally powerful in expressing smart contract logic. However, translating a smart contract from Ethereum to Cardano requires rethinking both the design and implementation. Rewriting the exact logic of an Ethereum smart contract to Cardano will likely generate an undesired result. A better approach is to analyze the original contract and produce a high-level specification of what it does. Then, create a new EUTXO-based architecture based on that specification to have the same intended properties as the original contract.

Further comparison between Cardano and Ethereum is outlined in section

[Cardano security](#), which highlights some of the security advantages that Cardano's EUTXO model brings over Ethereum's EVM model. You can find research articles about the technical implementation of Plutus in the [IO library](#). A good entry point is the research paper [Functional Blockchain Contracts](#), 2019 by Chakravarty et al. The official [Plutus documentation](#) provides short explainers and various learning resources. The [Plutus GitHub repository](#) also provides links to specifications, design documents, academic papers, and talks.

25.6.3. Plinth in comparison to Aiken

Plinth enables developers to write and compile their on-chain code to Plutus, or more precisely to Untyped Plutus Core (UPLC), which the Cardano nodes execute. Plinth provides a compiler plugin for the Glasgow Haskell Compiler (GHC), which compiles to Plutus. More details about the Plinth-to-Plutus compilation pipeline are outlined in section [Plutus security](#).

Plinth lets developers build secure applications, forge new assets, and create smart contracts in a predictable, deterministic environment with the highest level of assurance. Furthermore, developers don't have to run a full Cardano node to test their work. The tools that allow testing Plinth scripts are outlined in the [Plutus security](#) section. The [Plutus](#) repository includes the Plinth compiler (previously called PlutusTx), enabling developers to write Haskell code that can be compiled to Plutus. The repository also includes the combined [documentation](#), generated using Haddock, for all public Plutus code libraries. The official [Plinth user guide](#) provides developer-related information on Plutus and Plinth.

[Aiken](#) is another popular Cardano smart contract language written in Rust, which directly compiles to UPLC. This process is illustrated in the diagram below:

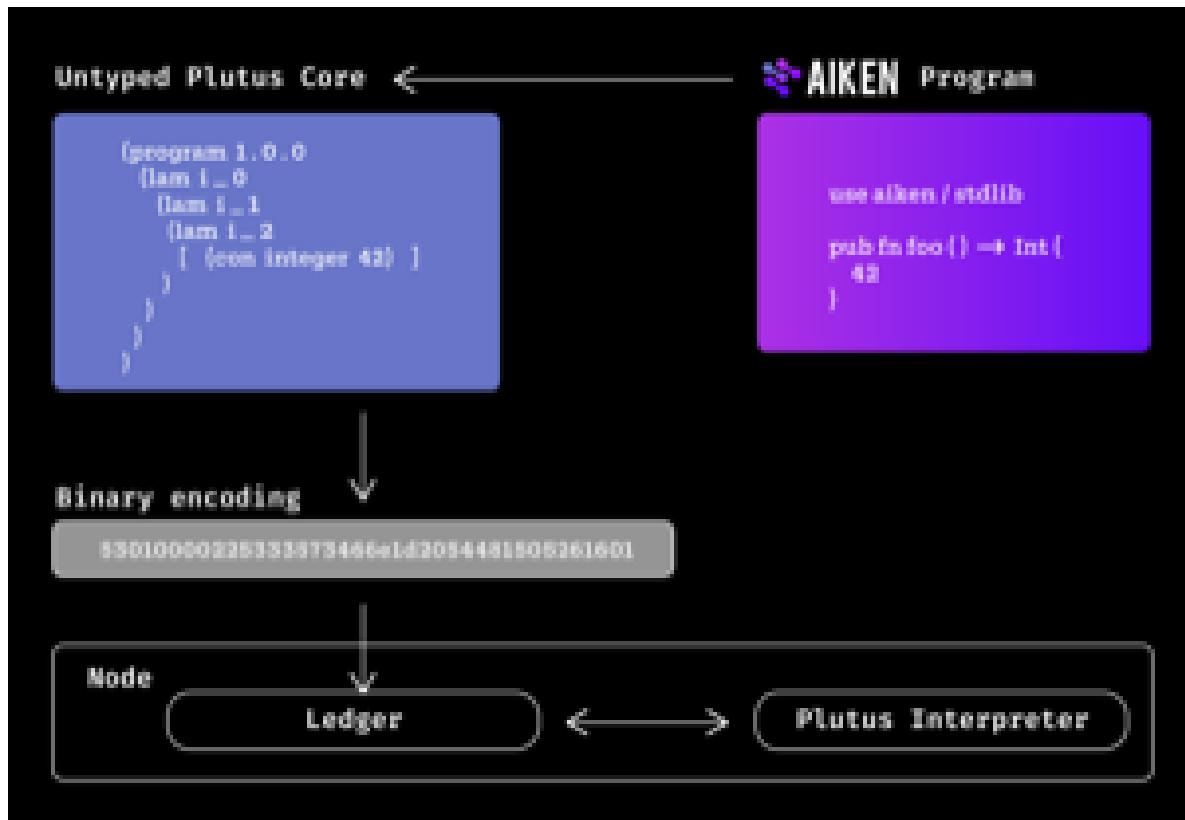


Figure 46. Aiken code transformation, source: [CF blog](#)

Aiken was developed with the support of [TxPipe](#) and the [Cardano Foundation](#). Currently, IO is also supporting the development of Aiken. More information about maintenance and support can be found at the [official page](#). Aiken takes inspiration from many modern languages, such as Gleam, Rust, and Elm, which are known for friendly error messages and an overall excellent developer experience. It is a purely functional language with strong static typing and inference. It offers a more accessible and familiar syntax to developers, which makes it easy to learn. Aiken enables cost-efficient smart contract development and comes with a modern development environment that has a package manager, helpful error diagnostics, a language-server protocol (LSP) with auto-formatting, and popular editor integration (VSCode, NeoVim, Emacs). The language is well documented and offers a built-in testing framework that ensures proper and robust smart contract execution with property-based testing. Aiken's testing framework uses the same underlying virtual machine as in real smart contract execution, ensuring that memory consumption and contract behavior during testing are identical to those on the mainnet.

Plinth has a more expressive type system than Aiken. It supports basic Haskell features such as regular algebraic data types, type classes, higher order functions, parametric polymorphism, etc. However, it doesn't support many of Haskell's more advanced features. For more information, see the *Differences from Haskell* section in the official [Plinth user guide](#). Previously, Plinth compiled and encoded data types using the [Scott-Encoding](#) approach ([Haskell example](#), see section *In Haskell. Scott encoding*), which created an upfront cost that caused the difference in execution costs between Haskell and other frameworks like Aiken. With the introduction of PlutusV3, sums of products allow for encoding data types in a more compact and cost-effective way. Also, Plutus now provides a way to deal directly with the [Data](#) type for all Plutus script versions (V1, V2, and V3), allowing for better code optimization. The [Plinth user guide](#) provides instructions on that. Aiken compiles all its custom data types into the built-in Plutus type [Data](#). We will look at this data type in section [Simple validation scripts](#).

Both languages are purely functional, which makes it easier for an auditor to assess the behavior of smart contracts written in them. Because Plinth is embedded in Haskell, it works well with the [Agda](#) proof-assistant tools, making it a good choice for projects that require a high level of security. You can read more about the topic of formally verifying Haskell code with the Agda proof assistant in the scientific paper [Reasonable Agda Is Correct Haskell: Writing Verified Haskell using agda2hs](#). Since Agda isn't Haskell-specific, it could also be used with Aiken. However, the gap to bridge between Aiken and Agda is larger than that between Plinth and Agda.

Both Plinth and Aiken compile to Plutus, so the logic of writing such smart contracts is very similar, even though the syntax is different. Learning one language will help a developer understand any other Cardano smart contract language that compiles to Plutus. To learn the Aiken smart contract language, one can check out the [Language tour](#) on the official documentation, which also provides a walkthrough of creating a [Hello](#)

[World project](#), covering the following topics:

- Pre-requisites
- Scaffolding
- Using the standard library
- The first validator
- Adding traces
- Writing a test
- End-to-end off-chain code (MeshJS and PyCardano).

The [Awesome Aiken](#) repository provides many links to Aiken libraries, DApps, tutorials, and video lessons. For those who prefer to learn from a book, the [I can Aiken](#) book, written by John Greene provides comprehensive information and learning examples.

25.6.4. Setting up a Plinth development environment

For your development environment, you can use one of the following options:

- Run a Nix shell that contains all the dependencies
- Run a local Docker container inside VSCode
- Use an online platform such as [Demeter.run](#)
- Install system dependencies manually.

The [plinth-template](#) GitHub repository provides instructions for the above mentioned options. The smart contract code presented throughout section [Plutus smart contracts](#) uses features introduced in the Chang hard fork, which enabled PlutusV3. The development environment you will use to run and test this code needs to support these features. The smart contract code was compiled with a Nix shell. Nix version [2.25.3](#) was used and a Nix shell provided by the [plinth-template](#) repository, commit [b9460088985331bb050f1782a32e4f92c4c00e67](#).

The [Demeter.Run](#) online platform offers various tools and development environments for the Cardano ecosystem. One can set up development environments for smart contract languages such as Plinth and Aiken, as well as programming languages like Rust or Python, come with useful Cardano tools and libraries written in those languages. The platform offers backend hosting for DApps and allows integration testing within its environments. It manages monitoring, security, and version upgrades. Depending on development needs, the platform provides various starter kits, including example code repositories from the community for learning or project initiation. Its pricing model is based on service usage, with users also able to access some free working time. Explore *Developer Tools* and *Infrastructure* sections on the [Cardano developer map](#) for more options.

To query various Plutus types, one can use the official [Plutus Haddock documentation](#), which presents types in Haskell syntax. Press CTRL + S to search for a keyword, such as a specific data type or function name. The search engine might provide more options for a single data type, function or type class you are querying. This is because implementations of types, functions, and type classes can change with a new Plutus version and the Haddock documentation keeps track of all of them. After locating the desired item, click the *Source* icon next to its name to open the corresponding Haskell source code. The software packages for the libraries hosted on the Plutus haddock documentation are contained in the [Haskell packages for Cardano](#) repository. It contains all Haskell packages used by Cardano that are not hosted on [Hackage](#) – the central archive for Haskell packages. Other compiled languages provide their own libraries that implement Plutus types. One example is the official [standard library](#) for the Aiken language.

Plinth data types cannot be explored in the standard GHCI REPL because the GHC compiler pipeline first compiles Plinth code to an intermediate language called Haskell Core. The Plutus compiler then takes this Haskell Core and compiles it to the Plutus script language. This means that GHC is

unaware of Plinth data types. Additionally, the Plinth libraries are not hosted on Hackage, which means the only way to query Plinth types from a REPL is to build it with a Cabal file that imports those libraries. There are some intermediate steps in the compilation pipeline which are explained in detail in section [Plutus security](#).

25.6.5. Simple validation scripts

In this section, we will look at basic Plinth validation scripts. All validation scripts and data types presented in the [Plutus smart contracts](#) section are PlutusV3 scripts and data types. An exception are PlutusV1 and V2 data types that were not modified with the introduction of PlutusV3 and can be still used in PlutusV3 scripts. A validation script or validator is the smart contract program that checks whether funds at a script address are allowed to be spent by a given transaction. As stated previously, a validation script in EUTXO cannot see the entire state of the blockchain; instead, it can view the entire transaction being validated. The script uses a single parameter: the script context. It returns a parameter that indicates whether the validation logic has passed or not. Depending on the type of the script context, there are two possible implementations of a Plinth validation script:

- In the low-level implementation, the script context is represented using the [BuiltinData](#) type, and the return parameter is of type [BuiltinUnit](#).
- In the high-level implementation, the script context is represented as a predefined Haskell type, and the return parameter is of type [Bool](#).

Below you can see two example validation script type signatures, one low-level and one high-level:

```
validatorName :: BuiltinData -> BuiltinUnit  
validatorName :: ScriptContext -> Bool
```

Both implementations can be used in smart contract code. The main

difference lies in code performance, with the low-level implementation offering better performance. Low-level validation scripts are referred to as *untyped validation scripts*, while high-level scripts are known as *typed validation scripts*.

The `BuiltinData` doesn't have its constructors exposed. The module that defines `BuiltinData` contains two conversion functions: `builtinDataToData` and `dataToBuiltinData`, that can convert `BuiltinData` back and forth to the `Data` type. They can be used in off-chain code, but not in on-chain code. The `Data` type has its constructor exposed, as illustrated in the code below:

```
data Data
  = Constr Integer [Data]
  | Map [(Data, Data)]
  | List [Data]
  | I Integer
  | B BS.ByteString
deriving stock (Show, Read, Eq, Ord, Generic, Data.Data.Data)
deriving anyclass (Hashable, NFData, NoThunks)
```

It is a recursive data type that contains integers and byte strings, which can be packed into lists and maps. Next, we show the Haskell implementation of the script context data type:

```
data ScriptContext = ScriptContext
  { scriptContextTxInfo    :: TxInfo
  , scriptContextRedeemer :: V2.Redeemer
  , scriptContextScriptInfo :: ScriptInfo
  }
deriving stock (Generic, Haskell.Eq, Haskell.Show)
deriving anyclass (HasBlueprintDefinition)
```

The script context contains:

- transaction information (inputs, outputs, validity interval, etc)
- redeemer (arbitrary data defined by the user)

- script info (defines the purpose of the script; if the purpose is a spending script, it potentially contains a datum).

We will look into those data types in more detail in section [Script context explained](#). If a transaction tries to spend multiple UTXOs at a script address, the spending script is run for every UTXO individually. The redeemer and the script information are individually assigned for every UTXO in that transaction. The transaction information is only one and is accessible to every script instance that is being triggered by a single transaction.

Let us now examine a simple untyped validator script that always succeeds:

```
{-# LANGUAGE DataKinds          #-}
{-# LANGUAGE DeriveAnyClass     #-}
{-# LANGUAGE DeriveGeneric       #-}
{-# LANGUAGE DerivingStrategies  #-}
{-# LANGUAGE FlexibleInstances   #-}
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
{-# LANGUAGE ImportQualifiedPost  #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE OverloadedStrings    #-}
{-# LANGUAGE PatternSynonyms     #-}
{-# LANGUAGE ScopedTypeVariables  #-}
{-# LANGUAGE Strict              #-}
{-# LANGUAGE TemplateHaskell      #-}
{-# LANGUAGE TypeApplications     #-}
{-# LANGUAGE UndecidableInstances  #-}
{-# LANGUAGE ViewPatterns         #-}
{-# LANGUAGE NoImplicitPrelude    #-}
{-# OPTIONS_GHC -fno-full-laziness #-}
{-# OPTIONS_GHC -fno-ignore-interface-pragmas #-}
{-# OPTIONS_GHC -fno-omit-interface-pragmas #-}
{-# OPTIONS_GHC -fno-spec-constr  #-}
{-# OPTIONS_GHC -fno-specialise  #-}
{-# OPTIONS_GHC -fno-strictness  #-}
{-# OPTIONS_GHC -fno-unbox-small-strict-fields #-}
{-# OPTIONS_GHC -fno-unbox-strict-fields #-}
{-# OPTIONS_GHC -fplugin-opt PlutusTx.Plugin:target-version=1.1.0 #-}
```

```

module Week02Validators where

import GHC.Generics
import PlutusLedgerApi.Common
import PlutusLedgerApi.V3
import PlutusTx
(unsafeFromBuiltinData),
import PlutusTx.Builtins
import PlutusTx.Bool
import PlutusTx.Prelude
compile, makeIsDataSchemaIndexed)
(unsafeDataAsI)
(Bool(..))
(BuiltinUnit, Eq(..), Integer,
Maybe(..), check, traceError,
traceIfFalse, ($), otherwise,
(.))
import PlutusTx.Blueprint
(HasBlueprintDefinition)
import PlutusTx.Blueprint.Definition (definitionRef)
import qualified PlutusTx.Builtins.Internal as BI (
    BuiltinList, BuiltinInteger,
head, snd, tail, unitval,
unsafeDataAsConstr)

{- -----
--- -}
{- ----- Always True validator
----- -}

{-# INLINEABLE mkGiftValidator #-}
mkGiftValidator :: BuiltinData -> BuiltinUnit
mkGiftValidator _ctx = check True

compiledMkGiftValidator :: CompiledCode (BuiltinData -> BuiltinUnit)
compiledMkGiftValidator = $$ (compile [| mkGiftValidator |])

serializedMkGiftValidator :: SerialisedScript
serializedMkGiftValidator = serialiseCompiledCode
compiledMkGiftValidator

```

The long list of language pragmas, GHC compiler options, and import statements is necessary to cover the functionality for all validators presented in the [Simple validation scripts](#) section. From the language pragmas, we note that we add the `NoImplicitPrelude` extension that prevents the standard `Prelude` module from being imported. The `PlutusTx` module defines a custom prelude where all functions use strict evaluation rather than lazy evaluation. This also applies to all other functions defined in any module used for Plinth coding. One can read more about GHC language pragmas at the [Haskell wiki](#), which provides links to official GHC documentation. The [GHC docs](#) also cover various GHC compiler options. We name our module `Week02Validators`. The naming of modules through the [Plutus smart contracts](#) code examples section follows the naming from the [fourth iteration](#) of the Plutus pioneer program. Next, we import several submodules from the `PlutusLedgerApi` and the `PlutusTx` modules that define functions for working with PlutusV3 scripts.

After that follows the on-chain validator code. We name the validator function `mkGiftValidator` that means *make gift validator*. It is an untyped validator that always succeeds. It takes in the script context as the only argument, which is of type `BuiltinData` and returns something of type `BuiltinUnit`. In the body of the function, all input values are ignored, and we use the `check` function which takes a value of type `Bool`, and returns a value of type `BuiltinUnit` if the input argument is true, or raises an error if it is false.

```
check :: Bool -> BI.BuiltinUnit
check b = if b then BI.unitval else traceError checkHasFailedError

unitval :: BuiltinUnit
unitval = BuiltinUnit()

data BuiltinUnit = BuiltinUnit ~() deriving stock Data
```

The `check` function returns the `unitval` variable that is a wrapper around

the unit (empty tuple) type. The unit type in Haskell can be viewed as an analogy to the Void type used in other programming languages like Java or C++. If an untyped validator returns the `BuiltinUnit` type the validation logic passes, and if an error is raised, the validation logic fails. It is now clear why this validator is called `Gift` – anyone can claim funds from this address, since the validation will always succeed.

Next, we compile the validator. We use the `PlutusTx.compile` function that takes a syntax tree of a function as input, which we can get if we put the Oxford brackets `[| | mkGiftValidator | |]` around the validator function. The `compile` function produces another syntax tree written in the Plutus language. Then the `$$` symbol, called splice, takes a syntax tree and splices it back to Haskell source code. The splice operator and the Oxford brackets can be used because we added the `TemplateHaskell` language pragma, which enables this language extension.

It is important to note that normally in Oxford brackets, you cannot reference anything defined outside of them. This can become an issue when validator functions are long expressions or when library functions are called within their body. A workaround is to make the function inlinable. By adding the `INLINABLE` pragma statement before or after the function definition, the GHC compiler replaces the function call in the Oxford brackets with the actual function body.

This completes the on-chain code. Next, we use the `serialiseCompiledCode` helper function that returns something of `ShortByteString` type, which is a compact representation of a `Word8` vector. At the end of this section, we present the code that lets us generate `Plutus blueprints` for all the validators presented in this section. Plutus blueprints allow documenting Plutus validators in `JSON` format and include the compiled validator code that is represented as a `CBORHEX`. The `Concise Binary Object Representation (CBOR)` is a binary data serialization format loosely based on `JSON`. The `CBORHEX` is a hexadecimal representation of this data format. For our `mkGiftValidator` the blueprint will show the following `CBORHEX`:

```
"compiledCode": "450101002499"
```

Since our contract is simple, the **CBORHEX** value is short. For more complex contracts, this value would increase in length. This compiled code can then be used in off-chain code when attaching the validator to a transaction. Let us now explore an example where the validation logic always fails, regardless of the input.

```
{-  
-----  
---- -}  
{- ----- Always False validator  
----- -}  
  
{-# INLINEABLE mkBurnValidator #-}  
mkBurnValidator :: BuiltinData -> BuiltinUnit  
mkBurnValidator _ctx = traceError "it burns!!!"  
  
compiledMkBurnValidator :: CompiledCode (BuiltinData -> BuiltinUnit)  
compiledMkBurnValidator = $$compile [||mkBurnValidator||]  
  
serializedMkBurnValidator :: SerialisedScript  
serializedMkBurnValidator = serialiseCompiledCode  
compiledMkBurnValidator
```

We call the validator **mkBurnValidator**, as no funds can be reclaimed from the contract once sent. The **traceError** function can produce an error and log a message displayed by the failed off-chain code transaction that tries to claim any funds from this script address. Also, in this case, we ignore the entire script context. Then we compile the validator and serialize it. We could also use the **check** function with a **False** argument, but in that case, no error message would be logged.

Next, we show an example where we make use of the redeemer in the validation logic. The redeemer type is defined as a wrapper around the **BuiltinData** data type:

```

newtype Redeemer = Redeemer {getRedeemer :: BuiltinData}
    deriving stock (Generic, Haskell.Show, Typeable)
    deriving newtype (Haskell.Eq, Haskell.Ord, Eq, ToData, FromData,
                      UnsafeFromData, Pretty)
    deriving anyclass (NFData, HasBlueprintDefinition)

```

We define an untyped validator that says the validation logic passes if the redeemer is an integer with the value 42; otherwise, it fails:

```

{-
-----
----- -}
{- ----- 42 validator untyped large CBOR
----- -}

{-# INLINEABLE mk42ValidatorLarge #-}
mk42ValidatorLarge :: BuiltinData -> BuiltinUnit
mk42ValidatorLarge ctx
| r == 42    = BI.unitval
| otherwise = traceError "Expected 42 integer redeemer"
where
  ctxTyped = case fromBuiltinData ctx of
    Just @ScriptContext c -> c
    Nothing -> traceError "ScriptContext could not be converted from
BuiltinData"
    r = case fromBuiltinData $ getRedeemer (scriptContextRedeemer
ctxTyped) of
      Just @Integer n -> n
      Nothing -> traceError "Redeemer is not a number"

compiledMk42ValidatorLarge :: CompiledCode (BuiltinData -> BuiltinUnit)
compiledMk42ValidatorLarge = $$compile [|mk42ValidatorLarge|]

serializedMk42ValidatorLarge :: SerialisedScript
serializedMk42ValidatorLarge = serialiseCompiledCode
compiledMk42ValidatorLarge

```

In the example above, we first convert the untyped script context to its typed form using the `fromBuiltinData` function. It takes in a variable of type `BuiltinData` and returns a `Maybe` type parameterized with a type

variable. If that type variable is a type that has an instance of the `FromData` type class, then the conversion will succeed. Otherwise, the function returns `Nothing`. You can find class instances for various Plutus types in the Plutus [Haddock documentation](#). There also exists the `unsafeFromBuiltinData` function that is defined in the `UnsafeFromData` type class. This function directly returns a type variable instead of wrapping it in a `Maybe` type. If the conversion fails, the function raises an error, and the validation logic fails. We call it *unsafe* since the conversion might fail with an error. Below you can see the `FromData` and `UnsafeFromData` type classes.

```
class FromData a where
    fromBuiltinData :: BuiltinData -> Maybe a

class UnsafeFromData a where
    unsafeFromBuiltinData :: BuiltinData -> a
```

Once we have the script context in typed form, we can access the redeemer, a wrapper around the `BuiltinData` type. Then we again try to convert that type, in this case to an `Integer`. Now that the redeemer is in the correct form, we can simply check if it is equal to 42 and return the `unitval` variable or raise an error and log a message. We could also use the `check` function as we did in our previous examples. After that, we compile and serialize the validator.

The reason why we stated "*large CBOR*" in the comment above the code is that converting the script context into a typed form is a costly operation and produces a large CBOR. Next, let us look at the same validator in untyped form, where we decode the script by keeping it in the form of the `BuiltinData` type. At the end of this section, we will compare `CBORHEX` lengths for all 42 validators presented in this section.

```
{-
----- -
----- }
```

```

{----- 42 validator untyped small CBOR
----- -}

{-# INLINEABLE mk42ValidatorSmall #-}
mk42ValidatorSmall :: BuiltinData -> BuiltinUnit
mk42ValidatorSmall ctx
| redeemerInt == 42 = BI.unitval
| otherwise          = traceError "Expected 42 integer redeemer"
where
  -- Lazily decode script context up to redeemer;
  -- is less expensive and results in much smaller tx size
constrArgs :: BuiltinData -> BI.BuiltinList BuiltinData
constrArgs = BI.snd . BI.unsafeDataAsConstr

scriptContextBL :: BI.BuiltinList BuiltinData
scriptContextBL = constrArgs ctx

redeemerBD :: BuiltinData
redeemerBD = BI.head . BI.tail $ scriptContextBL

redeemerInt :: BI.BuiltinInteger
redeemerInt = unsafeDataAsI redeemerBD

compiledMk42ValidatorSmall :: CompiledCode (BuiltinData -> BuiltinUnit)
compiledMk42ValidatorSmall = $$compile [|mk42ValidatorSmall|]

serializedMk42ValidatorSmall :: SerialisedScript
serializedMk42ValidatorSmall = serialiseCompiledCode
compiledMk42ValidatorSmall

```

First, we define a function called `constrArgs` that helps us to convert the `BuiltinData` to the `BuiltinList` type. We recall that the script context is an algebraic data type that combines the transaction info, redeemer, and script info types with a logical `AND`. Once we have converted the script context to a built-in list, we can use the `tail` and `head` functions to extract the needed element. In the case of the script context, the redeemer is in the second place, so we first apply the `tail` function and then the `head` function. Once we have the redeemer in the form of `BuiltinData`, we can use the `unsafeDataAsI` that converts it to an integer. It is called *unsafe* because it raises an error if the conversion fails. Next, we compile and

then serialize the validator.

We note that if the redeemer were a more structured algebraic data type, we could further decode it with the `constrArgs`, `tail` and `head` functions, or we could use the `fromBuiltinData` function to convert it to its typed form and access the elements we need.

Let us look now at the same validator that is written in a typed form.

```
{-
----- -
{-
----- 42 validator typed
----- -}

{-# INLINEABLE mk42TypedValidator #-}
mk42TypedValidator :: ScriptContext -> Bool
mk42TypedValidator ctx = traceIfFalse "Redeemer is a number different
than 42"
                                         $ 42 == r
where
  r = case fromBuiltinData $ getRedeemer (scriptContextRedeemer ctx) of
    Just @Integer n -> n
    Nothing -> traceError "Redeemer is not a number"

compiledMk42TypedValidator :: CompiledCode (BuiltinData -> BuiltinUnit)
compiledMk42TypedValidator = $$compile [| wrappedVal |])
where
  wrappedVal :: BuiltinData -> BuiltinUnit
  wrappedVal ctxUntyped = check $ mk42TypedValidator
  (unsafeFromBuiltinData ctxUntyped)

serializedMk42TypedValidator :: SerialisedScript
serializedMk42TypedValidator = serialiseCompiledCode
  compiledMk42TypedValidator
```

The type signature now changes to `ScriptContext → Bool`, which means the validator succeeds if the logic returns `True` and fails if it returns `False`. We can now access the redeemer directly from the script context in its typed form and then try to convert the data inside the redeemer from

`BuiltinData` to `Integer`. The issue we have now is that we can no longer compile our validator because it is in typed form. So we wrap it such that we use the `unsafeFromBuiltinData` that converts the script context from its untyped form to typed, and then apply the `mk42TypedValidator` and `check` functions. After that, we can compile the wrapped validator and serialize it.

At the end, we can look at one final 42 validator example where we define a custom redeemer type that is a wrapper around an `Integer` type.

```
{-
----- -
{- ----- 42 validator custom type
----- -}

-- Custom data types for our redeemer
data MySillyRedeemer = MkMySillyRedeemer Integer
    deriving stock (Generic)
    deriving anyclass (HasBlueprintDefinition)

makeIsDataSchemaIndexed ''MySillyRedeemer [(MkMySillyRedeemer, 0)]

{-# INLINEABLE mk42CustomValidator #-}
mk42CustomValidator :: ScriptContext -> Bool
mk42CustomValidator ctx = traceIfFalse "Redeemer is a number different
than 42"
    $ 42 == r
where
    r = case fromBuiltinData @MySillyRedeemer . getRedeemer $
        scriptContextRedeemer ctx of
            Just (MkMySillyRedeemer rInt) -> rInt
            Nothing -> traceError "Redeemer is not of MySillyRedeemer type."

compiledMk42CustomValidator :: CompiledCode (BuiltinData -> BuiltinUnit)
compiledMk42CustomValidator = $$compile [||wrappedVal||]
where
    wrappedVal :: BuiltinData -> BuiltinUnit
    wrappedVal ctx = check $ mk42CustomValidator (unsafeFromBuiltinData
ctx)
```

```

serializedMk42CustomValidator :: SerialisedScript
serializedMk42CustomValidator = serialiseCompiledCode
compiledMk42CustomValidator

```

Here, we first define the custom data type `MySillyRedeemer` which is a wrapper around the `Integer` type. Then we use the `makeIsDataSchemaIndexed` function that generates the `ToData`, `FromData`, `UnsafeFromData` and `HasBlueprintSchema` instances for our custom type, which contain functions for converting our custom data type to the `Data` type back and forth. We use template Haskell, which requires adding two single quotes in front of the type that returns the type's name. After we have defined our redeemer type, we write the validator logic in the same way as in the previous example, just that we now convert the redeemer from `BuiltinData` to the `MySillyRedeemer` type. We compile and serialize the validator in the same way as in the previous example.

Altogether, we have defined four different variants for the 42 validator. We note that if we were to import the `ScriptContext` data type from the `PlutusLedgerApi.Data.V3` module instead of the `PlutusLedgerApi.V3` module, the `CBORHEX` lengths decrease by around 10%. The table below shows compiled code lengths for the four validators and for the two different `PlutusLedgerApi` modules we can use.

42 validator	PlutusLedgerApi .V3	PlutusLedgerApi.Da ta.V3	Reductio n
Untyped (converting ScriptContext to typed form)	13842	12362	10.7%
Untyped (decoding ScriptContext as BuiltinData)	64	64	0%
Typed (Integer type redeemer)	5887	5248	10.8%

42 validator	PlutusLedgerApi.V3	PlutusLedgerApi.Data.V3	Reduction
Typed (custom type redeemer)	6066	5428	10.5%

We see that we get by far the most compact compiled code if we decode the `ScriptContext` in its untyped form and then convert the part we need to typed form. We call that lazy decoding. This also brings an on-chain code performance advantage. The largest `CBORHEX` is the one for the untyped validator, where we convert the entire `ScriptContext` into typed form. The reason is this type carries much information and convert the whole type into typed form is a costly operation.

The reason the compiled code is shorter if we import the `ScriptContext` from the `PlutusLedgerApi.Data.V3` module is that module provides an alternative interface that works directly with the Plutus Core `Data` type under the hood. Because of the updates to Plutus, data types can now be thin wrappers over the `BuiltinData` type, which allows retaining the user-friendliness of the data type version while also avoiding the upfront cost of decoding the `BuiltinData` into sums of products. We call a data type *data-backed* if it is representationally equivalent to `BuiltinData`.

Since Plutus now provides a way to deal directly with the `Data` type for all Plutus script versions (V1, V2, and V3), developers can move away from sums of products or Scott encoding. The [Plinth user guide](#) provides instructions on how to optimize scripts with the `PlutusTx.asData` module that contains template Haskell (TH) code for encoding algebraic data types (ADTs) as `Data` objects in Plutus Core such that they become *data-backed* types. The `PlutusLedgerApi.Data.V3` module already contains the `ScriptContext` in the form of a *data-backed* type. Also, one can look at the [Simplifying code before compilation](#) and [Other optimization techniques](#) guidelines.

We again state the various possibilities that a redeemer can be used for in

smart contracts:

- Indicating the purpose of interacting with a script, eg, placing a bet, paying a fee, or claiming a reward
- Providing information known only to a specific party, which can be used to unlock funds held at a script address
- Supplying a value that modifies the current datum data.

At the end, we show the code for generating Plutus blueprints for two of the six validators we have defined in this section to shorten the code. We choose the gift validator and the untyped 42 validator that generates a small CBOR.

```
{-# LANGUAGE DataKinds          #-}
{-# LANGUAGE DerivingStrategies  #-}
{-# LANGUAGE FlexibleContexts   #-}
{-# LANGUAGE FlexibleInstances   #-}
{-# LANGUAGE GADTs              #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE OverloadedStrings   #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE TypeApplications    #-}
{-# LANGUAGE UndecidableInstances #-}

module Main where

import qualified Data.ByteString.Short      as Short
import qualified Data.Set                   as Set
import           PlutusTx.Blueprint
import qualified Week02Validators          as Week02

{-
----- -
{-
----- ----- ENTRY POINT
----- -
}

main :: IO ()
main = writeBlueprint "blueprint.json" blueprint
```

```

{- -----
----- -}
{- ----- SHARED
----- -}

blueprint :: ContractBlueprint
blueprint =
    MkContractBlueprint
        { contractId = Just "plutus-pioneer-program"
        , contractPreamble = preamble
        , contractValidators =
            Set.fromList
                [ mkGiftVal
                , mk42ValSmall
                ]
        , contractDefinitions =
            deriveDefinitions
                @[ ()]
                , Integer
                ]
        }
    }

preamble :: Preamble
preamble =
    MkPreamble
        { preambleTitle = "Plutus Pioneer Program Blueprint"
        , preambleDescription = Just "Blueprint for the Plutus Pioneer
Program validators"
        , preambleVersion = "1.0.0"
        , preamblePlutusVersion = PlutusV3
        , preambleLicense = Just "MIT"
        }

```

We first define all language pragmas and import the necessary modules, including the `Week02.Validators` module, where we have defined our validators from this section. In the `main` function, we write the blueprint to a `JSON` file, and after that comes the blueprint definition. It contains the contract ID, the preamble that defines some general information, the contract validators, which we will define next, and the contract definitions. The definitions include a list of types that we have used in the

validator code. After that, we see the `preamble` definition. Next, we show the validator blueprint definitions for two validators that we have chosen.

```
{-
----- -
{ - ----- VALIDATORS - WEEK02
----- -}

mkGiftVal :: ValidatorBlueprint referencedTypes
mkGiftVal =
  MkValidatorBlueprint
    { validatorTitle = "Always True Validator"
      , validatorDescription = Just "Validator that always returns True
(always succeeds)"
      , validatorParameters = []
      , validatorRedeemer =
        MkArgumentBlueprint
          { argumentTitle = Just "Redeemer"
            , argumentDescription = Just "Redeemer for the always true
validator"
            , argumentPurpose = Set.fromList [Spend, Mint, Withdraw,
Publish]
            , argumentSchema = definitionRef @()
          }
      , validatorDatum = Nothing
      , validatorCompiledCode =
        Just . Short.fromShort $ Week02.serializedMkGiftValidator
    }

mk42ValSmall :: ValidatorBlueprint referencedTypes
mk42ValSmall =
  MkValidatorBlueprint
    { validatorTitle = "42 Validator untyped - small CBOR"
      , validatorDescription = Just "Validator that returns true only
if the redeemer is 42"
      , validatorParameters = []
      , validatorRedeemer =
        MkArgumentBlueprint
          { argumentTitle = Just "Redeemer"
            , argumentDescription = Just "Redeemer for the 42 validator"
            , argumentPurpose = Set.fromList [Spend, Mint, Withdraw,
Publish]
```

```

        , argumentSchema = definitionRef @Integer
    }
, validatorDatum = Nothing
, validatorCompiledCode =
    Just . Short.fromShort $ Week02.serializedMk42ValidatorSmall
}

```

For every validator we can use boilerplate code to define a blueprint. The blueprint defines the validator title, description, parameters that are defined if we have a parameterized validator, redeemer information, datum information, and the compiled code, which we reference from the [Week02.Validators](#) module that we have imported. One can extend the blueprint code such that it generates the data and compiled code for all six validators we have defined in this section. All validators presented in section [Plutus smart contracts](#) can be found at the [plinth-plutusV3](#) branch of the Plutus pioneer program, which also contains a [blueprint.json](#) file with all compiled validator code. Below is an example blueprint for the blueprint code we have defined in this section. The compiled validator code is contained in the "[compiledCode](#)" fields.

```
{
  "$id": "plutus-pioneer-program",
  "$schema": "https://cips.cardano.org/cips/cip57/schemas/plutus-
blueprint.json",
  "$vocabulary": {
    "https://cips.cardano.org/cips/cip57": true,
    "https://json-schema.org/draft/2020-12/vocab/applicator": true,
    "https://json-schema.org/draft/2020-12/vocab/core": true,
    "https://json-schema.org/draft/2020-12/vocab/validation": true
  },
  "preamble": {
    "title": "Plutus Pioneer Program Blueprint",
    "description": "Blueprint for the Plutus Pioneer Program validators",
    "version": "1.0.0",
    "plutusVersion": "v3",
    "license": "MIT"
  },
  "validators": [
    {
      "name": "Mk42ValidatorSmall"
    }
  ]
}
```

```

    "title": "Always True Validator",
    "description": "Validator that always returns True (always
succeeds)",
    "redeemer": {
        "title": "Redeemer",
        "description": "Redeemer for the always true validator",
        "purpose": {
            "oneOf": [
                "spend",
                "mint",
                "withdraw",
                "publish"
            ]
        },
        "schema": {
            "$ref": "#/definitions/Unit"
        }
    },
    "compiledCode": "450101002499",
    "hash": "acec2df7c07075dc7618ffc17c4d86aa786509e646057bd2bdab4cfc"
},
{
    "title": "42 Validator untyped - small CBOR",
    "description": "Validator that returns true only if the redeemer is
42",
    "redeemer": {
        "title": "Redeemer",
        "description": "Redeemer for the 42 validator",
        "purpose": {
            "oneOf": [
                "spend",
                "mint",
                "withdraw",
                "publish"
            ]
        },
        "schema": {
            "$ref": "#/definitions/Integer"
        }
    },
    "compiledCode":
"581e010100255333573466e1d2054375a6ae84d5d11aab9e3754002229308b01",
    "hash": "6828334183fe0deb5576416c73448fdf40cf1c158195b8a24fe9bc45"
}

```

```

],
"definitions": {
  "Integer": {
    "dataType": "integer"
  },
  "Unit": {
    "dataType": "constructor",
    "fields": [],
    "index": 0
  }
}
}

```

We also show the `cabal.project` and `.cabal` files we use to compile the code. Let's look at the `cabal.project` file.

```

repository cardano-haskell-packages
  url: https://chap.intersectmbo.org/
  secure: True
  root-keys:
    3e0cce471cf09815f930210f7827266fd09045445d65923e6d0238a6cd15126f
    443abb7fb497a134c343faf52f0b659bd7999bc06b7f63fa76dc99d631f9bea1
    a86a1f6ce86c449c46666bda44268677abf29b5b2d2eb5ec7af903ec2f117a82
    bcec67e8e99cabfa7764d75ad9b158d72bfacf70ca1d0ec8bc6b4406d1bf8413
    c00aae8461a256275598500ea0e187588c35a5d5d7454fb57eac18d9edb86a56
    d4a35cd3121aa00d18544bb0ac01c3e1691d618f462c46129271bccf39f7e8ee

  index-state:
    -- Bump both the following dates if you need newer packages from
    Hackage
    , hackage.haskell.org 2024-09-10T13:49:28Z
    -- Bump this if you need newer packages from CHaP
    , cardano-haskell-packages 2024-09-10T13:49:28Z

  packages:
    ./

```

In the file, we define the `CHaP repository` that contains all Haskell packages used by Cardano that are not hosted on `Hackage`, the Haskell community's central package archive. When compiling the project, the

`cabal` tool can then download the Plutus packages defined in the `.cabal` file. Next, we look at the `.cabal` file.

```
cabal-version: 3.0
name:          plinth-plutusV3
version:       0.1.0.0
license:
build-type:    Simple
extra-doc-files: README.md

common options
ghc-options: -Wall
default-language: Haskell2010

library scripts
import:         options
hs-source-dirs: src
exposed-modules:
  Week02.Validators

build-depends:
  , base
  , plutus-core      ^>=1.34
  , plutus-ledger-api ^>=1.34
  , plutus-tx        ^>=1.34

if !(impl(ghcjs) || os(ghcjs))
  build-depends: plutus-tx-plugin

executable gen-blueprint
import:         options
hs-source-dirs: app
main-is:        GenBlueprint.hs
build-depends:
  , base
  , bytestring
  , containers
  , plutus-core ^>=1.34.0.0
  , plutus-ledger-api ^>=1.34.0.0
  , plutus-tx ^>=1.34.0.0
  , plutus-tx-plugin ^>=1.34.0.0
  , scripts
```

In `.cabal`, we define our compiler options, the library where our validator code resides, and the build dependencies that list Plutus packages needed by our validator code. At the end, we define the executable project, which tells Cabal which file is the entry point for compiling the project, and we also list the build dependencies. We name the file with the code for generating blueprints `GenBlueprint.hs`. Our project should be structured in the following way:

```
.  
  app  
    GenBlueprint.hs  
  src  
    Week02  
      Validators.hs  
  cabal.project  
  validators.cabal  
  blueprint.json
```

The `blueprint.json` file will only be there after we have compiled the project. We can do this by running the following commands from the top of our project directory:

```
cabal update  
cabal run
```

At the end, we note that the `.cabal` file shown here lists all Plutus libraries needed to compile any Plutus code presented in the [Plutus smart contracts](#) section. It can be reused when compiling code from other sections that we will present next. One only needs to add the module names under the `exposed-modules:` section and update the `GenBlueprint.hs` file with additional blueprint definitions. As already mentioned, all Plinth validator code presented in this chapter, including blueprint and cabal configuration files, can be found at the [plinth-plutusV3](#) branch of the Plutus pioneer program.

25.6.6. Script context explained

In this section, we will examine the `ScriptContext` data type in more detail. All data types contained in the script context will be presented as Haskell data types. Let us recall the implementation of the script context data type.

```
data ScriptContext = ScriptContext
  { scriptContextTxInfo      :: TxInfo
  , scriptContextRedeemer   :: V2.Redeemer
  , scriptContextScriptInfo :: ScriptInfo
  }
  deriving stock (Generic, Haskell.Eq, Haskell.Show)
  deriving anyclass (HasBlueprintDefinition)
```

It contains transaction information, the redeemer, and script information. To reiterate: if a transaction attempts to spend multiple UTXOs at a script address, the spending script is executed separately for each UTXO. The redeemer and script information are assigned individually to each UTXO, whereas the transaction information is shared across all scripts triggered by that transaction.

In the previous code examples, the validation logic did not use the transaction and script information. However, in most validator scripts, they are used. First we look at the script information data type.

Script information

The `ScriptInfo` data type is defined as:

```
data ScriptInfo
  = MintingScript V2.CurrencySymbol
  | SpendingScript V3.TxOutRef (Haskell.Maybe V2.Datum)
  | RewardingScript V2.Credential
  | CertifyingScript
    Haskell.Integer
    TxCert
  | VotingScript Voter
  | ProposingScript
    Haskell.Integer
```

```

ProposalProcedure
deriving stock (Generic, Haskell.Show, Haskell.Eq)
deriving anyclass (HasBlueprintDefinition)
deriving (Pretty) via (PrettyShow ScriptInfo)

```

The script information contains information about the script's purpose. If a transaction tries to mint a native asset, the script info contains the [MintingScript](#) data constructor that carries the [CurrencySymbol](#) data type.

```

newtype CurrencySymbol = CurrencySymbol {unCurrencySymbol :: PlutusTx
.BuiltinByteString}
deriving stock (Generic, Data)
deriving anyclass (NFData, HasBlueprintDefinition)
deriving newtype (Haskell.Eq, Haskell.Ord, Eq, Ord,
                  PlutusTx.ToData, PlutusTx.FromData,
                  PlutusTx.UnsafeFromData)
deriving (Haskell.Show, Pretty)
via LedgerBytes

```

The currency symbol data type is wrapped around a [BuiltinByteString](#) which represents the hash of the minting policy that gets triggered when minting a native asset. We will talk more about native assets and minting in section [Minting policies and native tokens](#).

If a transaction tries to spend a UTXO at a script address, the script info contains the [SpendingScript](#) data constructor that carries a transaction output reference [TxOutRef](#) and a [Maybe V2.Datum](#).

```

data TxOutRef = TxOutRef
{ txOutRefId :: TxId
, txOutRefIdx :: Integer
}
deriving stock (Show, Eq, Ord, Generic)
deriving anyclass (NFData, HasBlueprintDefinition)

```

The transaction output reference contains the transaction ID, which is an SHA-256 hash of the transaction and the transaction index, which is an integer number. Each UTXO that a transaction creates gets a transaction

index assigned starting with 0 and increasing by 1. Because a transaction hash (ID) is unique, together with the output index it uniquely defines a UTXO. The reason transaction hashes are unique is because a new transaction hash includes the hash of the transaction that created the UTXO being consumed. Every transaction consumes at least one UTXO as it needs to pay some fees. By a simple induction argument, it follows that two identical transaction hashes can never exist. We also rely on the fact that the probability of a hash collision is negligible.

The [TxId](#) data type that represents an SHA-256 hash of the transaction is a wrapper around a [BuiltinByteString](#).

```
newtype TxId = TxId {getTxId :: PlutusTx.BuiltinByteString}
    deriving stock (Eq, Ord, Generic)
    deriving anyclass (NFData, HasBlueprintDefinition)
    deriving newtype (PlutusTx.Eq, PlutusTx.Ord, ToData, FromData,
UnsafeFromData)
    deriving (IsString, Show, Pretty)
    via LedgerBytes
```

For PlutusV3 scripts, a datum is no longer necessary to be attached to a UTXO when creating it at a script address, as was the case for PlutusV2 scripts. For this reason, the script info for a spending script contains a maybe datum.

```
newtype Datum = Datum {getDatum :: BuiltinData}
    deriving stock (Generic, Typeable, Haskell.Show)
    deriving newtype (Haskell.Eq, Haskell.Ord, Eq, ToData, FromData,
UnsafeFromData, Pretty)
    deriving anyclass (NFData, HasBlueprintDefinition)
```

The [Datum](#) type is the same as the [Redeemer](#) type, a wrapper around the [BuiltinData](#) type. The datum allows information to be attached to a UTXO that can then be accessed within the validation script logic. We note that the redeemer is defined when we construct a spending transaction that consumes one or more UTXOs at a script address, and the datum is defined

when we construct a producing transaction that creates one or more UTXOs at a script address. We will see examples of how to construct producing and spending transactions in section [Off-chain code with MeshJS](#).

The third option for script purposes is rewarding, which is used to withdraw staking rewards. The rewarding constructor carries the **Credential** data type.

```
data Credential =  
    PubKeyCredential PubKeyHash  
  | ScriptCredential ScriptHash  
  deriving stock (Eq, Ord, Show, Generic, Typeable)  
  deriving anyclass (NFData, HasBlueprintDefinition)
```

This data type either contains a public key hash or a script hash. In the case of a public key hash, the owner of the private key corresponding to the public key needs to sign the transaction to be able to spend the staking rewards. In the case of a script hash the transaction must include or reference the validation script that will be used to decide whether the staking rewards can be withdrawn by the transaction.

Both the **PubKeyHash** and **ScriptHash** data types are wrappers around the **BuiltinByteString** data type.

Other possibilities for the **ScriptInfo** data type cover the following script purposes:

- Certifying - for issuing certificates
- Voting and proposing - used when government actions are involved.

In the [Plutus smart contracts](#) section, we will focus on the spending and minting script purposes.

Decoding the script information in untyped form

Before we move on explaining the transaction information data type we

look at another validator example that shows how to decode the `ScriptInfo` data type in untyped form. Our validator follows the same logic as the 42 validator in the previous section, except that we now match the number 42 to the value of the datum instead of the redeemer.

```
{-
----- -
{- ----- Datum 42 validator untyped
----- -}

{-# INLINEABLE datum42Validator #-}
datum42Validator :: BuiltinData -> BuiltinUnit
datum42Validator ctx
| datumInt == 42 = BI.unitval
| otherwise      = traceError "Datum is a number different than 42"
where
  -- Lazily decode script context up to datum
  constrArgs :: BuiltinData -> BI.BuiltinList BuiltinData
  constrArgs = BI.snd . BI.unsafeDataAsConstr

  scriptInfoBD :: BuiltinData
  scriptInfoBD = BI.head . BI.tail . BI.tail $ constrArgs ctx

  maybeDatumBD :: BuiltinData
  maybeDatumBD = BI.head . BI.tail $ constrArgs scriptInfoBD

  datumBD :: BuiltinData
  datumBD = BI.head $ constrArgs maybeDatumBD

  datumInt :: BI.BuiltinInteger
  datumInt = unsafeDataAsI datumBD

compiledDatum42Validator :: CompiledCode (BuiltinData -> BuiltinUnit)
compiledDatum42Validator = $$compile [||datum42Validator||]

serializedDatum42Validator :: SerialisedScript
serializedDatum42Validator = serialiseCompiledCode
compiledDatum42Validator
```

As we have already explained in the previous section we need to define

the `constrArgs` helper function that helps us to convert the `BuiltinData` to the `BuiltinList` type. The script information is contained on the third place in the script context so we use two time the `tail` function and one time the `head` function to extract it.

Since we are working with a spending script the script info contains the spending constructor followed by the transaction output reference and a maybe datum. We use once the `tail` function and the `head` function to extract it. Because we now have something of type `Maybe V2.Datum` we can not directly convert it to an integer. We again use the `constrArgs` function and then only once the `head` function since there should be only one element in the list which is the datum. After that we have the datum in untyped form, and we can apply the `unsafeDataAsI` function to convert it to an integer number.

In case the datum was not attached to the UTXO or the datum has a different format this validator will fail without a used defined error message. If we do manage to extract an integer the validator will pass if it is 42 or fail with a user-defined message. We note also in this example in case the datum would be a more structured data type we could use the `fromBuiltinData` function that would convert it to typed form where extracting any information can be done in a simpler way.

We again state the various possibilities that a datum can be used for in smart contracts:

- Indicating who can consume a UTXO, when, and under what conditions
- Representing the current state of the UTXO
- Defining metadata and/or configurations.

Transaction information

Let us look at the transaction information `TxInfo` data type.

```
data TxInfo = TxInfo
```

```

{ txInfoInputs          :: [TxInInfo]
, txInfoReferenceInputs :: [TxInInfo]
, txInfoOutputs         :: [V2.TxOut]
, txInfoFee              :: V2.Lovelace
, txInfoMint             :: V3.MintValue
, txInfoTxCerts          :: [TxCert]
, txInfoWdrl              :: Map V2.Credential V2.Lovelace
, txInfoValidRange        :: V2.POSIXTimeRange
, txInfoSignatories       :: [V2.PubKeyHash]
, txInfoRedemers          :: Map ScriptPurpose V2.Redemer
, txInfoData              :: Map V2.DatumHash V2.Datum
, txInfoId                :: V3.TxId
, txInfoVotes              :: Map Voter (Map GovernanceActionId
Vote)
, txInfoProposalProcedures :: [ProposalProcedure]
, txInfoCurrentTreasuryAmount :: Haskell.Maybe V2.Lovelace
, txInfoTreasuryDonation    :: Haskell.Maybe V2.Lovelace
}
deriving stock (Generic, Haskell.Show, Haskell.Eq)
deriving anyclass (HasBlueprintDefinition)

```

It contains several fields that carry information about the transaction being validated. In the beginning, it contains a list of transaction inputs and reference inputs. Reference inputs are inputs accessible by the script context but not consumed by the transaction. They are only referenced, hence the name. A transaction may need access to a UTXO without consuming it because the UTXO can contain important information in the datum that scripts can access. This information can be arbitrary data contained in the datum or an attached reference script, which is a serialized smart contract compiled to Plutus. The advantage of reference scripts is that instead of appending a script to a transaction that wants to spend funds at the script address, we simply reference this script from the UTXO that carries it. This lowers the transaction size and reduces the transaction cost. A code example of this is presented in section [Off-chain code with MeshJS](#). Another advantage of reference inputs is that several transactions in the same block can use the same UTXO as a reference input, since it is not being consumed by any of those transactions.

Transaction inputs and reference inputs are lists of type transaction input info **TxInInfo** that defines the input of a pending transaction.

```
data TxInInfo = TxInInfo
  { txInInfoOutRef    :: V3.TxOutRef
  , txInInfoResolved :: V2.TxOut
  }
  deriving stock (Generic, Haskell.Show, Haskell.Eq)
  deriving anyclass (HasBlueprintDefinition)
```

A transaction input information contains a transaction output reference that we have previously explained. It defines the input UTXO, and it also contains resolved transaction input information in form of the transaction output **TxOut** data type.

```
data TxOut = TxOut
  { txOutAddress      :: Address
  , txOutValue        :: Value
  , txOutDatum        :: OutputDatum
  , txOutReferenceScript :: Maybe ScriptHash
  }
  deriving stock (Show, Eq, Generic)
  deriving anyclass (NFData, HasBlueprintDefinition)
```

A transaction output of a UTXO contains the address at which this UTXO resides, the value it contains, the output datum and possibly a script hash. We look first at the **Address** type.

```
data Address = Address
  { addressCredential     :: Credential
  , addressStakingCredential :: Maybe StakingCredential
  }
  deriving stock (Eq, Ord, Show, Generic, Typeable)
  deriving anyclass (NFData, HasBlueprintDefinition)
```

As explained in section [Cardano addresses](#), an address is composed of two parts: the payment part and the optional staking part. The payment part,

defined with the **Credential** data type, contains either a public key hash or a script hash as we have explained when talking about rewarding.

The optional staking part is defined with the **StakingCredential** data type that can be either a staking hash or a staking pointer.

```
data StakingCredential =
    StakingHash Credential
  | StakingPtr
      Integer -- ^ the slot number
      Integer -- ^ the transaction index (within the block)
      Integer -- ^ the certificate index (within the transaction)
deriving stock (Eq, Ord, Show, Generic, Typeable)
deriving anyclass (NFData, HasBlueprintDefinition)
```

The second information contained in a transaction output is the **Value** type that defines an amount of ada and/or native tokens.

```
newtype Value = Value { getValue :: Map CurrencySymbol (Map TokenName
Integer) }
deriving stock (Generic, Data, Typeable, Haskell.Show)
deriving anyclass (NFData)
deriving newtype (PlutusTx.ToData, PlutusTx.FromData, PlutusTx
.UnsafeFromData)
deriving Pretty via (PrettyShow Value)
```

Every native token is defined with a currency symbol and a token name. They are both wrappers around a **BuiltinByteString**. The currency symbol is computed as the hash of the minting policy, and the token name can be an arbitrary string (but should not be longer than 32 bytes). The **ada** token is defined by an empty byte string both for currency symbol and token name, which means one cannot mint ada.

The third part of a transaction output is the **OutputDatum** data type.

```
data OutputDatum
= NoOutputDatum
| OutputDatumHash DatumHash
```

```

| OutputDatum Datum
deriving stock (Show, Eq, Generic)
deriving anyclass (NFData, HasBlueprintDefinition)

```

A UTXO can either contain no datum, a datum hash or a datum. A datum hash contains a string of type `BuiltinByteString` and the datum type is a wrapper around the `BuiltinData` type same as the `Redeemer` type.

The last data for the `TxOut` data type defines a `Maybe ScriptHash`. A script hash is also a wrapper around a `BuiltinByteString`.

Next in the `TxInfo` data type are the transaction outputs which the transaction aims to create. A transaction output is defined with the `TxOut` data type that we have already shown. After that comes the data for fees and minting. Fees are defined with the `Lovelace` data type that is a wrapper around an `Integer` type.

```

newtype Lovelace = Lovelace { getLovelace :: Integer }
deriving stock (Generic, Typeable)
deriving (Pretty) via (PrettyShow Lovelace)
deriving anyclass (HasBlueprintDefinition)
deriving newtype (Haskell.Eq, Haskell.Ord, Haskell.Show, Haskell.Num,
                  Haskell.Real, Haskell.Enum, PlutusTx.Eq, PlutusTx
.Ord,
                  PlutusTx.ToData, PlutusTx.FromData, PlutusTx
.UnsafeFromData,
                  PlutusTx.AdditiveSemigroup, PlutusTx.AdditiveMonoid,
                  PlutusTx.AdditiveGroup, PlutusTx.Show)

```

Minting is defined with the `MintValue` type that is structured same as the `Value` type.

```

newtype MintValue = UnsafeMintValue (Map CurrencySymbol (Map TokenName
Integer))
deriving stock (Generic, Data, Typeable, Haskell.Show)
deriving anyclass (NFData)
deriving newtype (ToData, FromData, UnsafeFromData)
deriving (Pretty) via (PrettyShow MintValue)

```

After fees and minting comes the data that handles certificates, which is defined by a list of the **TxCert** data type.

```
data TxCert =  
    TxCertRegStaking V2.Credential (Haskell.Maybe V2.Lovelace)  
  | TxCertUnRegStaking V2.Credential (Haskell.Maybe V2.Lovelace)  
  | TxCertDelegStaking V2.Credential Delegatee  
  | TxCertRegDeleg V2.Credential Delegatee V2.Lovelace  
  | TxCertRegDRep DRepCredential V2.Lovelace  
  | TxCertUpdateDRep DRepCredential  
  | TxCertUnRegDRep DRepCredential V2.Lovelace  
  | TxCertPoolRegister  
    -- / poolId  
    V2.PubKeyHash  
    -- / pool VFR  
    V2.PubKeyHash  
  | TxCertPoolRetire V2.PubKeyHash Haskell.Integer  
  | TxCertAuthHotCommittee ColdCommitteeCredential  
HotCommitteeCredential  
  | TxCertResignColdCommittee ColdCommitteeCredential  
deriving stock (Generic, Haskell.Show, Haskell.Eq, Haskell.Ord)  
deriving anyclass (HasBlueprintDefinition)  
deriving (Pretty) via (PrettyShow TxCert)
```

The **TxCert** data type has eleven constructors representing the following certificates:

1. Registering staking credential with an optional deposit amount
2. Unregister staking credential with an optional refund amount
3. Delegate staking credential to a Delegatee
4. Register and delegate staking credential to a Delegatee in one certificate
(deposit is mandatory)
5. Register a DRep with a deposit value
6. Update a DRep
7. Unregister a DRep with mandatory refund value
8. A digest of the PoolParams

9. The retirement certificate and the Epoch in which the retirement will take place
10. Authorize a Hot credential for a specific Committee member's cold credential
11. Resign committee member's cold credential.

After certificates comes the data that manages withdrawals of staking rewards. The data is contained in a **Map** of credentials and lovelace. Next follows the transaction validity range, which is defined with the **POSIXTimeRange** data type.

```
type POSIXTimeRange = Interval POSIXTime

data Interval a = Interval { ivFrom :: LowerBound a, ivTo :: UpperBound a }
    deriving stock (Haskell.Show, Generic)
    deriving anyclass (NFData)
```

The **POSIXTimeRange** data type contains an **Interval** type, parameterized with the **POSIXTime** type. The **Interval** type holds data about the lower and upper bounds of the validity interval for the specified transaction. The lower and upper bound types are structured in the same way. They hold the **Extended** and **Closure** types.

```
data LowerBound a = LowerBound (Extended a) Closure
    deriving stock (Haskell.Show, Generic)
    deriving anyclass (NFData)
```

The **Closure** type is just a wrapper for a Boolean which indicates whether the boundary is included in the interval or not. The extended type has three possible constructor values, which represent negative infinity, positive infinity, or a finite bound parameterized, in our case, by a **POSIXTime** type.

```
newtype POSIXTime = POSIXTime {getPOSIXTime :: Integer}
```

```

deriving stock (Haskell.Eq, Haskell.Ord, Haskell.Show, Generic,
Typeable)
deriving anyclass (NFData, HasBlueprintDefinition)
deriving newtype (AdditiveSemigroup, AdditiveMonoid, AdditiveGroup,
Eq,
Ord, Enum, PlutusTx.ToData, PlutusTx.FromData,
PlutusTx.UnsafeFromData, Haskell.Num, Haskell.Enum,
Haskell.Real, Haskell.Integral)

```

`POSIXTime` is a wrapper for an integer, representing the number of milliseconds that have passed since January 1, 1970, at 00:00.

The module `PlutusLedgerApi.V1.Interval` defines the following helper functions that work with time intervals:

- `member`: checks whether a value is in an interval
- `interval`: takes two parameters as input and constructs an interval with included boundaries
- `from`: takes a value and returns an interval that includes all values greater than or equal to the given value
- `to`: takes a value and returns an interval that includes all values that are smaller than or equal to the given value
- `always`: an interval that covers every possible time
- `never`: an interval that is empty
- `singleton`: takes a value and returns an interval that only contains the single value
- `hull`: takes two intervals as input and returns the smallest interval containing both intervals
- `intersection`: takes two intervals as input and returns the largest interval contained in both of the intervals, if it exists
- `overlap`: checks whether two intervals have a value in common and returns a Boolean

- `contains`: checks whether the second interval is contained in the first one, and returns a Boolean
- `isEmpty`: checks whether an interval is empty and returns a Boolean
- `before`: checks whether a given time is before the given interval and returns a Boolean
- `after`: checks whether a given time is after the given interval and returns a Boolean.

After the validity range in the `TxInfo` data type comes the list of public key hashes that represent transaction signatories. We note again that the `PubKeyHash` data type is a wrapper around the `BuiltinByteString` data type.

Then come the redeemers and datums. As stated before, a transaction can spend multiple UTXOs that get individual redeemers assigned. In case any of these UTXOs have only a datum hash attached, the transaction also needs to contain the datums that belong to those hashes. Data for both redeemers and datums is packaged inside a `Map` object. The keys for the redeemer data are of type `ScriptPurpose`.

```
data ScriptPurpose
  = Minting V2.CurrencySymbol
  | Spending V3.TxOutRef
  | Rewarding V2.Credential
  | Certifying
    Haskell.Integer
    TxCert
  | Voting Voter
  | Proposing
    Haskell.Integer
    ProposalProcedure
deriving stock (Generic, Haskell.Show, Haskell.Eq, Haskell.Ord)
deriving anyclass (HasBlueprintDefinition)
deriving (Pretty) via (PrettyShow ScriptPurpose)
```

The `ScriptPurpose` data type is structured in the same way as the

`ScriptInfo` data type except that the constructor names are different and the *Spending* constructor holds only a transaction output reference without a maybe datum. The keys for the datums are datum hashes that map to actual datums.

Next comes the transaction ID of type `TxId` which represents the hash of the transaction being validated. As already stated this type is a wrapper around the `BuiltinByteString` data type.

After that come the votes and proposal procedures that are used when dealing with government actions. The last two pieces of data contained in the transaction information are the current treasury amount and treasury donation. Both are of type `Maybe V2.Lovelace`. Learn more about governance features, including the topics of voting and proposal submission in chapter [Cardano governance](#).

25.6.7. Time-dependent and parameterized validators

This section demonstrates a smart contract representing a vesting schema. In this scenario, a person sends a gift of ada to the smart contract, and the beneficiary can reclaim this gift after a set deadline has passed. Such a contract can take two approaches, depending on how the validator accesses the beneficiary and deadline information:

- From the datum that is attached to the UTXO we are creating at this script address
- From a parameter that is added as an input variable to the validator script.

First we present the approach when the validator uses the datum of the UTXO.

```
{-# LANGUAGE DataKinds          #-}
{-# LANGUAGE DeriveAnyClass      #-}
{-# LANGUAGE DeriveGeneric       #-}
{-# LANGUAGE DerivingStrategies  #-}
```

```

{-# LANGUAGE FlexibleInstances          #-}
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
{-# LANGUAGE ImportQualifiedPost        #-}
{-# LANGUAGE MultiParamTypeClasses     #-}
{-# LANGUAGE NoImplicitPrelude         #-}
{-# LANGUAGE OverloadedStrings         #-}
{-# LANGUAGE PatternSynonyms          #-}
{-# LANGUAGE ScopedTypeVariables      #-}
{-# LANGUAGE Strict                  #-}
{-# LANGUAGE TemplateHaskell          #-}
{-# LANGUAGE TypeApplications         #-}
{-# LANGUAGE UndecidableInstances      #-}
{-# LANGUAGE ViewPatterns             #-}

{-# OPTIONS_GHC -fno-full-laziness #-}
{-# OPTIONS_GHC -fno-ignore-interface-pragmas #-}
{-# OPTIONS_GHC -fno-omit-interface-pragmas #-}
{-# OPTIONS_GHC -fno-spec-constr #-}
{-# OPTIONS_GHC -fno-specialise #-}
{-# OPTIONS_GHC -fno-strictness #-}
{-# OPTIONS_GHC -fno-unbox-small-strict-fields #-}
{-# OPTIONS_GHC -fno-unbox-strict-fields #-}
{-# OPTIONS_GHC -fno-warn-unused-binds #-}
{-# OPTIONS_GHC -fplugin-opt PlutusTx.Plugin:target-version=1.1.0 #-}

```

```

module Week03.Vesting where

import           GHC.Generics          (Generic)
import           PlutusLedgerApi.Common (FromData
                                       (fromBuiltinData),
                                       SerialisedScript,
                                       serialiseCompiledCode)
import           PlutusLedgerApi.Data.V3 (POSIXTime, PubKeyHash)
import           PlutusLedgerApi.V1.Interval (contains, from)
import           PlutusLedgerApi.V3      (ScriptContext(..),
                                         ScriptInfo(..),
                                         TxInfo(
                                           txInfoValidRange),
                                         getDatum)
import           PlutusLedgerApi.V3.Contexts (txSignedBy)
import           PlutusTx                (BuiltinData,
                                         CompiledCode,
                                         UnsafeFromData
                                         (
                                           unsafeFromBuiltinData),
                                         )

```

```

            compile,
            makeIsDataSchemaIndexed,
            makeLift)
import           PlutusTx.Blueprint          (HasBlueprintDefinition)
import           PlutusTx.Blueprint.Definition (definitionRef)
import           PlutusTx.Bool                (Bool(..), (&&))
import           PlutusTx.Prelude             (BuiltinUnit, Maybe(..),
check,
                                         traceError,
traceIfFalse, ($),
                                         (..))

{-
-----
----- -}
{- ----- TYPES
----- -}

data VestingDatum = VestingDatum
  { beneficiary :: PubKeyHash
  , deadline    :: POSIXTime
  }
deriving stock (Generic)
deriving anyclass (HasBlueprintDefinition)

makeIsDataSchemaIndexed ''VestingDatum [(VestingDatum, 0)]

{-
-----
----- -}
{- ----- VALIDATOR
----- -}

{-# INLINEABLE vestingVal #-}
vestingVal :: ScriptContext -> Bool
vestingVal ctx =
  traceIfFalse "Is not the beneficiary" checkBeneficiary
  && traceIfFalse "Deadline not reached" checkDeadline
where
  checkBeneficiary :: Bool
  checkBeneficiary = txSignedBy info (beneficiary vestingDatum)

  checkDeadline :: Bool
  checkDeadline = from (deadline vestingDatum) `contains`
```

```

txInfoValidRange info

vestingDatum :: VestingDatum
vestingDatum = case scriptContextScriptInfo ctx of
  SpendingScript _txRef (Just datum) ->
    case (fromBuiltinData @VestingDatum . getDatum) datum of
      Just d -> d
      Nothing -> traceError "Expected correctly shaped datum"
    _ -> traceError "Expected SpendingScript with datum"

info :: TxInfo
info = scriptContextTxInfo ctx

{- -----
----- -}
{- ----- HELPERS
----- -}

compiledVestingVal :: CompiledCode (BuiltinData -> BuiltinUnit)
compiledVestingVal = $$compile [||wrappedVal||]
  where
    wrappedVal :: BuiltinData -> BuiltinUnit
    wrappedVal ctx = check $ vestingVal (unsafeFromBuiltinData ctx)

serializedVestingVal :: SerialisedScript
serializedVestingVal = serialiseCompiledCode compiledVestingVal

```

In the beginning we add all necessary language pragmas and import statements. The functions, types and type classes we import are necessary to cover the functionality for both validators presented in this section. We name the module `Week03.Vesting` since the idea for the vesting contract comes from the *Week03* examples from the Plutus pioneer program.

Then we define our `VestingDatum` datum type. It contains the beneficiaries public key hash and the deadline in form of `POSIXTime` after which the funds can be claimed. Because we are defining a custom data type, we need to use the `makeIsDataSchemaIndexed` function that generates the `ToData`, `FromData`, `UnsafeFromData`, and `HasBlueprintSchema` instances for our custom type. As stated in a

previous section, we use template Haskell, that requires to add two single quotes in front of the type to return the type's name.

Next comes the code for the validator. The validation logic says that the funds can be unlocked only when the deadline has been reached and the transaction is signed by the beneficiary. The helper variables `checkBeneficiary` and `checkDeadline`, are of type `Bool`. In the first variable, we use the helper function `txSignedBy` that takes a transaction info and a public key hash and checks whether this transaction has been signed with the correct key that belongs to the hash. In the second variable, we access the transaction validity range and check whether it is contained inside the interval starting with the deadline and going to infinity. As a side note we state when the validity range for a transaction is being defined, that we construct either with off-chain code libraries or with the use of the Cardano CLI, the transaction gets submitted by a node only if the time of submission for this transaction falls into the validity range of the transaction.

In the validator code the datum gets extracted from the script context and converted to typed form. In case the script purpose is incorrect, the datum is not attached or has an incorrect form we raise an error and log a message. At the end we wrap the validator function, compile and serialize it.

In the examples we have seen so far, the validators took in a single parameter, the script context and returned a `Bool` or a `BuiltinUnit`. If there is any variability in the contract, we model that by using the datum as in the vesting example, that contained the beneficiary and the deadline. An alternative approach is to use parameterized contracts, where variability is integrated into the contract by adding a parameter variable to the validator function. The code below shows this, containing a modified version of the previous vesting code example.

```
{-
```

```

----- -}
{----- ----- PARAMETERIZED TYPES
----- -}

data VestingParams = VestingParams
  { beneficiaryParam :: PubKeyHash
  , deadlineParam    :: POSIXTime
  }
  deriving stock (Generic)
  deriving anyclass (HasBlueprintDefinition)

makeLift ''VestingParams
makeIsDataSchemaIndexed ''VestingParams [(VestingParams, 0)]


{-
----- -}
{----- ----- PARAMETERIZED VALIDATOR
----- -}

{-# INLINEABLE paramVestingVal #-}
paramVestingVal :: VestingParams -> ScriptContext -> Bool
paramVestingVal vp ctx =
  traceIfFalse "Is not the beneficiary" checkBeneficiary
  && traceIfFalse "Deadline not reached" checkDeadline
where
  checkBeneficiary :: Bool
  checkBeneficiary = txSignedBy info (beneficiaryParam vp)

  checkDeadline :: Bool
  checkDeadline = from (deadlineParam vp) `contains` txInfoValidRange
info

  info :: TxInfo
  info = scriptContextTxInfo ctx


{-
----- -}
{----- ----- HELPERS
----- -}

compiledParamVestingVal :: CompiledCode (BuiltinData -> BuiltinData ->

```

```

BuiltinUnit)
compiledParamVestingVal = $$compile [||wrappedVal||])
where
  wrappedVal :: BuiltinData -> BuiltinData -> PlutusTx.Prelude
.BuiltinUnit
  wrappedVal params ctx = check $ paramVestingVal (unsafeFromBuiltinData
params)
                                         (unsafeFromBuiltinData
ctx)

serializedParamVestingVal :: SerialisedScript
serializedParamVestingVal = serialiseCompiledCode
compiledParamVestingVal

```

First, we create the `VestingParams` type, which was previously called `VestingDatum`. It holds the beneficiary's public key hash and the deadline after which the beneficiary can claim the funds. Then we again use Template Haskell and generate the necessary instances for our custom type. We also use the `makeLift` function that generates for our custom data type an instance of the `Lift` type class. This line only compiles if the `MultiParamTypeClasses` and `ScopedTypeVariables` language extensions are enabled. An instance of the `Lift` type class is needed in order that a validator containing a parameter can be compiled even though the parameter is not known at the time of compilation.

```

class Lift uni a where
  -- | Get a Plutus IR term corresponding to the given value.
  lift :: a -> RTCompile uni fun (Term TyName Name uni fun ())

```

It contains only the `lift` method, but we will not use that directly. There are a lot of instances for existing Plutus type in this type class such as `BuiltinData`, `BuiltinString`, `BuiltinInteger` and `Bool`. However, this is not used for functions, as they cannot be used to compile Haskell to Plutus validators. The reason it is possible to compile a parameterized validator in Plutus is that the input parameter is not some arbitrary Haskell function, it is static data that you can pass at runtime to a script function if

you create an instance of the [Lift](#) type class.

Next comes the validator code. The type signature of the validator function changes such that it takes the additional vesting parameter. We note that a validator function can take any number of parameters. Our example uses only one parameter. In the body of the validator function, we now read the deadline and the beneficiary's public key hash from the vesting parameter. Apart from that, the logic of the validator stays the same. Then the validator gets compiled and serialized. When compiling the validator, another type parameter needs to be added to the type signatures, and we also need to apply the [unsafeFromBuiltInData](#) function to the vesting parameter in order to wrap the validator before compiling it.

This concludes the procedure of writing a parameterized contract. Next we look at off-chain code that interacts with the validators presented in this section.

25.6.8. Off-chain code with MeshJS

The blockchain is passive – it only acts when a user interacts with it. The code that queries the blockchain, builds, and submits transactions is called off-chain code. Off-chain code does not need to have the same performance and security standards as on-chain code. In this section, we will showcase how to write off-chain code for the on-chain code example presented in the previous section.

After the Alonzo era, when smart contracts became available on Cardano, the [Plutus application platform](#) provided a way to write off-chain code. Developed by IO and implemented as a set of Haskell libraries, it allowed users to write and submit transactions using the Contract monad. A single Haskell file could contain on-chain and off-chain code. Currently, the platform is in maintenance mode and no longer under active development.

Another way for constructing off-chain transactions is by using the [Cardano CLI](#). You can find examples and read more about the Cardano CLI at the [Cardano developers](#) webpage.

There are several community-built tools for writing off-chain code in various programming languages. Some of them are listed at the end of section [Smart contract programming languages](#), where off-chain code is briefly covered. For those using Plinth and interested in writing off-chain code in Haskell, the [Atlas](#) application backend, developed by [GeniusYield](#), provides a solution. It allows code-sharing between the on-chain and off-chain components while enabling the creation of a backend for decentralized applications.

This section explores the [MeshJS](#) tool, developed by the [Mesh team](#). It can be used as a NodeJS package to construct and submit transactions interacting with a smart contract by writing JavaScript or TypeScript code. To run the off-chain code in this and other sections you can use [Deno](#) that is a runtime environment for JavaScript and TypeScript. All off-chain code presented in this and other sections was tested with Deno version [2.1.9](#) and MeshSDK packages version [1.9.0-beta.3](#). To install the MeshSDK packages that are needed by the off-chain code, the [npm](#) package manager for JavaScript runtime environment Node.js can be used. To fix a package version one can create the [package.json](#) file before installing the packages. Below is an example of this file.

```
{
  "dependencies": {
    "@meshsdk/core": "1.9.0-beta.3",
    "@meshsdk/common": "1.9.0-beta.3",
    "@meshsdk/core-cst": "1.9.0-beta.3",
    "deno": "2.1.9"
  }
}
```

The package that installs other MeshSDK packages required by our code is [@meshsdk/core](#). The official [npm page](#) for this package lists the MeshSDK

packages that will get installed as dependencies. To install all MeshJS packages locally, `cd` to the root location of your project, create the `package.json` file and run:

```
npm install @meshsdk/core
```

If no `package.json` file is present `npm` will create one and install the latest version of MeshSDK packages. For more information on how to install npm packages locally or globally and manage the `package.json` file one can read the [Getting packages from the registry](#) and [Creating a package.json file](#) npm documentation pages. The `deno` tool can be installed globally. From the location of the `package.json` file run:

```
npm install -g deno
```

Let us look now at off-chain code that interacts with the vesting smart contract `vestingVal` that we have defined in the previous section. Our code will reside in a TypeScript file to leverage some of the type system's features.

```
import {
    BlockfrostProvider,
    MeshWallet,
    Transaction,
    PlutusScript,
    resolvePlutusScriptAddress,
    applyCborEncoding,
    deserializeAddress,
    resolveSlotNo,
    mConStr0,
    Action
} from "@meshsdk/core";
import { UTx0 } from "@meshsdk/common";
import { secretSeed } from "./seed.ts";

// Define blockchain provider and wallet
const provider: BlockfrostProvider = new BlockfrostProvider(
```

```
"<blockfrost-key>");  
const wallet: MeshWallet = new MeshWallet({  
    networkId: 0, //0=testnet, 1=mainnet  
    fetcher: provider,  
    submitter: provider,  
    key: {  
        type: "mnemonic",  
        words: secretSeed  
    }  
});
```

First, we import the necessary components from the [@meshsdk/core](#) and the [@meshsdk/common](#) libraries. Then we import the seed phrase of our Cardano wallet. The `seed.ts` file should be in the following form:

```
export const secretSeed = [ "seed1", "seed2", ... ];
```

For instructions on how to create a wallet and generate a seed phrase see chapter [Wallets in the world of Cardano](#). Next, we define a provider which helps us to query the blockchain and submit transactions. We use [BlockFrost](#). For the code to work, one has to provide their blockfrost key, which users can get for free at the official webpage. The key is tied to a specific network (mainnet, preprod, preview or SanchoNet). The off-chain code we will present in this and other sections was tested on the preview network. There is a daily limit of how many request a user can make with a free BlockFrost account. Currently, it is set to 50.000. One can also pick another provider. MeshJS can connect to various providers. Examples can be found at the MeshJS [Providers](#) documentation page.

After we have defined our provider, we initiate a Mesh wallet where we input the provider and our secret seed. We also define on which network we will work (0 stands for testnet which is both preview or preprod). Mesh also allows providing the key in other forms. One can define the type of the key as `"root"` and then provide the root key in `bech32` format. Another option is also to define the type of key as `"cli"` and then provide the payment and staking keys. To see all options you can look at the source

code in the official mesh GitHub repository that defines the [MeshWallet](#) class. Next we read out some wallet information and define our vesting script.

```
// Define address and public key hash of it
const walletAddress: string = await wallet.getChangeAddress();
const signerHash: string = deserializeAddress(walletAddress).pubKeyHash;

// Set the vesting deadline
const deadlineDate: Date = new Date("2025-03-05T12:30:10Z")
const deadlinePOSIX: bigint = BigInt(deadlineDate.getTime());

// Defining our vesting script
const vestingScript: PlutusScript = {
    code: applyCborEncoding(
"590ed20101003232323232323232323232323232259..."),
    version: "V3"
};
const vestingAddr: string = resolvePlutusScriptAddress(vestingScript,
0);
```

First, we define the wallet address and the public key hash that belongs to that address. We will set the beneficiary to our own public key hash so we can claim the funds back ourself. Then we define the deadline, which sets the time after which the funds can be claimed from the script address. We set the time in ISO UTC format and then convert it to [POSIX](#) time in milliseconds.

After we have defined our data that we will attach in the datum, we can define our vesting script and compute the address of this script. We note that we have to apply the function [applyCborEncoding](#) that takes the raw script [CBORHEX](#) from the blueprint and formats it in the correct way. The compiled code in our code snippet is shortened. When running this off-chain code, the full compiled code of the validator has to be provided. That can be found in the [blueprint.json](#) file at the [plinth-plutusV3](#) branch of the Plutus pioneer program repository. Now we can send some funds to the vesting script.

```

// Function for creating UTXO at vesting script
async function sendFunds(amount: string): Promise<string> {
  const tx = new Transaction({ initiator: wallet })
    .setNetwork("preview")
    .sendLovelace(
      { address: vestingAddr,
        datum: {value: mConStr0([signerHash, deadlinePOSIX]), inline: true }},
      amount)
    .setChangeAddress(walletAddress);

  const txUnsigned = await tx.build();
  const txSigned = await wallet.signTx(txUnsigned);
  const txHash = await wallet.submitTx(txSigned);
  return txHash
}

```

The `sendFunds` takes an amount of lovelace provided as a string, and creates a transaction that sends the specified amount to the vesting script address. When creating a transaction we use the `Transaction` class where we provide our wallet as initiator. Then we set the network. The Mesh source code states that setting the network is mainly used to know the cost models to be used to calculate script integrity hash. After that we use the `sendLovelace` function, where we specify our script address, the datum that we want to attach, if we want to inline it, and in the end the amount we want to send. We create the datum with the `mConStr0` function that helps us to create a Mesh Data index 0 constructor object. At the end of the transaction we set the change address to our address which means that the lovelace change amount of the UTXOs we will spend will go back to our wallet. Once we have our transaction we build it, sign it and submit it. The `sendFunds` function in the end then returns the transaction hash of the submitted transaction.

Let us look again at the custom vesting data type we use in our validator code:

```
data VestingDatum = VestingDatum
```

```

{ beneficiary :: PubKeyHash
, deadline    :: POSIXTime
}
deriving stock (Generic)
deriving anyclass (HasBlueprintDefinition)

makeIsDataSchemaIndexed ''VestingDatum [(VestingDatum, 0)]

```

We imagine for a moment that we define the above data type as an algebraic data type that also uses a logical **OR** such that it flips the order of the arguments for the second data constructor.

```

data VestingDatumMix = VestingDatum1 { beneficiary1 :: PubKeyHash,
                                         deadline1 :: POSIXTime }
                        | VestingDatum2 { deadline2 :: POSIXTime,
                                         beneficiary2 :: PubKeyHash }
deriving stock (Generic)
deriving anyclass (HasBlueprintDefinition)

makeIsDataSchemaIndexed ''VestingDatumMix [(VestingDatum1, 0),
                                            (VestingDatum2, 1)]

```

When we make the data schema, we then assign the numbers 0 and 1 to the data constructors in the same order we have defined them. Now that we have our datum type, we can modify our validator such that it can work with this type.

```

{-# INLINEABLE vestingValMix #-}
vestingValMix :: ScriptContext -> Bool
vestingValMix ctx =
  traceIfFalse "Is not the beneficiary" checkBeneficiary
  && traceIfFalse "Deadline not reached" checkDeadline
where
  checkBeneficiary :: Bool
  checkBeneficiary = txSignedBy info beneficiaryMix

  checkDeadline :: Bool
  checkDeadline = from deadlineMix `contains` txInfoValidRange info

  vestingDatum :: VestingDatumMix

```

```

vestingDatum = case scriptContextScriptInfo ctx of
  SpendingScript _txRef (Just datum) ->
    case (fromBuiltinData @VestingDatumMix . getDatum) datum of
      Just d -> d
      Nothing -> traceError "Expected correctly shaped datum"
    _ -> traceError "Expected SpendingScript with datum"

variables :: (PubKeyHash, POSIXTime)
variables@(beneficiaryMix, deadlineMix) = case vestingDatum of
  VestingDatum1 b d -> (b, d)
  VestingDatum2 d b -> (b, d)

info :: TxInfo
info = scriptContextTxInfo ctx

```

We now have two options in our off-chain code how to define the datum. In the case of the **VestingDatum1** data constructor, we would attach the datum the same way as we have shown in the **sendFunds** function. For the second case, if we would want to provide the datum as the second option where the arguments are flipped, we would do that the following way:

```
mConStr1([deadlinePOSIX, signerHash])
```

Notice that, beside flipping the arguments, we use now the function **mConStr1** instead of **mConStr0**. With it, we create a Mesh Data index 1 constructor object that corresponds to the second data constructor **VestingDatum2**. If our vesting type would have a third data constructor, we could use the function **mConStr2**. The MeshJS library currently provides these functions up to **mConStr3**.

Next we define the functions needed to claim the vested funds.

```

// Returns a UTXO at a given address that contains the given transaction
hash
async function getUtxo(scriptAddress: string, txHash: string): Promise
<UTx0> {
  const utxos = await provider.fetchAddressUTxOs(scriptAddress);
  if (utxos.length == 0) {

```

```

        throw 'No listing found.';
    }
    let filteredUtxo = utxos.find((utxo: any) => {
        return utxo.input.txHash == txHash;
    })!;
    return filteredUtxo
}

// Function for claiming funds
async function claimFunds(txHashVestedUTXO: string): Promise<string> {
    const assetUtxo: UTx0 = await getUtxo(vestingAddr,
txHashVestedUTXO);
    const redeemer: Pick<Action, "data"> = { data: { alternative: 0,
fields: [] } };
    const slot: string = resolveSlotNo('preview', Date.now() - 40000);

    const tx = new Transaction({ initiator: wallet, fetcher: provider })
        .setNetwork("preview")
        .redeemValue({ value: assetUtxo,
            script: vestingScript,
            redeemer: redeemer})
        .setTimeToStart(slot)
        .sendValue(walletAddress, assetUtxo)
        .setRequiredSigners([walletAddress]);

    const txUnsigned = await tx.build();
    const txSigned = await wallet.signTx(txUnsigned);
    const txHash = await wallet.submitTx(txSigned);
    return txHash
}

// Function calls
//console.log(await sendFunds("5000000"));
//console.log(await claimFunds("<txHash>"));

```

First, we define the `getUtxo` function. It takes in script address and a transaction hash both of type `string`. It then checks if, at the given address, a UTXO exists that contains the provided transaction hash and then returns this UTXO. After that we define our function for claiming funds from the vesting script address.

The `claimFunds` function takes in a transaction hash, which should correspond to the UTXO we want to claim at the vesting address. We then first look up that UTXO. Then we define an empty redeemer. We note that if the redeemer would need to represent a custom data type that has more than one data constructor we would state which data constructor we are using with the number provided in the `"alternative:"` field, and we would input the actual data in the list that follows the `"fields:"` keyword. You can find an additional explanation of how to define a redeemer for a custom data type at the [Using redeemer](#) MeshJS docs section that also showcases a code example.

In the case that the redeemer does not represent a custom data type, but a Plutus supported type as eg an `Integer` we would define the redeemer as:

```
const redeemer: Pick<Action, "data"> = { data: BigInt(42) };
```

The number is of course code specific. You can find off-chain code for the 42 validator example we have presented in a previous section at the [plinth-plutusV3](#) branch.

After we have defined the redeemer, we define the start slot for the transaction. We use the `resolveSlotNo` that takes in the network we work on and a timestamp in milliseconds. We use the current time and subtract 40.000 milliseconds. Then we define our transaction. We first set the network and then use the `redeemValue` function to redeem our vested UTXO. It takes in the UTXO we want to redeem, the script at which the UTXO is residing and the redeemer. The redeemer is optional and could be skipped. The same goes for the datum, which we haven't provided here because we have inlined the datum to the UTXO we are claiming.

Next, we set the validity interval for the transaction with the `setTimeToStart` function. We provide the slot we have previously defined, and the functions sets the validaty interval from this slot onwards to infinity. There is also a `setTimeToExpire` function that sets the time until

which slot the transaction is still considered valid. After that, we define that the lovelace carried by our vested UTXO should go to our own address. In the end, we add the required signers for the transaction and input our own wallet address that is linked with the public key hash we have set in the datum of the vested UTXO.

Once we have created our transaction we again have to build it, sign it and submit it. Then we return the transaction hash. At the end, we define the function calls for the `sendFunds` function that we call with 5 ada, and the `claimFunds` function that we call with the transaction hash the previous function call returns and gets logged to the console. For a user to test this code, one should first uncomment the first function call and run the code. After the transaction is successfully submitted, we comment the first command again, copy the logged transaction hash to the second command, uncomment it and run the code again. If we name our file that contains the off-chain code `vesting.ts` we can run the code with the command below.

```
deno run -A vesting.ts
```

After completing the transactions, the transaction hashes output to the console can be used to check transaction details on the preview.cardanoscan.io webpage.

Now that we have seen the off-chain code for the vesting validator, we can also look at the off-chain code that works with the parameterized vesting validator `paramVestingVal` that we have also defined in the previous section.

```
import {
    BlockfrostProvider,
    MeshWallet,
    Transaction,
    PlutusScript,
    resolvePlutusScriptAddress,
```

```

applyCborEncoding,
deserializeAddress,
resolveSlotNo,
Data,
MeshTxBuilder,
Action
} from "@meshsdk/core";
import { applyParamsToScript } from "@meshsdk/core-cst";
import { UTx0 } from "@meshsdk/common";
import { secretSeed } from "./seed.ts";

// Define blockchain provider and wallet
const provider: BlockfrostProvider = new BlockfrostProvider(
"<blockfrost-key>");
const wallet: MeshWallet = new MeshWallet({
  networkId: 0, //0=testnet, 1=mainnet
  fetcher: provider,
  submitter: provider,
  key: {
    type: "mnemonic",
    words: secretSeed
  }
});

```

We first define the list of imports that in addition contains the **Data** type, **MeshTxBuilder** class and the **applyParamsToScript** function. Then we define our blockchain provider and initialize our wallet. Next, we define the vesting parameters and scripts we will use.

```

// Define address and public key hash of the wallet
const walletAddress: string = await wallet.getChangeAddress();
const beneficiaryPKH: string = deserializeAddress(walletAddress
).pubKeyHash;

// Set the vesting deadline
const deadlineDate: Date = new Date("2025-03-05T12:30:10Z")
const deadlinePOSIX: bigint = BigInt(deadlineDate.getTime());

// Defining the parameter for the script
const scriptParameter: Data = { alternative: 0, fields: [beneficiaryPKH,
deadlinePOSIX] };

```

```

// Defining our vesting script
const vestingParamScript: PlutusScript = {
    code: applyParamsToScript(
        applyCborEncoding(
"590e38010100323232323232323232323232322259..."),
        [scriptParameter]),
    version: "V3"
};

const vestingParamAddr: string = resolvePlutusScriptAddress
(vestingParamScript, 0);

// Defining burn address
const burnScript: PlutusScript = {
    code: applyCborEncoding("450101002601"),
    version: "V3"
};
const burnAddr: string = resolvePlutusScriptAddress(burnScript, 0);

```

As before, we define our wallet address, public key hash and the deadline in POSIX milliseconds time. Then we define the script parameter that we will apply to the parameterized script and that contains our own public key hash and the deadline we have defined. Next we define our parameterized vesting script. We again shorten the **CBORHEX** compiled code. The full compiled code can be found in the **blueprint.json** file that resides at the **plinth-plutusV3** branch. We use now the function **applyParamsToScript** and add the script parameter we have previously defined. If there were more than one parameter, we would enter all of them in the list in the correct order. Then we compute the script address. We also define the burn script, from which no funds can be retrieved. The reason for this is that we will later show in the code how to deploy our vesting script to a UTXO that we create at the burn script. And we will then reference the script from that UTXO when we will claim our vested funds. Next, we can send some funds to the parameterized vesting script.

```

// Function for creating UTXO at vesting script
async function sendFunds(amount: string): Promise<string> {
    const tx = new Transaction({ initiator: wallet })

```

```

    .setNetwork("preview")
    .sendLovelace({ address: vestingParamAddr }, amount)
    .setChangeAddress(walletAddress);

    const txUnsigned = await tx.build();
    const txSigned = await wallet.signTx(txUnsigned);
    const txHash = await wallet.submitTx(txSigned);
    return txHash
}

```

We define the `sendFunds` similarly to the previous vesting example. The difference is that this time we don't specify a datum in the `sendLovelace` function. This means we will create a UTXO without a datum. After that, we can deploy our parameterized vesting script to a UTXO.

```

// Deploy a reference script
async function deployRefScript(lovelaceAmount: string): Promise<string>
{
    const utxos = await wallet.getUtxos();
    const txBuilder = new MeshTxBuilder({
        fetcher: provider
    });

    const unsignedTx = await txBuilder
        .txOut(burnAddr, [{ unit: "lovelace", quantity: lovelaceAmount }])
        .txOutReferenceScript(vestingParamScript.code, vestingParamScript
.version)
        .changeAddress(walletAddress)
        .selectUtxosFrom(utxos)
        .complete();

    const signedTx = await wallet.signTx(unsignedTx);
    const txHash = await wallet.submitTx(signedTx);
    return txHash
}

```

The `deployRefScript` takes in a lovelace amount in form or a string which will be used to deploy the reference script. First we look up the UTXO at our wallet. Then we define a transaction builder that we will use to build the transaction. After that we define our transaction. We first specify that

we want to deploy our vesting script at the burn address and the amount of lovelace we will spend. Then we specify the parameterized script we want to deploy and its Plutus version. After that we define our change address and from which UTXOs we want to select our funds. We complete the transaction, sign and submit it and return the transaction hash. Next, we can define our functions for claiming the vested funds.

```
// Returns a UTXO at a given address that contains the given transaction hash
async function getUtxo(scriptAddress: string, txHash: string): Promise<UTx0> {
    const utxos = await provider.fetchAddressUTxOs(scriptAddress);
    if (utxos.length == 0) {
        throw 'No listing found.';
    }
    let filteredUtxo = utxos.find((utxo: any) => {
        return utxo.input.txHash == txHash;
    });
    return filteredUtxo
}

// Function for claiming funds
async function claimFunds(txHashVestedUTXO: string,
                           txHashRefUTXO: string): Promise<string> {
    const assetUtxo: UTx0 = await getUtxo(vestingParamAddr,
    txHashVestedUTXO);
    const refScriptUtxo: UTx0 = await getUtxo(burnAddr, txHashRefUTXO);
    const redeemer: Pick<Action, "data"> = { data: { alternative: 0,
fields: [] } };
    const slot: string = resolveSlotNo("preview", Date.now() - 40000);

    const tx = new Transaction({ initiator: wallet, fetcher: provider })
        .setNetwork("preview")
        .redeemValue({ value: assetUtxo,
                      script: refScriptUtxo,
                      redeemer: redeemer})
        .setTimeToStart(slot)
        .sendValue(walletAddress, assetUtxo)
        .setRequiredSigners([walletAddress]);

    const txUnsigned = await tx.build();
```

```

    const txSigned = await wallet.signTx(txUnsigned);
    const txHash = await wallet.submitTx(txSigned);
    return txHash
}

// Function calls
//console.log(await sendFunds("3000000"));
//console.log(await deployRefScript("20000000"));
//console.log(await claimFunds("<tx-hash>", "<tx-hash>"));

```

The `getUtxo` is defined the same way as in our previous off-chain code example. The `claimFunds` function is defined in a similar way as in the previous code. It takes in the transaction hash of our UTXO we have vested at the script, and also the transaction hash of the UTXO that contains our deployed vesting script. In the body of the function, we first define our UTXO that we want to claim. Then we define the UTXO that contains our script. After that we again define the redeemer and slot, as in the previous off-chain code.

The transaction to claim the funds is also structured similarly to the previous example. The only difference is that now the `redeemValue` function takes in the reference UTXO that contains the parameterized vesting script instead of the actual script. We again set the validity interval and the required signature to the transaction. After that we build, sign and submit the transaction and return the transaction hash.

At the end, we define the function calls by logging their return values. We send 3 ada to the parameterized vesting script. Then we deploy the script to the burn address and use 20 ada for that. The amount can not be too low and depends on the script size we are deploying. Lastly we claim the funds. There we have to input the transaction hashes that the previous two function calls returned. As in our previous off-chain code example, we have to uncomment only one line at a time and execute the code. If we name our file `vesting-param.ts` we can again execute it with Deno as:

```
deno run -A vesting-param.ts
```

We can again check the transaction details by inputting the hashes at the preview.cardanoscan.io webpage.

We note that the reason it would make sense to deploy a script to an unredeemable UTXO is that referencing a script from a UTXO is cheaper than attaching it to the transaction. Of course, the upfront cost is larger because we need to pay fees for deploying the script. So this use case pays off if we need to reference this script many times, which could be the case for a DApp. Also, if the UTXO is created at a script address where funds are burned, it becomes unspendable and permanently accessible.

We make an important distinction between off-chain and on-chain code. Because Cardano uses the hard fork combinator technology, all Plutus script versions are supported by the blockchain (read more in section [Cardano node layers](#)). Once you have written on-chain code that works, it will work indefinitely. That is not the case for off-chain code.

Because the ledger rules can be updated, it can happen that conditions which a transaction needs to fulfill may change. An example is the required ada fee when attaching a reference script to a transaction. Some off-chain code libraries may calculate this fee automatically. This calculation then holds true for a specific Cardano era and may not work in future eras if blockchain parameters change. For this reason, it is good practice to upgrade the off-chain code library versions your code is using and test the code when upgrades to the Cardano ledger happen.

More information about MeshJS is available at:

- [MeshJS guides](#)
- [MeshJS documentation](#)
- [MeshJS examples repository](#)

- [Mesh project based learning page](#)

In the next section we show how to write a minting policy and off-chain code that interacts with the policy.

25.6.9. Minting policies and native tokens

In Plutus, a minting policy or minting script defines the conditions under which native tokens can be minted. Each UTXO has an address, a value, and potentially a datum. In previous examples, the value was always ada. To include native tokens in a UTXO, they must be explicitly created. The [IO blog on native tokens](#) states that many blockchains, creating custom tokens traditionally requires writing and deploying smart contracts – code that defines how the token behaves and how it can be transferred, created (minted), or destroyed (burned). This approach makes these user-defined tokens non-native, meaning the underlying blockchain's fundamental structure does not directly support them which can lead to inefficiencies, higher costs, and increased complexity. In Cardano all tokens are native, which means developers can create and manage their own tokens without relying on complex smart contracts. These tokens are treated as first-class citizens by the ledger, enabling secure and efficient handling of many token types directly at the protocol level. The blog further states that the key advantages of native tokens in Cardano over traditional smart contract-based approaches are:

- Efficiency and lower costs. Because native tokens are supported at the ledger level, their creation, transfer, and management require no custom contract logic. This reduces transaction size, increases throughput, and significantly lowers fees. Operations are faster and consume fewer resources compared to tokens built with smart contracts.
- Security and simplicity. Native tokens inherit the same security properties as ada. There is no additional contract logic to audit or maintain, reducing the risk of vulnerabilities and exploits. This makes

native tokens simpler and safer to use in DApps.

- Developer experience and compatibility. Creating and managing native tokens is more straightforward than writing and deploying custom contracts. Native tokens integrate seamlessly with wallets, tools, and the wider ecosystem.

One can read more about the security advantages of native tokens on Cardano in the [Cardano security](#) section. Before we look at an example of a minting policy let us look again at the Haskell definition of the **Value** type. This type is defined in the module `PlutusLedgerApi.V1.Value` and defines an amount of ada and/or native tokens.

```
newtype Value = Value { getValue :: Map CurrencySymbol (Map TokenName Integer) }
    deriving stock (Generic, Data, Typeable, Haskell.Show)
    deriving anyclass (NFData)
    deriving newtype (PlutusTx.ToData, PlutusTx.FromData, PlutusTx.UnsafeFromData)
    deriving Pretty via (PrettyShow Value)
```

The **Value** type is a map object that connects a currency symbol to another map, which uses token names as its keys. Both token names and currency symbols are wrappers for the type **BuiltinByteString**, representing a byte string. As said before, the currency symbol represents the hash of the minting policy and the **ada** token is defined by an empty byte string both for currency symbol and token name, which means one cannot mint ada. These two byte strings define a native token or coin, while the integer represents the token's amount. Additionally, there is another type called **AssetClass**.

```
newtype AssetClass = AssetClass
    { unAssetClass :: (CurrencySymbol, TokenName)
    }
    deriving stock (Generic, Data, Typeable)
    deriving newtype
        ( Haskell.Eq
```

```

    , Haskell.Ord
    , Haskell.Show
    , Eq
    , Ord
    , PlutusTx.ToData
    , PlutusTx.FromData
    , PlutusTx.UnsafeFromData
)
deriving anyclass (NFData, HasBlueprintDefinition)
deriving (Pretty) via (PrettyShow (CurrencySymbol, TokenName))

```

It combines the currency symbol and token name to define an asset class, which can represent a native token or ada. Because the value type is a map, it can contain different tokens and amounts, including ada. To construct a value, we can use the function `assetClass`, which takes a currency symbol and token name, and returns a variable of type asset class. We can then use the function `assetClassValue`, which takes an asset class and integer, and returns a variable of type value. With the `assetClassValueOf` function, we can check how many tokens of a specific asset class are contained in a value type variable. Below, you can see the type signatures of these functions.

```

assetClass :: CurrencySymbol -> TokenName -> AssetClass
assetClassValue :: AssetClass -> Integer -> Value
assetClassValueOf :: Value -> AssetClass -> Integer

```

To construct a value variable and check the amount of tokens in it for a specific asset class, we can do as follows from Prelude that we have build with the required Plutus libraries.

```

Prelude> import PlutusLedgerApi.V1.Value
Prelude PlutusLedgerApi.V1.Value> :set -XOverloadedStrings
Prelude PlutusLedgerApi.V1.Value> myAssetClass = assetClass "a507ff33"
"MyToken"
Prelude PlutusLedgerApi.V1.Value> myTokenValue = assetClassValue
myAssetClass 77

```

To construct an asset class for ada, we can use the `adaSymbol` and `adaToken` variables provided by the `PlutusLedgerApi.V1.Value` module.

```
Prelude PlutusLedgerApi.V1.Value> ada = assetClass adaSymbol adaToken
Prelude PlutusLedgerApi.V1.Value> adaValue = assetClassValue ada
100000000
```

We can also combine different variables of type value with the semigroup operator `<>` as the value type has an instance of the semigroup type class. Using the `assetClassValueOf` function, we can get the quantity of a given asset class contained in a value.

```
Prelude PlutusLedgerApi.V1.Value> combined = myTokenValue <> adaValue
Value (Map [(,Map [("",100000000)]),(a507ff33,Map [("MyToken",77)])])
Prelude PlutusLedgerApi.V1.Value> assetClassValueOf combined
myAssetClass
77
```

We can also extract information from a value type variable as a list of triples using the `flattenValue` function, which takes a value type and returns a list of triples.

```
Prelude PlutusLedgerApi.V1.Value> :t flattenValue
flattenValue :: Value -> [(CurrencySymbol, TokenName, Integer)]
Prelude PlutusLedgerApi.V1.Value> flattenValue combined
{empty}[(, "",100000000),(a507ff33,"MyToken",77)]
```

So far, we have focused on the spending constructor of the `ScriptInfo` type variable that defines the purpose of the currently-executing script. The `TxInfo` data type contains the `MintValue` type that is structured the same way as the `Value` type. Minting policies are triggered if this field contains a non-zero value. Each currency symbol defined in this variable activates the corresponding minting policy, linking them through the policy's hash.

A minting policy, like a spending validation script, only requires one input: the script context. If for the currently-executing script the `ScriptInfo`

data type defines the minting constructor that is parameterized with a currency symbol, then the corresponding minting policy gets triggered. The minting constructor does not contain a maybe datum like the spending constructor does, because datums can exist at UTXOs sitting at script addresses, and minting scripts do not consume these UTXOs; they only produce new ones.

A single transaction may trigger several different minting policies if multiple native tokens with distinct currency symbols are minted. Each policy receives its own script info and redeemer as input, and they share the same transaction information data type. All minting policies within a transaction must pass for the transaction to succeed; otherwise it fails.

Now, let us examine an example of a minting policy where only the owner of a specific public key, who signs the transaction, is permitted to mint or burn tokens. This key could represent a project or company acting like a central bank in traditional finance, responsible for minting and burning fiat currencies. Our minting policy will be parameterized, accepting an additional public key hash as a parameter.

```
{-# LANGUAGE DataKinds          #-}  
{-# LANGUAGE DeriveAnyClass     #-}  
{-# LANGUAGE DeriveGeneric      #-}  
{-# LANGUAGE DerivingStrategies #-}  
{-# LANGUAGE FlexibleInstances   #-}  
{-# LANGUAGE GeneralizedNewtypeDeriving #-}  
{-# LANGUAGE ImportQualifiedPost #-}  
{-# LANGUAGE MultiParamTypeClasses #-}  
{-# LANGUAGE NoImplicitPrelude    #-}  
{-# LANGUAGE OverloadedStrings    #-}  
{-# LANGUAGE PatternSynonyms     #-}  
{-# LANGUAGE ScopedTypeVariables  #-}  
{-# LANGUAGE Strict              #-}  
{-# LANGUAGE TemplateHaskell     #-}  
{-# LANGUAGE TypeApplications     #-}  
{-# LANGUAGE UndecidableInstances #-}  
{-# LANGUAGE ViewPatterns        #-}  
{-# OPTIONS_GHC -fno-full-laziness #-}
```

```

{-# OPTIONS_GHC -fno-ignore-interface-pragmas #-}
{-# OPTIONS_GHC -fno-omit-interface-pragmas #-}
{-# OPTIONS_GHC -fno-spec-constr #-}
{-# OPTIONS_GHC -fno-specialise #-}
{-# OPTIONS_GHC -fno-strictness #-}
{-# OPTIONS_GHC -fno-unbox-small-strict-fields #-}
{-# OPTIONS_GHC -fno-unbox-strict-fields #-}
{-# OPTIONS_GHC -fplugin-opt PlutusTx.Plugin:target-version=1.1.0 #-}

module Week05.Minting where

import           PlutusLedgerApi.Common      (SerialisedScript,
                                               serialiseCompiledCode)
import           PlutusLedgerApi.V1.Value    (flattenValue)
import           PlutusLedgerApi.V3          (ScriptContext(..),
                                              TokenName,
                                              TxInInfo(txInInfoOutRef),
                                              TxInfo(txInfoInputs,
                                              txInfoMint),
                                              TxOutRef(TxOutRef), TxId
                                              (TxId),
                                              PubKeyHash)
import           PlutusTx                      (BuiltInData, CompiledCode,
                                              UnsafeFromData,
                                              unsafeFromBuiltInData,
                                              compile)
import           PlutusLedgerApi.V3.Contexts (txSignedBy)
import           PlutusTx.Bool                (Bool(..), (&&))
import           PlutusTx.Prelude             (BuiltinUnit, Eq((==)),
                                              any, check,
                                              traceIfFalse, ($))

{-
----- -
----- -}
{- ----- VALIDATOR
----- -}

{-# INLINABLE signedVal #-}
signedVal :: PubKeyHash -> ScriptContext -> Bool
signedVal pkh ctx = traceIfFalse "missing signature" $
                     txSignedBy (scriptContextTxInfo ctx)
pkh

```

```

{- -----
----- -}
{- ----- HELPERS
----- -}

compiledSignedVal :: CompiledCode (BuiltinData -> BuiltinData ->
BuiltinUnit)
compiledSignedVal = $$compile [||wrappedVal||]
where
  wrappedVal :: BuiltinData -> BuiltinData -> BuiltinUnit
  wrappedVal pkh ctx = check $ signedVal
    (unsafeFromBuiltinData pkh)
    (unsafeFromBuiltinData ctx)

serializedSignedVal :: SerialisedScript
serializedSignedVal = serialiseCompiledCode compiledSignedVal

```

First, we add all the necessary language pragmas. We then name the module `Week05.Minting` since the idea for the code examples is taken from the 5th week of the 4th Plutus pioneer program. After that, we import the modules we need. Next we define our minting policy. It is parameterized with a public key hash and the validation checkes if the correct signature is present. We do not check if the currently executing script has a minting purpose specified. In that sense, this validation logic could also be used as a spending script. Then we compile the minting policy as in the examples before, and serialize it. Now that we have our minting policy, we can look at the off-chain code that interacts with it.

```

import {
  BlockfrostProvider,
  MeshWallet,
  Transaction,
  PlutusScript,
  applyCborEncoding,
  deserializeAddress,
  Mint,
  Action
} from "@meshsdk/core";

```

```

import { applyParamsToScript } from "@meshsdk/core-cst";
import { secretSeed } from "./seed.ts";

// Define blockchain provider and wallet
const provider: BlockfrostProvider = new BlockfrostProvider(
"<blockfrost-key>");
const wallet: MeshWallet = new MeshWallet({
    networkId: 0, //0=testnet, 1=mainnet
    fetcher: provider,
    submitter: provider,
    key: {
        type: "mnemonic",
        words: secretSeed
    }
});

// Define address and public key hash of it
const walletAddress: string = await wallet.getChangeAddress();
const signerHash: string = deserializeAddress(walletAddress).pubKeyHash;

// Defining our minting policy
const mintingPolicy: PlutusScript = {
    code: applyParamsToScript(
        applyCborEncoding(
"590b49010100323232323232323232323232322259..."),
        [signerHash]),
    version: "V3"
};

```

First we import all necessary MeshJS classes and functions, then we define our blockchain provider and wallet, same as before. Next we read out our wallet address and the public key hash. And then we define our minting policy, where we apply the public key hash as a parameter. Then follows the code for minting tokens.

```

// Defining our token
const token: Mint = {
    assetName: 'MyTokens',
    assetQuantity: '2',
    recipient: { address: walletAddress }
}

```

```

// Minting our tokens
async function mintTokens(): Promise<string> {
    const redeemer: Pick<Action, "data"> = { data: { alternative: 0,
fields: [] } }

    const tx = new Transaction({ initiator: wallet, fetcher: provider })
        .setNetwork("preview")
        .mintAsset(mintingPolicy, token, redeemer)
        .setRequiredSigners([walletAddress]);

    const txUnsigned = await tx.build();
    const txSigned = await wallet.signTx(txUnsigned);
    const txHash = await wallet.submitTx(txSigned);
    return txHash
}

// Function calls
console.log(await mintTokens());

```

We first define the tokens we want to mint where we specify the token name, amount of tokens, and the address where the tokens should be created. In Plutus, a token name can be an arbitrary byte string which is limited to 32 bytes. If a human-readable name is set, wallets also display it in that form. The number of tokens has to be a positive integer. After setting the tokens we want to mint, we define our minting function. In it, we first define an empty redeemer. Next, we create our transaction. In the transaction we set the network, say which tokens we want to mint with which minting policy, add the redeemer, and sign the transaction with our wallet key. At the end we build, sign and submit the transaction and return the transaction hash. Now we can make our function call and mint the tokens. If we name our file `signed-minting.ts`, we can mint our tokens as:

```
deno run -A signed-minting.ts
```

We can then check again the transaction details on the

preview.cardanoscans.io webpage. This was an example of minting fungible tokens. Fungibility means that one token is interchangeable with another token of the same type, meaning each unit has the same value as any other unit.

Plutus also allows the minting of non-fungible tokens (NFTs), which are unique tokens that can only be minted once. The key to writing such a minting policy is referencing something unique on the Cardano blockchain, and for that, we use UTXOs. UTXOs can exist only once, and once consumed as input to a transaction, they can never exist again. UTXOs are defined by transaction hashes, which are unique, and by output indices. The reason every transaction has a unique hash was explained in section [Script context explained](#).

When minting an NFT, the idea is to include a specific parameter in the minting policy – namely, a UTXO transaction hash and ID – and have the policy check that the transaction performing the minting consumes that specific UTXO. Let us look at an example of such minting policy.

```
{-
-----
----- -}
{- ----- NFT VALIDATOR
----- -}

{-# INLINEABLE nftVal #-}
nftVal :: TxOutRef -> TokenName -> ScriptContext -> Bool
nftVal oref tn ctx =
    traceIfFalse "UTx0 not consumed" checkHasUTx0 &&
    traceIfFalse "You can only mint one!" checkMintedAmount
where
    checkHasUTx0 :: Bool
    checkHasUTx0 = any (\i -> txInInfoOutRef i == oref) $ txInfoInputs
info

    checkMintedAmount :: Bool
    checkMintedAmount = case flattenValue (txInfoMint info) of
        [(_ , tn', amt)] -> tn' == tn && amt == 1
```

```

    -> False

info :: TxInfo
info = scriptContextTxInfo ctx

{-
-----
----- -}
{-
----- HELPERS
----- -}

compiledNftVal :: CompiledCode (BuiltinData -> BuiltinData ->
BuiltinData ->
                    BuiltinData -> BuiltinUnit)
compiledNftVal = $$compile [||wrappedVal||]
where
  wrappedVal :: BuiltinData -> BuiltinData -> BuiltinData ->
                    BuiltinData -> BuiltinUnit
  wrappedVal tid idx tn ctx =
    let oref :: TxOutRef
      oref = TxOutRef
        (TxId $ unsafeFromBuiltinData tid)
        (unsafeFromBuiltinData idx)
    in check $ nftVal
      oref
      (unsafeFromBuiltinData tn)
      (unsafeFromBuiltinData ctx)

  serializedNFTVal :: SerialisedScript
  serializedNFTVal = serialiseCompiledCode compiledNftVal

```

The language pragmas and import of modules from the previous minting policy example also cover the functionality needed in this minting policy. Our script will be parameterized by two inputs: the transaction output reference - **TxOutRef** - of the UTXO we are spending and the token name. The condition we check is that the minting transaction consumes the specified transaction reference passed to the script. The **any** function is used to check this. It takes a Boolean-returning function and applies it to a list, returning true if at least one element satisfies the condition. We also ensure that only one token with the specified name is minted by using the

`flattenValue` function, which converts a value type into a list of triples. These checks ensure only one coin is minted with the currency symbol tied to a specific parameterized script.

After we have defined our minting policy, we compile it. The transaction output reference is made up of two types, which are the transaction hash and output index. Before we compile our script in untyped form, we can change it such that it takes in the transaction hash and output index as two separate parameters. We do that in the `wrappedVal` helper function. It takes in four parameters that are all of the type `BuiltinData` and converts them to their custom data types before applying them to our typed minting policy. After we compile our untyped minting policy, we serialize it. In principle, we could also leave our policy such that it takes in the transaction output reference as a whole and then model that in our off-chain code. In our code, we wanted to show how to split up such a variable and correctly apply the `unsafeFromBuiltinData` to individual parts. Next, we show the off-chain code that interacts with our NFT minting policy.

```
import {
    BlockfrostProvider,
    MeshWallet,
    Transaction,
    PlutusScript,
    applyCborEncoding,
    UTxO,
    Action,
    Mint
} from "@meshsdk/core";
import { applyParamsToScript } from "@meshsdk/core-cst";
import { secretSeed } from "./seed.ts";

// Define blockchain provider and wallet
const provider: BlockfrostProvider = new BlockfrostProvider(
    "<blockfrost-key>");
const wallet: MeshWallet = new MeshWallet({
    networkId: 0, //0=testnet, 1=mainnet
    fetcher: provider,
```

```

    submitter: provider,
    key: {
      type: "mnemonic",
      words: secretSeed
    }
  });

// Define address and public key hash of it
const walletAddress: string = await wallet.getChangeAddress();

// Defining the UTXO we want to spend
const utxos: UTxO[] = await wallet.getUtxos();
const utxo: UTxO = utxos[0];

// Defining our NFT policy
const nftPolicy: PlutusScript = {
  code: applyParamsToScript(
    applyCborEncoding(
      "590bf5010100323232323232323232323232323232222259..."),
      [utxo.input.txHash, BigInt(utxo.input.outputIndex), "My NFT"
    ]),
  version: "V3"
};

```

First, we again import all necessary MeshJS classes and functions and define our provider and wallet. After that we set our wallet address and read out the first UTXO sitting at our wallet address. We will use this UTXO to parameterize our minting policy and also spend it when creating the minting transaction. Next we define our NFT minting policy. We again shorted the compiled code and parameterize the code with the transaction hash and output index of the UTXO we selected. We also provide the name of our NFT we want to mint. Now we can look at the function that mints the NFT.

```

// Defining our NFT token
const nftToken: Mint = {
  assetName: 'My NFT',
  assetQuantity: '1',
  recipient: { address: walletAddress }
}

```

```

// Minting our NFT
async function mintNFT(): Promise<string> {
    const redeemer: Pick<Action, "data"> = { data: { alternative: 0,
fields: [] } }

    const tx = new Transaction({ initiator: wallet, fetcher: provider })
        .setNetwork("preview")
        .setTxInputs([utxo])
        .mintAsset(nftPolicy, nftToken, redeemer);

    const txUnsigned = await tx.build();
    const txSigned = await wallet.signTx(txUnsigned);
    const txHash = await wallet.submitTx(txSigned);
    return txHash
}

// Function calls
console.log(await mintNFT());

```

We define our token that we want to mint and provide the token name, token quantity (which should be one), and set the recipient address to our own address. Then we define our minting function. We set an empty redeemer and create the transaction, similar to the previous minting policy. The difference is that in this example we specify the UTXO we want to spend, and we do not need to add the required signature linked to our wallet as we did in the previous minting example. At the end we build, sign and submit the transaction and return the transaction hash. When we call the minting function, we again log the hash such that it can be used at the preview.cardanoscan.io webpage to inspect our transaction that minted the NFT.

If we run this code multiple times, we would create tokens with the same name but different currency symbols because we would parameterize the minting script each time with a different UTXO. This difference in currency symbols ensures that each token is truly an NFT, as the asset class is defined by both the token name and its unique currency symbol.

It is also possible to associate an image with a token or NFT. It can be referenced from the [InterPlanetary file system](#) or embedded as a [base64](#) encoded string into the datum metadata of the token or NFT. For more information, see [CIP-68](#) that defines the datum metadata standard. The [MeshJS docs](#) provide an example of how to mint assets with the CIP-68 metadata standard.

At the end of this section, we note that the utility of a token can be determined by its use case, the market, or the community that issues and adopts it. Tokens can be used for various purposes such as issuing a new cryptocurrency, representing access rights, or enabling in-app transactions. Native tokens enable a wide range of applications by combining secure on-chain logic with the flexibility of custom asset creation. From the [IO blog on native tokens](#) we get the following use cases for native tokens:

- Digital collectibles and credentials. NFTs can represent ownership of digital art, in-game assets, event tickets, or even academic credentials – benefiting from the security and transparency of the Cardano ledger.
- Supply chain and asset tracking. Tokens can represent physical items and track their journey across a supply chain. Because native tokens can be uniquely defined and managed without smart contracts, this approach is more efficient and secure.
- Algorithmic stablecoins. Native tokens can represent assets with values tied to external references. Forging policy scripts define the rules that keep these tokens stable, enabling automated monetary policies and more robust decentralized finance (DeFi) primitives.
- Tokenized roles and permissions. Roles within a decentralized system – such as operator, validator, or participant – can also be represented by unique native tokens. Holding a token grants permission to perform specific actions, allowing for modular and transferable system designs.
- Access control and licensing. A token could grant access to a service,

event, or dataset. For example, holding a specific token could unlock premium features in a DApp or grant entry to a token-gated community.

- Decentralized governance. Native tokens can represent voting rights or stake in decision-making processes. This enables on-chain governance systems with transparent and auditable participation.

To learn more about native tokens, see:

- [Native tokens](#) overview on Cardano Docs
- [Discover native tokens](#) section on the Developer Portal
- [Ledger explanations](#) page, which also covers native tokens
- [Native Custom Tokens in the Extended UTXO Model](#) scientific paper.

25.6.10. PlutusV3 features

PlutusV3 focuses on performance throughput, smart contract size, platform capabilities, and interoperability with other blockchains. Specifically, PlutusV3 brings:

- Bitwise primitives
- Cryptographic primitives
- Sums of products.

Bitwise primitives allow for the manipulation of data at the lowest level of bits. Those operations are the fundamental building blocks of many algorithms and data structures. There are numerous applications, ranging from efficient representation and manipulation of sets of integers to the implementation of cryptographic primitives and fast searches. In particular, PlutusV3 enables converting integers to byte strings, which allows, for example, to hash arbitrary data, which hasn't been possible in the past.

Cryptographic primitives form the foundation of blockchain technology.

The choice of primitives significantly influences blockchain's capabilities. Because other blockchains choose different primitives from Cardano, it is challenging to interoperate with them. An example of interoperability is checking Ethereum signatures in Plutus, which is possible with PlutusV3. There are three groups of new cryptographic primitives in PlutusV3:

- BLS12-381 is about elliptic curve pairing and including 17 primitives that support cryptographic curves. Use cases include sidechains, ZKP, Hydra, Mithril and ATALA
- Blake2b_224 is a cryptographic hash function that enables hashing of public keys and scripts within Plutus, facilitating various interesting applications
- Keccak-256 is another cryptographic hash function that supports Ethereum signature verification within Plutus scripts.

These cryptographic primitives bring new possibilities, unlock better ways to achieve scalability, and allow for better interoperability with Ethereum and EVM-based blockchains. The [PlutusV3 Overview](#) video demonstrates how to lock ada in a Plutus validator that can only be unlocked by the owner of an ETH wallet. The demonstration starts at 4:10. You can also read more about PlutusV3 and the features that it brings in the IO blog [Unlocking more opportunities with PlutusV3](#). The official documentation provides further information on the [Chang hard fork](#) that enables PlutusV3.

The last feature PlutusV3 brings is sums of products. One common method for encoding data types in Plutus Core is the Scott approach. With the introduction of PlutusV3, sums of products allow for encoding data types more compactly and cost-effectively than Scott encoding, so the latter is no longer required. The sums of products method is designed to enhance script efficiency and optimize code generation for Plutus Core compilers. This involves implementing new term constructors for packing fields into constructor values and enabling efficient tag inspection for case branches.

As a result, programs may experience a performance increase of up to 30%, marking a significant optimization that streamlines operations and ensures faster execution of smart contracts. For further details, refer to [CIP-85](#). Sums of products do not change built-in functions available in Plutus, but change the Plutus language itself. Also Plutus now provides a way to deal directly with the [Data](#) type for all Plutus script versions (V1, V2 and V3) that allows for better code optimization. This is discussed further in section [Simple validation scripts](#). The [Plinth user guide](#) also provides information on that.

25.7. Smart contract security

25.7.1. Cardano security

The security of smart contracts is an important topic, since these programs can manage significant amounts of funds. Coindesk's estimate of the loss of funds due to various attacks and hacks, including scams, was over \$3.2bn in 2021 and over \$3.7bn in 2022 (source: [CoinDesk](#)). If we look at hacks to blockchain smart contracts and bridges only, up to October 2023 they have totaled \$4.28bn (source: [Emurgo](#)). The graph below displays the cumulative amount of lost funds in audited and unaudited codebases for Web3 projects in a three-year period (data taken from the [Rekt leaderboard](#)). Out of 137 compromised projects, 58 were audited, and 79 were not, meaning that unaudited Web3 projects account for a substantially larger cumulative loss of funds, even though their number is not significantly higher than the number of audited projects.

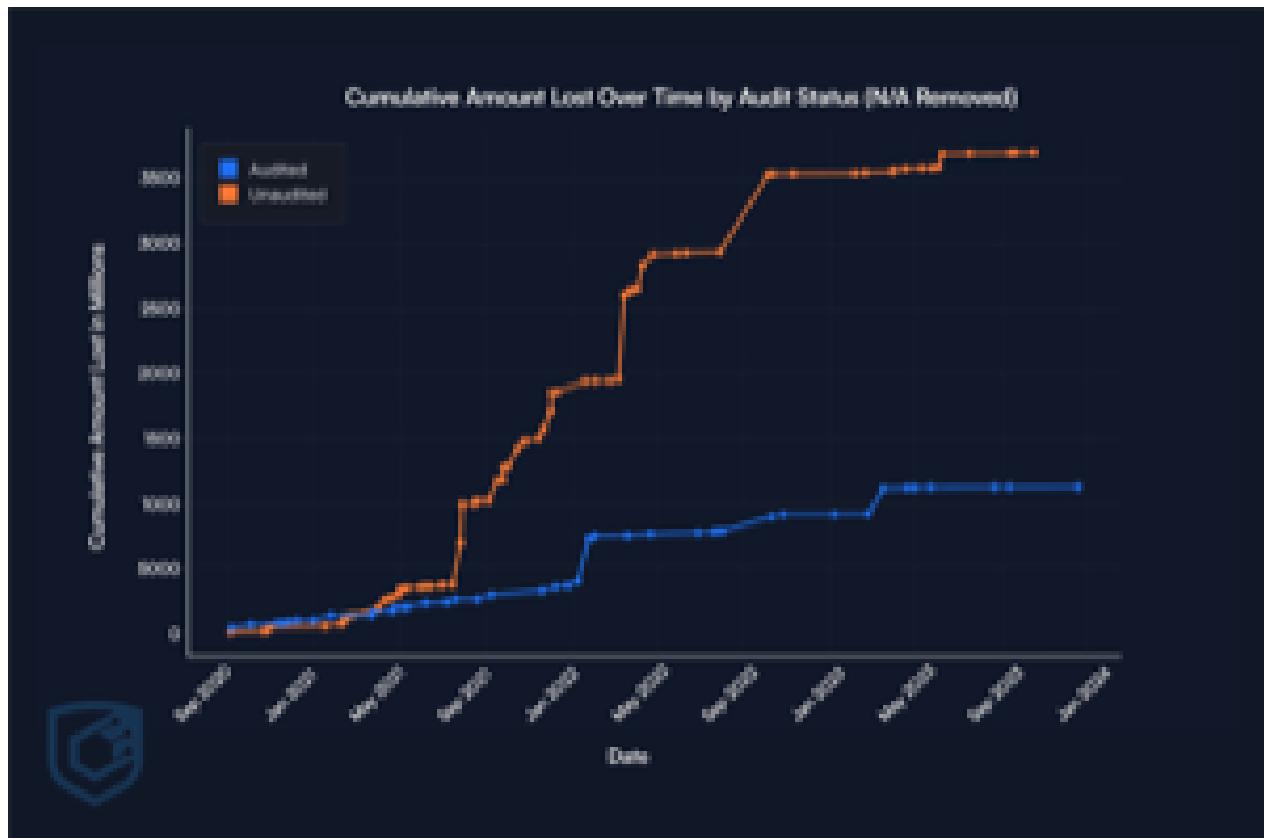


Figure 47. Loss of funds, source: Cyfrin blog

Nevertheless, a \$1.1bn loss in audited projects is still high. This implies that an audit by itself may not always ensure a project's safety. Other factors, such as the auditing methodology and the security features of the underlying blockchain also influence the safety of a project.

A software audit is a type of software review where one or more auditors (usually third parties) conduct an examination of a software product to assess compliance with specifications, standards, contractual agreements, and other criteria (source: [Wikipedia](#)). Blockchain auditing can be defined as the systematic process of reviewing and analyzing a blockchain's codebase and its specifications. It involves code revision to ensure no hidden or unwanted design flaws or bugs that may lead to production issues. Beside security guarantees, an audit can also prove compliance with industry standards and best practices and increase trust and credibility. Audits can be grouped into manual auditing, where each line of code is analyzed by the auditors, and automated auditing, where auditors use software auditing tools. A combination of both is also possible. Auditing of smart contracts is further discussed in the [Plutus](#)

[security](#) section.

Cardano was designed with security in mind from the beginning. Input | Output (IO) has invested much effort into academic research, and its research team has written and collaborated on more than [250 scientific papers](#). Some of the important research areas that IO targets are cryptography, software engineering, distributed systems, networks, formal verification, and programming languages. Ouroboros, Cardano's consensus protocol, is explained in detail in section [Ouroboros Consensus](#). Ouroboros represents the outcome of long research presented in multiple papers. It is the first blockchain consensus protocol developed through peer-reviewed research with mathematically proven security features that guarantee the protocol is secure as long as 51% of the active stake (ada locked in for an epoch snapshot) is held by honest participants. The extended UTXO (EUTXO) model, explained in section [The EUTXO model](#), was also developed through extensive research and is well-tested by the community. There have been no major security breaches, and the Cardano mainnet has not experienced any downtime. An event happened in January 2023 that affected the functionality of around half of all Cardano nodes. During this event, the network was still processing transactions, and within minutes, nodes started to recover by being manually restarted by stake pool operators or by system management software. After the restart, the nodes that were down managed to continue growing the chain and producing blocks. The consensus algorithm worked, and the network never split, which was due to Cardano's robust design. The issue was later identified and fixed. Details can be found in [this](#) and [this](#) post-mortem.

In EUTXO, transactions can be validated *before* they are submitted to the blockchain. Given that the inputs are present at the time of submission and that the validity interval contains the time of transaction processing, a valid transaction will *always* succeed with a predictable outcome. There is no global state and a smart contract only sees data contained in the script context that provides access to the datum if present, the redeemer, all

inputs and outputs and other transaction information. Section [Plutus smart contracts](#) explains these concepts in detail. We say that transactions are deterministic in EUTXO since their effect on the ledger is predictable. If the inputs to a valid transaction are not present at the time of submission, the submitted transaction will not be processed, and the user will not be charged any fees. Because of determinism in EUTXO, multiple transactions that target the same validator can be processed inside the same block if the transactions consume different inputs. Beside this, [reference inputs](#) enable multiple transactions referencing the same UTXO, which is possible because none of those transactions consumes the referenced UTXO. These things make Cardano suitable for parallelism and concurrency. One can read more about these topics in this [IO blog](#).

In Ethereum, transactions that target the same validator can be processed within the same block, but they both do not necessarily succeed, since the first transaction could alter the state such that the second transaction fails. Because the state can change between when a user submits a transaction and when it is processed, this means an Ethereum transaction can fail after the user pays the transaction fee. We note that in Cardano, it is possible that a user forces an invalid transaction to be processed, which will fail. In such a case, the user will be charged fees because an attacker could try to flood the network with invalid transactions to congest it, and the fees make such an attack very expensive.

This [IO blog](#) explains in more detail the determinism of transaction and script validation in Cardano. The blog states that account-based blockchains like Ethereum are in-deterministic, which means that they cannot guarantee the predictability of the transaction's effect on-chain. It further explains the key points that make the outcomes of script and transaction validation predictable on Cardano, which are:

1. The script interpreter will always terminate and return the same validation result when applied to the same arguments

2. A transaction necessarily fixes all arguments that will be passed to the script interpreter during validation
3. A transaction specifies all the actions it is taking that require script validation
4. Compulsory signatures on a transaction ensure that it cannot be altered by an adversary in a way that causes scripts to fail
5. Applying a transaction in the EUTXO ledger model is deterministic.

Also, transaction fees are deterministic in Cardano. A user knows how much a transaction will cost before they submit it. The reason for this is that protocol parameters, rather than network traffic, govern pricing on Cardano. Determinism of fees and transactions can also be advantageous in Web3 applications, such as decentralized exchanges (DEXs). On DEXs users trade with the help of smart contracts that eliminate the need for a central entity which is in centralized exchanges holding custody of traders tokens. Liquidity providers that provide token pairs to DEXs, such that normal users can trade those tokens, face the issue of impermanent loss on DEXs that use the [Automated market maker](#) (AMM) design model. In an AMM-style DEX, smart contracts enable automated trading of cryptocurrency pairs, which are often a fungible token or a native cryptocurrency together with a stablecoin.

To explain how impermanent loss works, let us consider an example of an AMM DEX where the deposited token pair is ETH and USDT (Tether, a stablecoin pegged to the value of the USD. If the price of 1 ETH is 1,000 USD and a liquidity provider wants to add 1 ETH to the pool, he must also add 1,000 USDT. Then, over time, the price of ETH rises to 3,000 USD, which is three times larger than the initial price. During this time, [arbitrage traders](#) add USDT to the pool and remove ETH from it to take advantage of small price differences across two different markets. The pool reserves change such that the initial amount of ETH that the liquidity provider added, gets divided by 1.732 (the square root of 3), and the USDT

amount gets multiplied by 1.732. That means that now every ETH in the pool is backed by 3,000 USDT. If the liquidity provider now wants to withdraw his funds, he gets around 0.5774 ETH and 1,732 USDT, which at the current market price of ETH adds to around 3,464 USD. But if the trader had held on to the initial funds of one ETH and 1,000 USDT, he would now have 4,000 USD, meaning, his impermanent loss was around 536 USD. The reason it pays off to be a DEX liquidity provider is that they get a percentage of the trading fees. Arbitrage and normal traders that both use the pool pay for every transaction they make. The percentage of trading fees that liquidity providers get can be larger than the amount of impermanent loss. This [IO blog](#) states that the EUTXO accounting model is a suitable environment for DEXs using the order book model. An order book model simply lists the buy/sell orders, and if there is supply and demand, the asset can be traded. An advantage of this design is concentrated liquidity (liquidity allocated within a custom price range). This minimizes the effects of impermanent loss, creating a more stable environment for liquidity providers. The blog explains in more detail how impermanent loss affects DEXs built on the account-based accounting models compared to EUTXO.

Cardano's design also makes front-running invasion attacks that could target transactions on a decentralized exchange harder to carry out. The Cardano node is designed such that it processes transactions in the same order as they arrive into the mempool independent of the transaction fees associated with them. This is not the case for all blockchains where a node might favor transactions that carry higher fees. It is still possible for an adversary that operates a stake pool to inspect transactions that arrive in the mempool since they are not encrypted. They could then create their own transaction with similar parameters and insert it in front of the transaction that was received in the pool. In reality, this is hard to achieve because Cardano's mainnet has a high number of block producing nodes (around 3000 at the time of writing) and also because of Cardano's design, none of these nodes controls a significant amount of staked ada. You can

read more about this topic in chapter [Stake pools and stake pool operation](#). We also note that for transactions that interact with the same validator and spend different UTXOs, rearranging them is inconsequential because there is no global state that would affect both of the transaction. You can read more about the mempool and front-running invasion attacks on Cardano in the following [Cexplorer blog](#).

Since Cardano transactions are deterministic, their interaction with smart contracts is also deterministic, meaning the outcome of the validation logic is predictable. In Ethereum, a smart contract function could, in the short term, give up control of a transaction when it calls another contract. This other contract could then make recursive withdrawal calls to the first contract and drain its funds before the first contract updates its state. This is known as a reentrancy attack. The following [geeksforgeeks post](#) provides more information on such types of attacks and lists also examples of such attacks. Because such attacks are possible in Ethereum, the interaction of transactions with smart contracts is sometimes hard to predict. A reentrancy attack happened to the [DAO smart contract](#) that was [hacked in 2016](#). On that occasion, 3.6m ETH (worth around 50 million USD at the time) was affected. Such attacks are not possible in Cardano due to the design of the EUTXO accounting model. EUTXO's determinism doesn't allow the outputs to be different from what the user signed. Even if a validator is dependent on the decision of another validator, that is predetermined and known in advanced before submitting a transaction. Such a case is possible for instance if we want to make sure that a transaction runs a minting policy in tandem with a spending script. We can check in the spending script that tokens from that policy are being minted. And in the minting policy, we can check that a UTXO from that spending address is being consumed in the transaction. Those validators are not *calling* each other, but we know they have to be both triggered by a single transaction, since we enforce this condition inside those validators. When we build a transaction, we have to specify all the validators that run and all their inputs and outputs, so there's no way that

an untrusted validator can take over that transaction, since it's not part of it.

The EUTXO model also treats its tokens as native, meaning they are treated the same way as ada. For this reason, they get the same security and transfer logic as ada does. Some other blockchains use smart contracts to manage tokens and to set the allowed transaction logic, as Ethereum does with [ERC-20 tokens](#). Buying or selling these tokens requires interaction with a smart contract, which incurs higher transaction costs. Such tokens also carry a higher risk because the security of the token is based on the smart contract's correctness and security. This [Emurgo blog](#) states that blockchains that use smart contracts to manage native tokens can sometimes require smart contracts to have a hierarchy of permissions, as in a DEX, for example. If in such a DEX, the web of authority delegations has a complex structure, design mistakes can happen. This can cause an attacker to be able to exploit the system, potentially resulting in a user who wants to swap his tokens to lose their funds. So, a user who wants to use such a DEX must be sure that all the contracts are safe. Another security advantage of native tokens is that over- and under-flow vulnerabilities are not present, as Cardano's scripting language does not have fixed-size integers and the ledger itself tracks the movement of tokens. The table below summarizes some of the security features of Cardano native tokens.

Cardano native tokens	Comment
No user errors in copying standard code	All shared functionality is provided by the ledger
No over-/under-flow vulnerabilities	Cardano's scripting languages don't have fixed-size integers
No unprotected functions	An untrusted validator can not take control over the transaction

As in the [Minting policies and native tokens](#) section we again state that

further information on Cardano native tokens can be found at:

- [Official Cardano docs](#)
- [Cardano developers docs](#)
- [Cardano ledger docs](#)
- The [Native Custom Tokens in the Extended UTXO Model](#) scientific paper.

Cardano also enables smooth upgrades to its protocol using a hard fork combinator (HFC). Hard forks introduce radical changes to the blockchain protocol. They might cause downtime, a chain restart, or compatibility loss. In Cardano, the HFC is designed to enable the combination of several protocols without having to make significant adjustments. It allows Cardano nodes to run multiple protocol versions and enables smooth transitions during hard forks. If a hard fork upgrades the Plutus language to a new major version, smart contracts using an older version of Plutus can still be processed. You can read more about the HFC in [Cardano node layers](#) section, subsection **Consensus and storage layer**.

Cardano smart contracts see a predictable number of inputs and can only produce a set number of outputs that are always the same given the same inputs. Because of that, they are also perfect for audits. The job of a smart contract auditor is also more straightforward because outcomes that cannot be foreseen are minimized by the design of the EUTXO model. A smart contract auditor only needs to follow the logic of the validator to assess if all possible outcomes are desirable and is not concerned with a global state that might influence the final output. Besides auditing smart contracts, off-chain code and other parts of a DApp may also be audited to assess the security of the entire application.

Further learning resources about the EUTXO model in addition to section [The EUTXO model](#) can be found at:

- [The EUTXO Handbook](#)

- [The Extended UTXO Model](#) scientific paper
- [EUTXO explained on Cardano Docs](#)
- [EUTXO vs account-based models infographic](#)
- [Six reasons why EUTXO wins](#) blog.

In this section, we presented some of Cardano's security features. The security features of the Plutus and Marlowe smart contract languages are further discussed in the remaining sections of the [Smart contract security](#) section.

25.7.2. Plutus security

Cardano offers Plinth, previously called PlutusTx, which is a Turing-complete smart contract language realized on top of Haskell, a high-assurance, pure functional programming language with a strong type system. You can read more about the characteristics and advantages of the Haskell programming language in section [Features and benefits of Haskell](#). The Glasgow Haskell Compiler (GHC), that is the most well-established Haskell compiler, is used to compile Plinth code. On-chain code is compiled into Untyped Plutus Core (known simply as Plutus), which is the scripting language used by Cardano to implement the EUTXO model. Cardano nodes execute the compiled Plutus scripts during transaction validation with the help of the Plutus virtual machine (VM). A VM is a virtual environment that executes code securely inside a node, when talking in context of a blockchain. When a Cardano node executes validator code compiled to Untyped Plutus Core, this code does not access the computer's resources directly as a normal computer programs would do. For security reasons, VMs offer a safe execution environment. Every Cardano node has a built-in Plutus virtual machine that is written in Haskell. There is also a plan to develop a Cardano node written in the Rust language. The project is managed by the [Pragma](#) member-based organisation (MBO and is called [Amaru](#).

Plutus is a simple, functional language similar to Haskell, and a large subset of Haskell can be used to write Plutus scripts. Its simple and deterministic design allows careful cost control of program execution. Plutus is also a variant of [lambda calculus](#), specifically, [System F](#). Lambda calculus is a formal system in mathematical logic for expressing computation based on function abstraction. GHC's internal language (GHC Core) is also based on System F, but it does differ from Plutus in some ways. The Plutus language was designed with the following requirements in mind:

- minimalism
- safety
- formalization
- size
- extensibility
- multiple source languages.

You can read a short explanation of each of the above requirements in the [Plutus Platform](#) technical report, and you can read more about the design details of the Plutus language in its [formal specification](#) document. The [Plutus](#) GitHub page features links to related materials.

Plutus scripts can be generated using Plinth, a GHC plug-in that runs during the GHC compilation process. It modifies the program that GHC is compiling however it likes. Under the hood, though, the compilation process is more complex. The image below shows the compilation process of the on-chain validation code.



Figure 48. Plutus compilation pipeline

We start with Plinth, a subset of Haskell representing the high-level language developers use to code the smart contract. Compiling this code goes through several steps:

- First, it transforms into the GHC core, part of the GHC compilation pipeline.
- Second, it gets compiled into the Plutus Intermediate Representation (PIR). From now on, it will be independent of GHC and Haskell and have total control of the compilation pipeline.
- Third, it transforms it into Plutus Core (Plutus). This is an intermediate form that the compiler uses to statically check the program.
- Fourth, we get to Untyped Plutus Core, which the Cardano node runs to validate the transaction. This language is hard to code directly, and it is usually stored in binary representation.

It is wise to break down compilation pipelines by introducing intermediate languages to ensure no step is too large, and to test each step independently. For more information about the compilation process, refer to this [IO blog](#).

You can also use a different high-level language if you have a compiler that compiles to Untyped Plutus Core. Some projects are doing this by replacing parts -or all- of the compilation pipeline. Validators can be coded in pre-existing languages like Typescript ([Plu-ts](#)) and Python ([OpShin](#)), or they can be coded in new languages like [Aiken](#). A comparison of Aiken, which is the most used smart contract language in the Cardano ecosystem as of 2023, and Plinth is presented in section [Plinth in comparison to Aiken](#).

Haskell was also chosen as a programming language to implement the Cardano node which contains the ledger, consensus and network layer protocols that define Cardano. You can read more about these protocols in chapter [How Cardano works](#). The choice of Haskell enabled the research

and engineering teams that work on Cardano to reuse the existing Haskell infrastructure, libraries, and tools that enable developers to build high-assurance software. To assess the security of the Plutus language, formal methods were used that are the most stringent form of reasoning about code correctness. Formal methods are mathematically rigorous techniques for the [specification](#), development, [analysis](#), and [verification](#) of software and hardware systems. They are not well suited for any arbitrary programming language but work well with pure functional programming languages such as Haskell. With formal methods, one can mathematically prove that the software under evaluation is correct and does exactly what it is supposed to do.

In the process of formalizing Plutus, first starting with the paper specification of the language, the semantics of the language were formalized, and then the properties of the language were proven to indicate that the design of the language was correct. Semantics are given first by describing the reduction semantics of the language with [Structural Operational Semantics](#) (SOS). Then, the semantics on an abstract machine (the [CEK machine](#)) are given and shown as equivalent to the SOS. Other properties are also shown. For example, the soundness and correctness of the typing system. From this formal specification, an executable is produced by compiling the specification into Haskell. This executable is a [reference implementation](#) that behaves correctly (by definition), but it's not efficient enough to be used in production. Then, real (efficient) implementations can compare their results against those of the reference implementation. This is done by automated testing in the conformance test suite. The production code is tested against the reference implementation. The [Plutus](#) repository contains the formalization of Plutus in Agda. The project is located in the *plutus-metatheory* folder. You can read more about formal methods and functional programming in section [The importance of program correctness for cryptocurrency ledgers](#).

Besides formal methods, property-based testing is also used to deliver high-assurance software for Cardano. For Haskell code, property-based tests can be written using Haskell libraries such as [QuickCheck](#) or [Hedgehog](#), which allow developers to state properties that should always be in a program. Those libraries use sophisticated algorithms to generate test cases and search for minimal counterexamples that violate those properties. This [IO blog](#) further explains that for code interacting with the external world, particularly network applications such as the Cardano node, it can be hard to find minimal counterexamples. This is attributed to the fact that the execution order is not deterministic because it can change every time the software runs. In other words, the same code can be run hundreds of times and only fail once. The blog explains that a way to get around this is by using simulations with deterministic execution order. Running tests in such a simulation allows developers to reliably find and fix a class of bugs in testing that would otherwise only occur randomly in production.

Formal verification and property-based testing can also be used to secure the development of smart contracts. While Haskell is a great programming language for implementing reliable software, it cannot catch all possible mistakes a programmer can make, so the code still needs to be tested. One can write property-based tests for PlutusV3 Plinth smart contracts by combining the following tools:

- [Atlas](#) a Plutus application backend developed by GeniusYield
- [Cardano ledger backend](#) (CLB) library developed by MLabs
- [QuickCheck](#) Haskell library.

This [MLabs blog](#) shows how to use Atlas with the CLB to test plinth scripts. The Atlas application backend is an open-source tool that covers all functionalities needed to work with Plinth smart contracts and enables writing the complete server side code in Haskell. It is the Haskell alternative to TypeScript off-chain tools such as MeshJS and Evolution-SDK

. It was developed by the company [GeniusYield](#) in collaboration with other companies such as MLabs, Well-Typed, and Plank. It significantly abstracts away the complexity of building Cardano transactions and interacting with Plinth smart contracts. Different data providers can be configured to power the Atlas framework. It streamlines the process of building transactions and executing smart contracts. One can read more about the benefits that the Atlas application backend provides at their [documentation page](#). Another tool one can use for testing Plinth smart contracts or any other smart contract language that compiles to Untyped Plutus Core is the [Cooked validators](#) library developed by Tweag. Aiken, that is a standalone smart contract language comes with its own build-in toolkit for testing. You can read more about it at the [Aiken documentation](#).

For testing smart contracts, third-party auditing companies can write the tests as part of their auditing process of DApp smart contracts and server-side code. Their auditing services may also include formal verification. For Cardano DApps, companies such as Tweag, MLabs, Runtime Verification, Hachi, Vacuumlabs, FYEO Inc., Anastasia Labs, TxPipe, and some others, offer Cardano auditing services. IO is also working with some of these companies on a certification program for DApps running on Cardano. The program creates three levels of certification for DApps in a store.

- Level 1: automated tooling
- Level 2: in-depth audit
- Level 3: formal verification.

The below descriptions were originally included in this [IO blog](#):

- **Automated tooling** will give continual assurance about a range of properties for smart contracts. It covers the discovery of different types of issues or bugs and is characterized as low cost, low effort, and accessible to everyone while providing a substantial level of assurance. It can be applied repeatedly and automatically, so each time there is a release or a sub-release of an application, the application can be tested

to confirm it still has the properties that we expect.

- **In-depth audit** looks at the technology and processes that led to it being produced. It involves a manual audit and verification of smart contracts within the DApp itself. The testing is performed at a much deeper level and involves more manual effort than can address a DApp in its entirety, even if it is written in a variety of languages.
- **Formal verification** is more specialized, where the auditor aims to provide full assurance of critical aspects of applications through formal verification of smart contracts. Formal verification involves ensuring that a smart contract serves the specific business or technical requirements defined at the outset.

When using formal verification for smart contracts, the general procedure is to first define the model that serves as an exact representation of the smart contract behavior. Then, the smart contract's specifications are defined in a language that both the model and the verification tools can comprehend. After that, the process of verification can start, where mathematical tools and logic are used to prove the correctness of the defined assumptions. In the end, the results of the verification process are analyzed, and issues are addressed.

IO is also developing a new formal verification tool for Cardano smart contracts. Proof assistants such as Agda and Rocq (formerly Coq), offer strong guarantees, but require experienced developers to use. This new tool combines the power of the Lean4 proof system and SMT solvers, such as Z3, to automatically check that contracts behave as intended. The goal is to eliminate the need for developers to write any formal proofs themselves. It will work by simply clicking a button and the targeted smart contract will be formally verified. This tool can be integrated directly into the CI/CD pipeline and contracts can be formally verified on every commit – instantly, continuously, and affordably. The tool still remains under development at the time of writing. One can read more about the tool in this [IO blog](#).

In case a company decides for an audit, no matter the audit type, the company requesting it has to provide a standard set of documents and information:

- General specification and design documents that unambiguously describe the architecture, interfaces, and requirements characterizing the DApp.
- On-chain specification that clearly specifies the smart contract properties.
- Off-chain specification that describes the expected behavior (assumptions, constraints, and requirements) for all interfacing components of the DApp.
- A description of how the DApp has been tested, together with the results of the tests, and details of how those test results can be replicated.
- A final version of the source code together with versioning information.

You can read further details of the above points in cips.cardano.org/cip/CIP-52[CIP-0052], which defines guidelines and best practices for auditing DApps built on Cardano. Furthermore, cips.cardano.org/cip/CIP-57[CIP-0057] defines a language specification or so-called blueprint for documenting Plutus contracts through the form of a JSON schema that sets the vocabulary and validation rules to specify the Plutus contract interface. This enables a better understanding of Plutus smart contracts written in any of the high-level languages that compile to Plutus. It allows for the development of tools that generate contract API references and documentation. DApp developers also make their contracts easier to audit as they can specify the expected behavior, and the auditing information is provided in a more standardized way.

Different companies may use different approaches to perform smart contract audits. Sometimes, auditing companies publicly disclose their methodology. This [Tweag blog](#) presents the auditing process in their

company. Beside auditing smart contracts, other parts of a DApp can be audited as the off-chain code. After the audit process, the vulnerabilities are usually identified and grouped into categories:

- **Critical vulnerabilities** that pose a big risk for the DApp
- **High vulnerabilities** that may affect a large portion of the DApp's security
- **Medium vulnerabilities** are issues that affect the proper working of a DApp but are not security threats that would need immediate action
- **Low vulnerabilities** that may affect things such as performance but pose no risk to security.

We state at the end of this section that good practice in writing Plutus smart contracts is to avoid common security vulnerabilities which are:

- other redeemer
- other token name
- arbitrary datum
- unbounded datum
- unbounded value
- unbounded inputs
- multiple satisfaction
- missing UTXO authentication
- UTXO contention
- cheap spam
- insufficient staking key control.

This [MLabs guide](#) describes each vulnerability by a property statement, test, impact, and provides an explanation of it. Audit reports are often made publicly available for the community to read because they are a great tool for learning what mistakes not to make when developing a

DApp. Below are links to repositories from the auditing companies Tweag and Vacuumlabs that made some of their audit reports freely available to the public:

- [Tweag audit reports](#)
- [Vacuum Labs audit reports.](#)

Other auditing companies that might not have their auditing reports grouped into a single repository, are also worth exploring.

25.7.3. Marlowe security and best practices

Marlowe security

In the design process of the Marlowe smart contract language, trade-offs were made in regard to how powerful the language should be. In computer science, the [Halting problem](#) states that it is impossible to write a program that takes as input another program written in a Turing-complete programming language, and decides whether that program will stop or not. Such programming languages are so powerful that it is difficult or impossible to automatically analyze programs written in them. You always have to write tests or use other methods as formal verification to check certain properties of such programs. In practice, for a large class of programs, it may be possible to have automated tools, but there can be no general tools that decide whether the program stops or not. This is only one example of undecidability. More often, the security properties of the program need to be checked and it must be proven that the program behaves as expected. On the other hand, if you are willing to give up Turing-completeness, you can have simple but powerful program analysis tools and program language built-in security features. Powerful analysis tools also exist for Turing-complete programs; they are just more complex to build, and such programs have fewer built-in security features.

Marlowe is not a Turing-complete language, so you cannot express arbitrary logic in it. This gives the option to statically analyze how long a

contract will run. All Marlowe contracts are guaranteed to have a finite lifetime. There are other things Marlowe contracts take care of, such as ensuring that no funds are forever locked in a contract. In Cardano's implementation of Marlowe, there is an edge case where it is possible to lock funds by creating a contract that exceeds the limitations of the blockchain, for example, by making too many external payments in the same transaction. The chapter *Tools for Detecting Potential Locking/Blocking of Marlowe* in the [Marlowe best practices](#) page explains how to prevent such a scenario. In the real world, you can enforce contracts by using the legal system or financial regulations. However, due to the decentralized nature of the blockchain, this may not be possible. When you use Marlowe to create a smart contract, you can't force anyone to make a payment that fulfills a contract. Marlowe considers that and implements a finite lifetime mechanism to wait for someone to make a payment. If the contract is waiting for an action, like a payment, and that action does not happen, the finite lifetime mechanism enables a timeout mechanism that always says what action can happen when the timeout is reached. Marlowe provides the following safety guarantees:

- Execution always terminates
- Contracts expire eventually and deterministically
- No assets retained when the contract reaches its final state
- Conservation of value.

The safety-first principle used in Marlowe's design has limited its functionality for a good reason. There are two additional limitations considered when a Marlowe contract is created:

- (1) the number of participants in a contract is limited to the predetermined amount when the contract was designed;
- (2) existing Marlowe contract logic cannot be revised. To ensure the above-mentioned safety guarantees, the following programming language

constructs are absent from the Marlowe language to ensure Marlowe's safety:

- Recursion is not allowed.
- Looping is not supported.
- Functions or macros may not be defined.
- Timeouts must be numeric constants.
- Only **Case** continuations may be Merkleized. The Faustus programming language relaxes some of the limitations above, yet it compiles to safe Marlowe. Besides Faustus, you can "relax" the above limitations if you embed Marlowe into another language (JavaScript, for example). By using the TS-SDK, you can have JavaScript functions and loops that unfold a larger Marlowe contract.

Merkleization is the process of constructing [Merkle trees](#) that can be used to verify any kind of data stored, handled, and transferred in and between computers. It enables large smart contracts to be concisely represented on-chain. Instead of storing gigabytes of data representing a huge contract, only the hash of that contract needs to be represented on-chain. You can learn more about Merkleization in Marlowe in [this video](#) from IO Academy. Merkleization is hidden through the Marlowe object concept, and you can read about the marlowe-object package in its frontend [documentation](#). The [Marlowe Object Format](#) is a way to describe a Marlowe contract as a set of labeled objects that reference one another, which can be linked to create a Marlowe contract. The term comes from the object files produced by a compiler, which can be linked to create an executable program.

Marlowe also contains additional constraints imposed by Cardano:

1. No more than one Marlowe contract may be run in a single Marlowe transaction
2. Token names are limited to 32 bytes

3. An invalid initial state will prevent a Marlowe contract from executing properly on Cardano. Some of the smart contract execution paths may be blocked
4. Each transaction output on the Cardano ledger must include a minimum lovelace value in the UTXO
5. Because the Cardano ledger contains a rule that the size (in words) of the bytes storing the information about native tokens in a UTXO cannot exceed the protocol parameter maxValueSize, there is a limit to the number of different types of tokens that can be held in a single UTXO
6. The size (in bytes) of a transaction is limited by a Cardano ledger rule specified by the protocol parameter maxTxSize
7. If a Marlowe contract does too much computation or uses too much memory in a transaction, it may violate the rule that execution costs in Plutus are limited to specific usage of "memory" and CPU "steps".

You can look up the Marlowe [best practices guide](#) to read more about these limitations. While Marlowe contracts might run on the Playground, they would not run on-chain if the contract violates the [Cardano ledger restrictions](#). You can check a Marlowe contract for compatibility with the Cardano ledger rules by using the *marlowe-cli run analyze* command. The Marlowe CLI [GitHub page](#) provides additional information about this topic.

The following properties regarding the safety of Marlowe were proven using formal methods:

- Conservation of funds
- Contracts always terminate
- Positivity of account balances
- Quiescence
- Idempotency of reductions

- Composition of inputs.

The formal specification and proofs for the Marlowe language were made using the Isabelle proof assistant, which can be used for formal software verification. The proofs were audited and can be found in the [Marlowe GitHub repository](#). This repository also contains the `marlowe-spec` software tool, with which a Marlowe interpreter written in any smart contract language can be validated for compliance with the Marlowe formal specifications. The specifications can be found in the `marlowe-cardano` [GitHub repository](#) that contains the official Cardano implementation of Marlowe. The Marlowe interpreter and the role-payout validator are implemented as Plinth scripts and can be found in the `marlowe-plutus` [GitHub repository](#), which also provides instructions on how to compile the scripts.

The auditing company Tweag performed an audit of the Marlowe high-level language and the implementation of Marlowe on Cardano before Marlowe's deployment on mainnet. The audit discovered significant issues, including handling of negative deposits, prevention of "double satisfaction", enforcement of state invariants, an implementation difference between formal specification versus the Plinth implementation, and the proof of money preservation theorem. The issues were addressed before deploying Marlowe on mainnet. The full [audit report](#) can be found at Tweag's GitHub page. IO also issued a report on the [Marlowe GitHub page](#) explaining the actions taken. You can read a short description of the issues and how they were addressed in this [IO blog](#), which also talks about the security of Marlowe and Cardano.

Marlowe best practices

Marlowe contracts can be designed in many ways. Once you have designed your contract, it is good practice to simulate it on the Playground first and create a walkthrough of all possible execution paths. The Playground also offers a set of tools that can perform a static analysis of

the contract. No matter the Playground programming language you choose, the tools can be accessed from the bottom bar in the Static Analysis tab. Once you click on it, the analysis console opens up, displaying a section on the right-hand side where you can set the contract input parameters. Then, you can make three possible analyses:

- Analyze for warnings
- Analyze reachability
- Analyze for refunds on Close.

The analysis for warnings action analyzes the contract for potential mistakes a smart contract designer could make. An example of a warning would be that a contract would try to pay an amount of ada to a party that exceeds the total amount of ada that the contract is holding at the time of the payment action. The analysis will produce a description of the warning, and it will display an *offending sequence* that shows the contract actions leading to the warning. The reachability analysis action analyzes if any unreachable subcontracts exist. In other words, it checks if an execution path of the smart contract exists that cannot be reached with any possible transaction. The analysis for refunds on Close action analyzes if any of the Close constructs refund any money. That means it checks if all refunds are explicitly stated in the contract or if some happened because funds remain in the contract when it reaches the end of any possible execution path. Even if funds remain, this does not represent an issue since its owners can redeem the funds because no funds are locked by the contract indefinitely.

Besides being a smart contract programming language, Marlowe is also an ecosystem of tools to enable the deployment, interaction, and analysis of Marlowe smart contracts. Formal proofs, extensive testing, and analysis tools provide strong assurances for the safety of Marlowe contracts. Marlowe Runtime, which is used under the hood by the TS-SDK, provides off-chain services that discover Marlowe's contract history and build

transactions that apply input to Marlowe's contracts. If a project using the TS-SDK decides to deploy its own instance of Marlowe Runtime, precautions should be taken if there is an interest in examining the Marlowe transactions produced by the Runtime. If Marlowe Runtime is deployed as part of a web service, then one must be aware of the possibility of person-in-the-middle, cross-site scripting, and other attacks. Marlowe Runtime contains a registry of known Marlowe script versions that it uses to create new Marlowe transactions. One should only use Marlowe Runtime to create contracts if one trusts the script hashes in that registry. The Marlowe test suite verifies that the script registry has not been inadvertently or maliciously altered. Of course, this does not guarantee that the test itself has not been altered.

Another Marlowe tool is the Marlowe CLI, which includes capabilities for analyzing the suitability of a Marlowe contract for execution on the Cardano blockchain. As mentioned in the previous section, the *marlowe-cli run analyze* command enables a smart contract designer to check that his contract does not violate any of the [Cardano ledger restrictions](#). Its command line interface allows the user to select which checks to perform, and its output highlights the causes and location of violations. It is important to realize that the tool tests whether or not it is *possible* to avoid violating the ledger rules in executing a contract. This does not mean that it is *trivial* to construct every valid transaction for the contract. Further details can also be found in the [Marlowe best practices guide](#), which explains how to avoid protocol limits when designing Marlowe contracts in the following cases:

1. Choose role and token names that are no longer than 32 bytes
2. Ensure that the contract's initial state is valid
3. Ensure that the initial value in the contract contains sufficient lovelace
4. Avoid contracts that use an excessive number of native token types
5. Avoid or break up complex logic in the contract

6. Use reference scripts and merkleization to reduce on-chain size
7. Be parsimonious in the use of the contract's internal state.

Look up the [Marlowe debugging cookbook](#) for technical approaches on how to debug Marlowe contracts on Cardano. Some types of contracts are not well suited for using Marlowe on the Cardano blockchain, including:

- Contracts that contain timeouts spaced close to the Cardano block production rate
- Contracts that require floating point arithmetic
- Contracts that require many bound values in their internal state
- Contracts that make many differently named choices by many parties
- Contracts that have many alternative and simultaneous actions possible. For example, a distributed exchange (DEX), order book, market maker, or token marketplace would be poorly suited for Marlowe unless the Marlowe contracts were supervised by Plinth contracts or semi-centralized off-chain infrastructure.

Also, security and best practices should be taken into account when constructing transactions targeting a Marlowe contract. A user should be able to answer the following questions before signing a Marlowe transaction:

- Does the transaction operate a Marlowe contract?
- What is the current contract and what is its state?
- What input is being applied to the contract?
- What else is occurring in the transaction?

The [Marlowe security guide](#) answers all these questions in detail. As a conclusion, when developing a Marlowe smart contract, one should:

- Use the [Marlowe Playground](#) to simulate the contract and perform static analysis

- Use the *marlowe-cli run analyze* tool to check if the contract can run on the Cardano network and does not violate any of the Cardano ledger rules
- Read and understand the Marlowe guides referenced in this section and check if the Marlowe smart contract satisfies all criteria
- Test off-chain code by running all execution paths of the contract on a Cardano testnet. You can read more about the Cardano testnet at [Cardano docs](#).

Chapter 26. Decentralized Applications

Decentralized Applications (DApps) aim to empower end-users through ensuring that the encoded digital processes and data stored are not centrally controlled (as discussed in previous chapters). So far in the previous chapters, we have covered some of the components that support decentralized applications including the blockchain network, smart contract/on-chain scripts and wallets. In this chapter we focus on those technological components that support user interaction and bridge the gap between foundational blockchain and on-chain script infrastructure and users.

26.1. Traditional Web Application Architecture

To be able to appreciate the novelty that decentralized applications bring, we first provide an overview of traditional web applications. [Figure 11](#) depicts an abstract overview of traditional web application interaction.

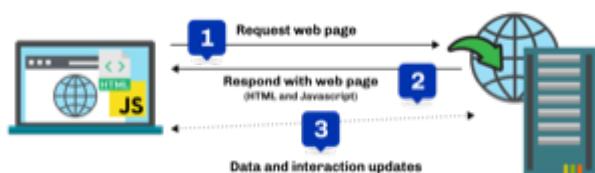


Figure 11. Traditional web architecture.

When browsing web sites, at an abstract level, the process can be broken down into three steps:

1. Your request for a web page (depicted in Step 1 in [Figure 11](#)). When you use a web browser on your computer (or mobile phone) and go to a website, really what is happening is your browser makes a 'request' to the web server that hosts the web page, asking it for the content of the

specific page.

2. The web server which is centrally owned and controlled by some entity, receives the request and puts together its 'response'—more specifically the web page content to return. This 'response' is sent back to your web browser, operating on your device (depicted in Step 2 of [Figure 11](#)).
3. Once a web page has been loaded in your web browser, subsequent data and interaction can take place between the web page itself and the web server (depicted in Step 3 of [Figure 11](#)). This could be to retrieve, for example, the latest news articles on a news website. Web applications are typically composed of more than one web page. Usage of one web page may lead to directing a user to another web page, resulting in a request for a new web page (which would initiate the process for requesting a new web page depicted in step 1).

Some web pages have static content, i.e. their content is always the same (and perhaps hard-coded). For example, consider a web page that always displays the text 'Hello World!'. While other pages may be more dynamic, which allow for the web server to decide on the specific content to respond with. For example, consider a web page that displays the current date and time—the content dynamically changes according to the date and time. Really, most web pages nowadays consist of some degree of static content and some amount of dynamic content. Though it is also not unheard of to have completely static web pages—for example, many company landing pages often have a static view of what they want their clients to see as the first page of their web site.

The actual 'response' that the web server sends back to your web browser most typically consists of two parts: (a) HTML code; and (b) JavaScript code (and some other media assets and styling sheets). The HTML code describes what the web page should look like.^[1] Your browser uses the HTML code it receives as a means of 'drawing' the visual aspect to the web page. Consider a web page that can be used as a calculator. The HTML would describe and provide the instructions for your browser to visualise

all aspects of the calculator including the different buttons on the calculator, and the display. Yet with such a visual representation (that HTML provides) only, pressing the buttons would not have any effect. A second component is needed to provide instructions to the browser that describes the functionality of the web page—i.e. for the calculator use-case, functionality for displaying a '1' when the '1' button is pressed, and for adding two numbers together and displaying their result when required. This functionality is encoded within the JavaScript component of a web page.

The HTML and JavaScript code described above is sent to your browser on your computer, and is executed in your browser (on your computer). Really this means that you could manipulate the HTML and JavaScript code, which means you can change what a web page you interact with looks like, and also its functionality.^[2] This code received from the web browser, which is executed in your browser, is what we call "client-side"—i.e. code that executes in the client's^[3] browser. Since the code can be manipulated by yourself (the client), web site operators cannot just trust any interactions or code that take place on your device, and it is for this reason that some functionality is not encoded into the 'client-side' HTML and JavaScript but encoded into 'server-side' code which executes on the web server owned by the particular entity the original web page was received from. For the code that is executed on the web server, the website operator has full control over that logic and as users of that website we must trust the operator. To exemplify this further consider an online shopping platform. Whilst end-users are displayed with products and prices on screen and are able to select which products they would like to purchase, the site operator cannot trust end-users to calculate the total cost due after selecting products. Since as described above, end-users would be able to manipulate the total calculation taking place within their browser. Therefore, the web site is built in such a way that code concerned with total calculation is ultimately done on the web server ('server-side'). Once a total is calculated and a client is ready to pay, clients'

are typically sent to a payment page where they enter their bank card details. Again, processing of card transactions is typically done 'server-side' for the same reasons. Yet, in this scenario, clients that are making purchases need to trust the operator and the code that is handling the total calculation (and executing on the web server) to not overcharge after entering their card details. Traditional web applications and their architectures heavily support keeping site operators in a position of power, i.e. where platform providers are in full control and end-users are reliant on the platforms, and must trust that the platforms do their job properly—and when they do not must try to seek recourse in some other manner if possible (e.g. through contacting the operator, or seeking legal recourse if necessary and possible).

26.2. Decentralized Application Architecture

The advent of blockchain platforms and smart contracts built on them, provided a new place where code could be executed, i.e. in smart contracts on a blockchain. Executing code in smart contracts, unlike code executed in client browsers or server-side, provides guarantees that the code cannot be manipulated or altered—not even by the programmer that created the code! Furthermore the code executed in smart contracts is publicly available, and therefore users can look into the smart contract logic prior to interacting with them. Through the availability of smart contract code, a new architecture was possible that enabled for Decentralized Applications (DApps). DApps build on traditional web applications, in that they also make use of client-side and server-side code, yet smart contracts can be utilized to provide guarantees to all stakeholders. Consider the example discussed in the section above, regarding the calculation of the total due for an online shopping platform. If the calculation of the total is done on the client-side, then users could manipulate the amount due, therefore service providers calculate totals on the server-side. While this makes complete sense, server operators could manipulate the logic to calculate a higher total due—though end-users are likely to notice this before

payment. Yet in a similar vein server operators could display one amount, and process a transaction for a different amount. This is really the status quo of traditional web architecture. Using a DApp, instead such logic could be delegated into a smart contract that would provide guarantees to both the service provider as well as to end-users, since the code is transparent and publicly available for everyone to see. Furthermore, the code is immutable, it cannot be changed—not even by the developer that wrote it. We will now provide an overview of typical DApp architecture, followed by a walkthrough on how to use a DApp, and then close off with code snippets demonstrating how a DApp can be built. [Figure 12](#) depicts an abstract overview of typical DApp interaction.

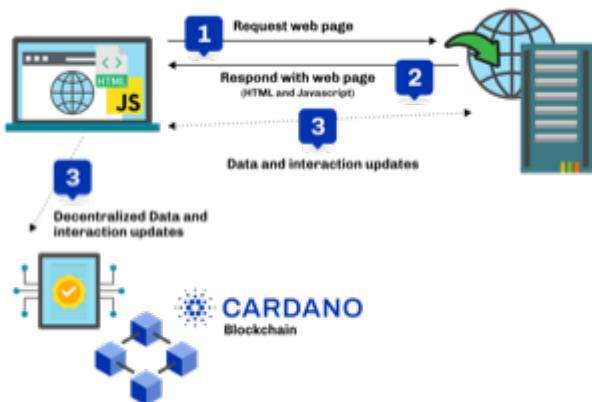


Figure 12. Decentralized Application (DApp) architecture.

Interaction with DApps, at an abstract level, can be broken down into three steps exactly like traditional web applications. Yet following the initial web page request (depicted in step 1) and web page response (depicted in step 2), future interaction (depicted in interactions marked as 3) between the client browser is not restricted only to communication with the centralized platform (or other centralized services), but can interact with decentralized platforms such as Cardano’s blockchain and on-chain scripts running on top of it. Consider the same example discussed above. Initially, an end-user can request the web page that displays different products (and in turn their browser receiving the web page including HTML and JavaScript). Thereafter, when the user wants to select a product to purchase which will require for the total to be calculated, this interaction could take place directly from the client’s web browser with

on-chain code deployed on the blockchain. Thereafter, payment could be made again through direct interaction between the user's web browser and on-chain scripts. Fulfilment of the order could even take place by underlying on-chain scripts—if the service is one that can be fulfilled digitally through on-chain scripts, e.g. if a user was buying a particular token in exchange for another cryptocurrency.

While the server is responsible for delivery of the initial HTML and JavaScript sent to the client's web browser, and even most of the logic coordinating future interaction, any required guarantees can be provided based on logic encoded in on-chain script code. Really the front-end HTML code, and JavaScript code that coordinates the interaction with underlying on-chain script code can be seen as the gateway facilitating interaction with the actual logic encoded in the on-chain scripts. If it was desirable to do away with the centralized request/response delivering the initial HTML/JavaScript code, then other distributed platforms could be used to replace the centralized components such as IPFS (which will discuss in more depth later in this chapter).

To further exemplify how guarantees can be provided through on-chain scripts (in spite of initial HTML/JavaScript code being delivered by centralized infrastructure), we'll now introduce how to use a DApp, and then later how to build a DApp.

26.2.1. Prerequisites

We now list a number of prerequisite steps required in order to follow upcoming sections on how to use a DApp and how to build a DApp.

Installing a Wallet

Prior to building a DApp, we need to ensure that we have a wallet installed in a browser that will allow for the client web-page to communicate with the blockchain directly. We'll use the Lace wallet (introduced in [Chapter 6](#)) in the rest of this chapter, but you may use any other wallet that supports

the Cardano blockchain.

Configuring Wallet to Connect To Testnet

We'll configure the Wallet to connect to the Cardano test network so that we can test without having to spend real cryptocurrency. In Lace, you can do this by:

1. Clicking on the currently selected Wallet (as depicted in [Figure 13](#))
2. Then selecting 'Settings'
3. Then click on 'Network' to 'Switch from mainnet to testnet'
4. Click on 'Preprod' which is meant for pre-production testing

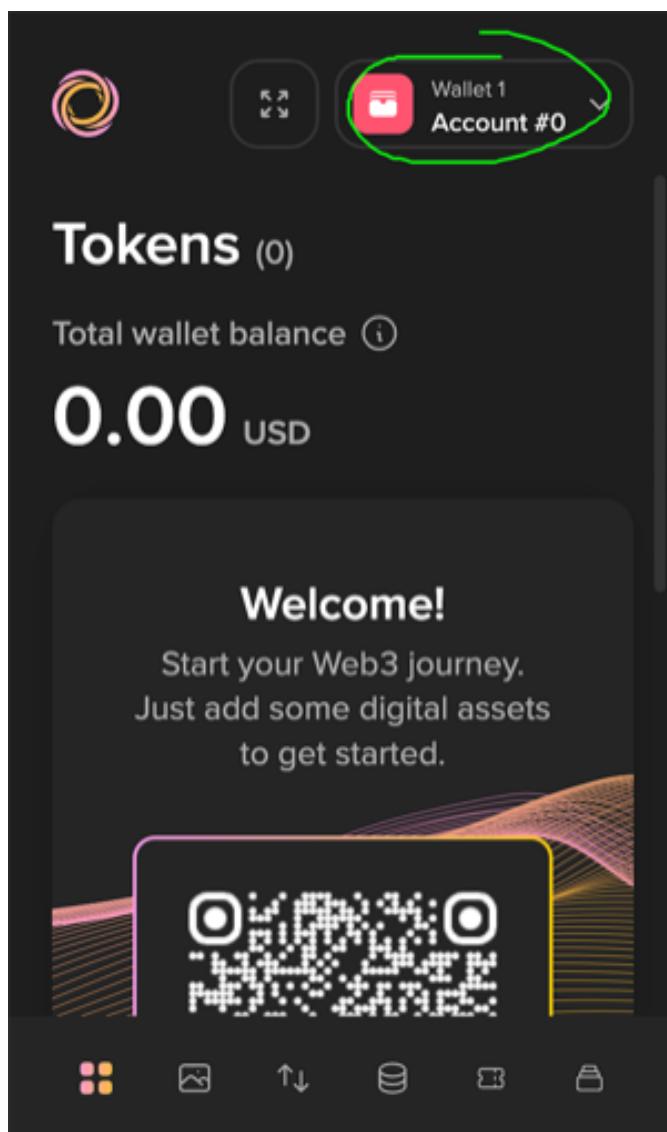


Figure 13. Finding Wallet Settings.

You can check whether you are connected to a test network in Lace to see

if the test network is listed at the top of the wallet screen as depicted in [Figure 14](#).

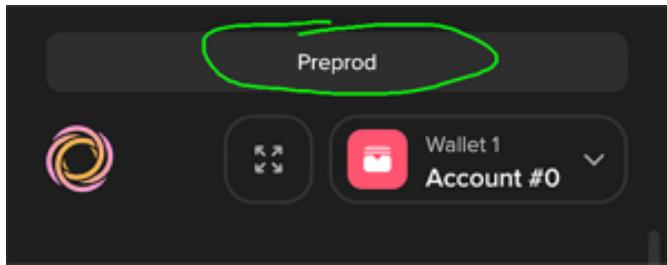


Figure 14. Preprod indication.

Receiving Test Cryptocurrency

In order to interact with the blockchain, users must spend some cryptocurrency. Since we want to avoid spending real cryptocurrency whilst testing we've switched to the Preprod test network (as discussed above), and need to obtain some test cryptocurrency. To do so we'll request some test Ada (Cardano's cryptocurrency) from a faucet.^[4] One such faucet can be found here:

<https://docs.cardano.org/cardano-testnets/tools/faucet>

To retrieve test Ada, configure the fields as follows:

- Environment: Preprod Testnet
- Action: Receive test ADA

Then, copy your wallet address. In Lace this can be done by clicking on 'Copy address' located at the bottom of the main screen of the wallet as depicted in [Figure 15](#). Then paste the address in the address field. Ensure to click on "I'm not a robot" and press 'Request Funds'. A success message should appear shortly, and the test Ada should appear in your wallet within a few minutes.

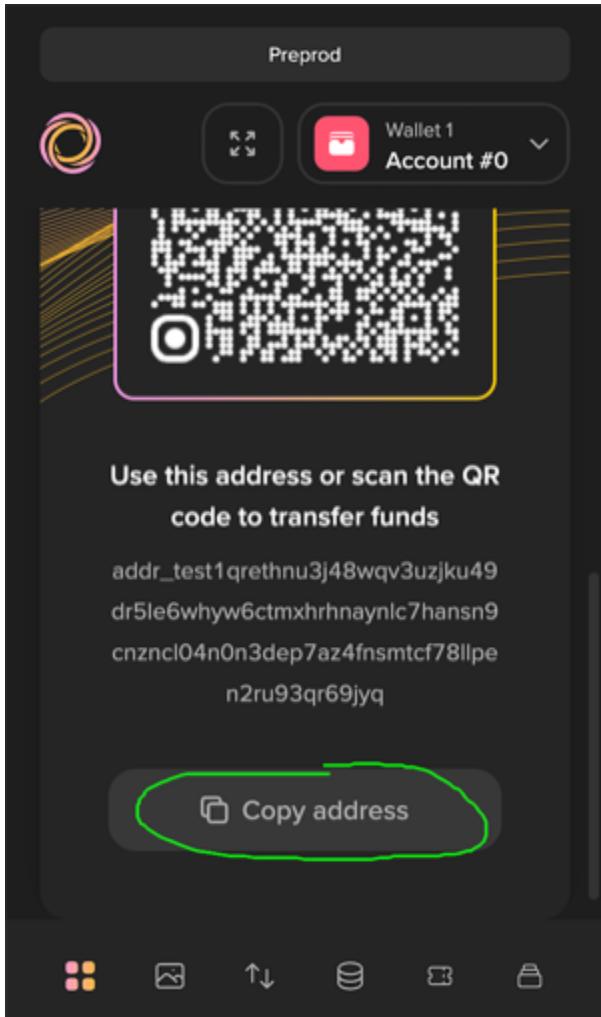


Figure 15. Copy wallet address.

The filled in details are depicted in [Figure 16](#).

Note: The public address of the wallet is hidden, since all transactions are publicly available for anyone to see. You should keep this in mind when sharing your wallet addresses.

A screenshot of a web browser showing the "Testnet faucet" tool on the Cardano Testnet website. The URL is "docs.cardano.org/cardano-testnets/tools/faucet". On the left, a sidebar lists "Testnet environments", "Getting started with Cardano testnets", "Creating a local testnet", "Resources", "Support and feedback", and "Tools" (which is currently selected). Under "Tools", "Testnet faucet" is highlighted. The main form has fields for "Environment" (set to "Preprod Testnet"), "Action" (set to "Receive test ADA"), "Address required" (containing the redacted wallet address), "API Key optional" (with an empty API Key field), and a "ReCaptcha" checkbox. A "Request funds" button is at the bottom.

Figure 16. Requesting test Ada from a faucet.

26.2.2. Using a DApp

Now that we have some test cryptocurrency in our wallet, let's try to use a DApp. We'll use a decentralized exchange (DEX) to swap some of our testnet Ada for some other token. More specifically we'll use a preprod test network version of the Minswap DEX as follows:

1. Go to <https://testnet-preprod.minswap.org/>
2. Connect your wallet by clicking 'Connect Wallet', then choosing 'Lace' (or a different wallet if you are not using Lace).
3. The wallet will popup asking you to confirm that you want to connect your wallet to the minswap.org site. By doing so we'll be able to use our wallet with the minswap.org site and interact directly with the blockchain. So, we'll press "Authorize". You can then choose whether you want to always allow the site to connect to your wallet, or whether it can only connect this time. Once your wallet is connected, go back to the Minswap main screen by pressing the 'X' as depicted in [Figure 17](#).

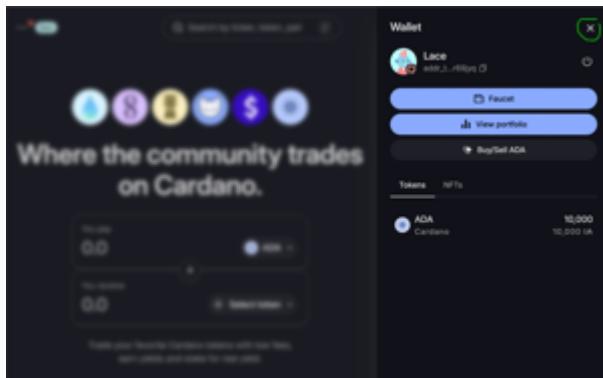


Figure 17. Closing Minswap's side-bar.

4. Click on the 'Trade' link in the top left (depicted in [Figure 18](#)) so that we're sent to the 'swap' functionality.

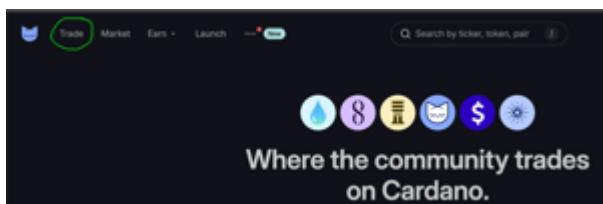


Figure 18. Click the 'Trade' link.

5. The DApp is likely automatically loaded with details to swap from Ada

(which you should have in your wallet) to Min (Minswap's own token). The testnet version of Minswap only supports swapping between Ada and Min. When you use the mainnet's version though you can choose to swap to other tokens as well. Enter an amount of Ada that you will swap in from your wallet, and the amount of Min that will be swapped out will be displayed (Figure 19 depicts a swap of 123 test Ada to the relevant amount of test Min at the time of writing).

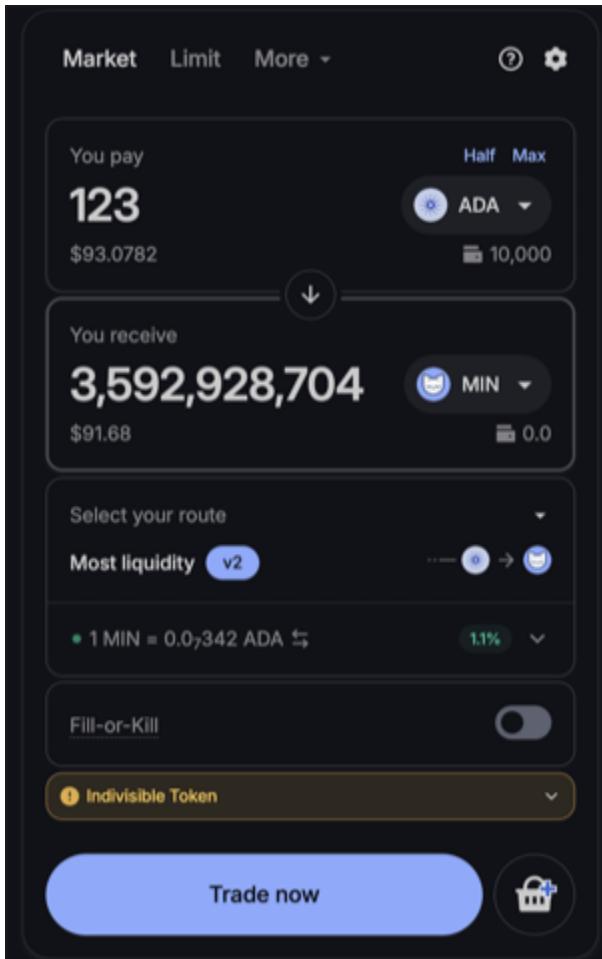


Figure 19. Review trade details.

6. You can then confirm the swap by clicking 'Trade now'. This should initiate your wallet to pop-up prompting you to choose whether you agree to the transaction as depicted in Figure 20.

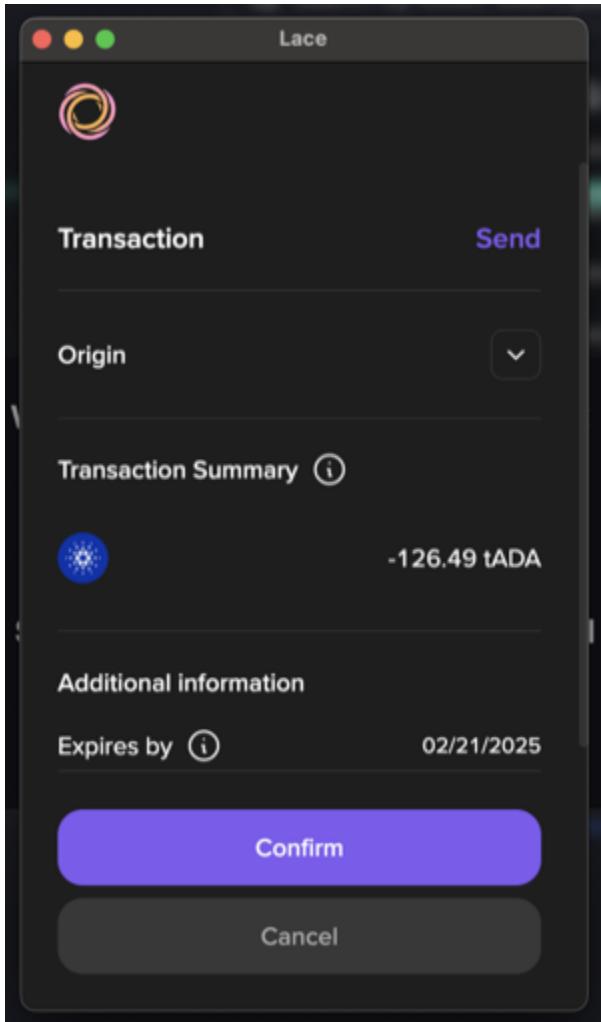


Figure 20. Confirm trade in Lace.

7. Once you confirm the transaction you may be required to enter the password you set for the wallet.
8. You should then see that the transaction was signed by your wallet as depicted in [Figure 21](#).

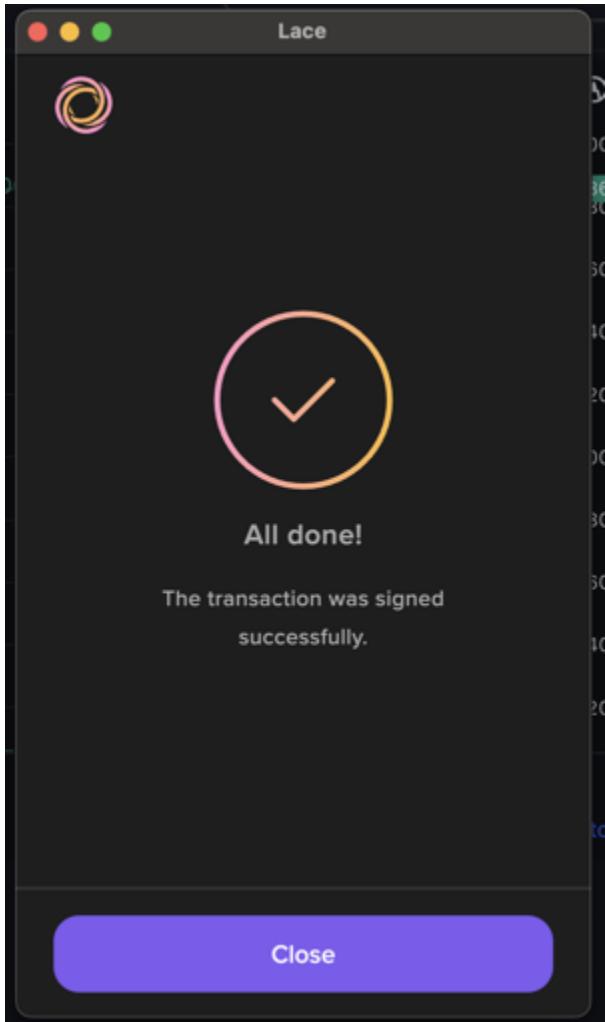


Figure 21. Transaction signed and submitted.

9. Once the transaction is confirmed on the blockchain, and the Minswap interface updates, you should see your balance of Min has increased (and Ada decreased) as depicted in [Figure 22](#).

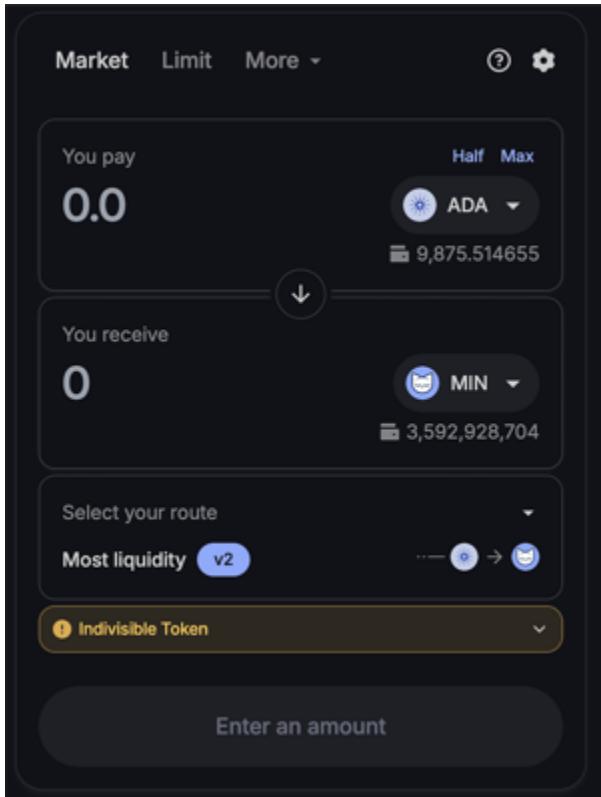


Figure 22. Balances updated in Minswap.

And that's it! You have used your first DApp (if you haven't already done so, of course). To further build on what was discussed in the previous section, it is important to highlight the different interactions that took place from your browser and what it was communicating with. First we requested the DApp by visiting the web site (i.e. <https://testnet-preprod.minswap.org/>), through which your browser requested the web page (i.e. HTML and JavaScript and other images and media-assets) from the centralized Minswap server. We then instructed the DApp to connect to our wallet, and confirmed in the wallet that we agree to it connecting with the DApp. Our wallet runs on our computers and is the interface that we can trust to verify interactions with the underlying blockchain. The DApp fetches swap prices to display on screen by communicating with the centralized server directly—and though this may raise eyebrows in that the centralized server can manipulate prices, the guarantees with respect to actual swap prices used are provided through the final on-chain script call that is used to initiate the swap (discussed next). When the user agrees to the swap in the wallet popup, it is at this point that the wallet directly communicates with the on-chain script code (deployed on the blockchain),

within which the swap price is guaranteed to be the current price as defined by the on-chain logic. So, the guarantee provided to the user is that the swap will be performed at the current price (defined with decentralized on-chain script code)—irrespective of whether the centralized server reports a different price. This potential price discrepancy is why such DEXs allow for users to specify a 'slippage' amount and/or minimum/maximum prices for swaps—so that users can express what minimum/maximum swap price they agree to in the case that there is a discrepancy between the prices reported on screen (by the centralized server) and the actual current price that the swap would use. This discrepancy emerges not only from the fact that servers may report different prices, but also given that time passes between user acceptance and the time the actual swap would take place—and within this time it may be the case that other swaps were executed for the specific price-pair that would affect the swap price.

Having explored using a DApp, let's now delve into aspects of internal workings of a DApp by re-creating parts of a DApp.

26.2.3. Creating a DApp

We'll now create the following aspects of a DApp:

- Server-side code: A NodeJS server that will send a page's HTML /JavaScript to the end-user.
- Client-side code: This is the code that will be sent from the server (discussed above), but will execute in the client-side browser. This code will connect to the wallet and communicate with a deployed on-chain script.

We will not create on-chain script code in this section (since that is handled in [Chapter 8](#)). Indeed, DApps can be created that communicate with existing deployed on-chain scripts that may not necessarily be written by the same developers/teams—just as we demonstrate now

below.

Creating a Server (with NodeJS)

We now discuss creating a NodeJS server that will be used to serve content to requesting users. You can use any other framework to create server-side code if you wish (such as Python, PHP, .NET, Java, or any other framework you may prefer). We'll use NodeJS' express package. Follow these steps to create the server:

1. First, you need to ensure that NodeJS is installed, and that you can run 'node' and 'npm' from the command line.
2. Create a new directory where your server code will be saved. We'll refer to this as the 'server' directory.
3. In the server directory, run: **npm init**
and for ease of this tutorial, you can just keep all default settings.

This will create a package.json file that defines the settings of the NodeJS project. Verify that the 'main' setting is set to 'index.js'. This setting defines the main entry point file for code in the NodeJS project.

4. Create the 'index.js' file in the server directory.
5. The template code is provided below.

```
const express = require('express');
const app = express();
const port = 3000;

app.get('/', (req, res) => {
    res.sendFile(__dirname + '/index.html');
});

app.listen(port, () => {
    console.log(`Server is running at http://localhost:${port}`);
});
```

6. We are making use of the 'express' package, and therefore need to install it. You can do so by running the following command: **npm install express**
7. Create an HTML file that the server will send to the client. We'll call this index.html. For now, just put the text 'Hello World!' in index.html and save the file.
8. Thereafter you can run the server using the following command: **node index.js**
9. Open a browser, and go to the url: localhost:3000
You should see a page similar to [Figure 23](#)

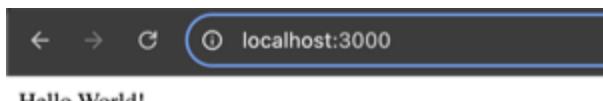


Figure 23. A first web server.

Creating the Client-Side Code to Connect to the Wallet

Now that we have a server able to send HTML/JavaScript to end-users, let's write the client-side code to connect to a user's wallet and interact with the underlying on-chain scripts. We'll only provide the bare minimal code that is needed. Indeed, you may want to look into implementing a full HTML page (including html, head and body tags), but we'll only provide the necessities for the sake of simplicity.

First, we'll create an HTML button and JavaScript that will connect the client-side code to the wallet. The code to provide a connect button is provided below.

```

<button id="connectWallet" onclick="connectWallet()">Connect Wallet
</button>

<script>
async function connectWallet() {
    if (window.cardano && window.cardano.lace) {
        try {
            let lace = await window.cardano.lace.enable();
            const walletAddresses = await lace.getUsedAddresses();
            console.log("Connected to Lace:", walletAddresses);
        } catch (error) {
            console.error("Error connecting to Lace Wallet:", error);
        }
    } else {
        console.error("Lace Wallet not found");
    }
}
</script>

```

After reloading the webpage (i.e. refreshing the url, localhost:3000), you should see the button on screen. If the code is correct, once you press the button, the Lace wallet should pop-up requesting the user to allow for the underlying client-side code to be able to connect to the Lace wallet as depicted in [Figure 24](#). Upon confirming that the DApp can connect to the wallet, we will not see any changes in the page, since we did not provide any code to do so. However, if you check the developer console in the browser you should see the output messages stating that we successfully managed to connect the wallet to the client-side JavaScript and also the addresses used.

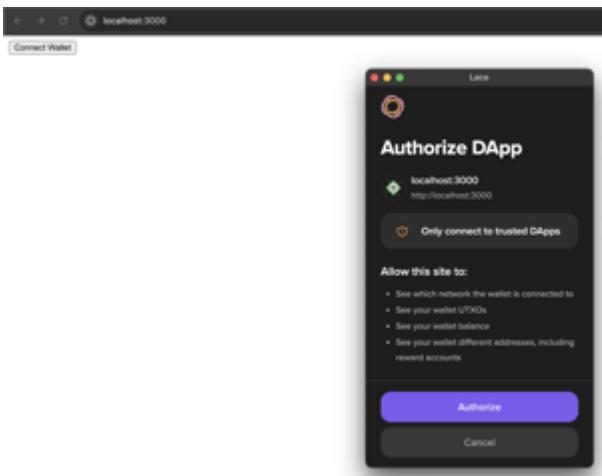


Figure 24. Connect the DApp to Lace.

Now that we have connected the client-side code to the wallet, we'll write some code that will interact with an on-chain script. Just before we do this though, we'll now package some libraries that we need to use in the client-side JavaScript.

Packaging Libraries for use in Client-Side JavaScript

In the client-side JavaScript code, we'll use Mesh—a library that will provide an easier-to-use interface to interact with the on-chain script code deployed on the blockchain. To do so, we'll package the Mesh library using webpack and serve it to the client-side JavaScript code. Indeed, you can use a different method to package and serve the library. The code we provide here may require changes (especially when considering different versions of SDKs used, e.g. NodeJS). If the code does not work out-of-the-box you may need to investigate how to package and deploy libraries and/or fix this code as required for your environment. We will not delve into the intricacies of this code but you may want to read up on how to package and serve libraries for client-side JavaScript code.

To export the Mesh library follow these steps:

1. Install webpack and webpack-cli by running:

```
npm install --save-dev webpack webpack-cli
```

2. Install @meshsdk/core, path-browserify, stream-browserify, crypto-browserify, buffer, and process by running:

```
npm install @meshsdk/core path-browserify stream-browserify crypto-browserify buffer process
```

3. In the NodeJS application, create the file ./mesh-entry.js with the following code:

```
import * as Mesh from '@meshsdk/core';
export {
  BrowserWallet,
  Transaction,
  resolvePlutusScriptAddress,
  applyCborEncoding,
  MeshTxBuilder,
  BlockfrostProvider,
} from '@meshsdk/core';
```

4. Create the ./webpack.config.js file with the following code:

```
const path = require('path');
const webpack = require('webpack');

module.exports = {
  entry: './mesh-entry.js',
  mode: 'production',
  output: {
    filename: 'mesh.bundle.js',
    path: path.resolve(__dirname, 'public/js'),
    library: 'Mesh',
    libraryTarget: 'window',
  },
  experiments: {
    topLevelAwait: true,
  },
  resolve: {
    fallback: {
      fs: false,
      path: require.resolve('path-browserify'),
    },
  },
}
```

```
        stream: require.resolve('stream-browserify'),
        crypto: require.resolve('crypto-browserify'),
        buffer: require.resolve('buffer/'),
        process: require.resolve('process'),
    },
},
plugins: [
    new webpack.ProvidePlugin({
        Buffer: ['buffer', 'Buffer'],
        process: 'process',
    }),
],
};
};
```

5. Run webpack to generate the bundled Mesh library:

```
npx webpack
```

6. If successful, the bundled client-side JavaScript code will be generated at the following path: ./public/js/mesh.bundle.js
7. The NodeJS ./index.js application should be updated to allow for the bundled Mesh library to be served to clients by adding the following line:

```
app.use(express.static(__dirname + '/public'));
```

The full updated ./index.js code follows:

```
const express = require('express');

const app = express();
const port = 3000;

app.use(express.static(__dirname + '/public')); //added now

app.get('/', (req, res) => {
    res.sendFile(__dirname + '/index.html');
});
```

```
app.listen(port, () => {
  console.log(`Server is running at http://localhost:${port}`);
});
```

Using the Bundled Mesh Library in the Client-Side JavaScript

Now, we'll use the bundled mesh library in the client-side JavaScript to communicate with on-chain script.

We'll expand on the HTML file described above (from the [Creating Client-Side Code](#) Section). Again, for simplicity sake we'll encode all HTML and JavaScript into a single file (in index.html). We'll start by adding the boilerplate functionality to use the bundled library:

1. Import the bundled library:

```
<script src="js/mesh.bundle.js"></script>
```

2. In the script tag, we'll get references to the objects and functions needed:

```
<script>
const { BrowserWallet,
        Transaction,
        resolvePlutusScriptAddress,
        applyCborEncoding,
        MeshTxBuilder,
        BlockfrostProvider,
      } = window.Mesh;
```

3. The full updated index.html should look like this:

```
<button id="connectWallet" onclick="connectWallet()">Connect Wallet
</button>

<script src="js/mesh.bundle.js"></script>

<script>
const { BrowserWallet, //added now
```

```

        Transaction, //added now
        resolvePlutusScriptAddress, //added now
        applyCborEncoding, //added now
        MeshTxBuilder, //added now
        BlockfrostProvider, //added now
    } = window.Mesh; //added now

    async function connectWallet() {
        if (window.cardano && window.cardano.lace) {
            try {
                let lace = await window.cardano.lace.enable();
                const walletAddresses = await lace.getUsedAddresses();
                console.log("Connected to Lace:", walletAddresses);
            } catch (error) {
                console.error("Error connecting to Lace Wallet:", error);
            }
        } else {
            console.error("Lace Wallet not found");
        }
    }
</script>

```

- To test this code, the Node server will need to be started (potentially restarted), and the page loaded by opening the url `localhost:3000` in a browser. Then check to make sure that loading of the library and loading of the Mesh library objects and functions do not raise any errors (though you might see an error relating to not being able to load favicon.ico).

26.2.4. Interacting with the Redeemer 42 On-Chain Script Code

To demonstrate DApp interaction, we'll write client-side JavaScript code to interact with the Redeemer 42 on-chain script code (discussed in [Chapter 8](#)).^[5] You can read Section [Simple validation scripts](#) to get a better understanding of the Redeemer 42 Script (if you have not already done so). We'll send funds, deploy a reference script and then claim back the funds sent.

The Redeemer 42's reference script that the DApp will interact with has

already been deployed to the preprod network. Its transaction hash is:
ac43f379762d68839a75d95146c332e6025e5a305fffc071308d138849109bfc

Sending Funds to the Redeemer 42 On-chain Scripts

To add functionality that sends funds to the Redeemer 42 on-chain script code follow these steps:

1. First, we'll add some variable definitions at the top of the script tag:

```
<script>
let wallet;
let walletAddress;

let txHashAssetUtxo;
```

2. Then, we'll modify the `connectWallet` function to get a reference to the wallet that we can use with the `BrowserWallet` class imported as follows:

```
async function connectWallet() {
  if (window.cardano && window.cardano.lace) {
    try {
      let lace = await window.cardano.lace.enable();
      wallet = await BrowserWallet.enable('lace'); //changed now
      walletAddress = await wallet.getChangeAddress(); //added now
      console.log("Connected to Lace:", walletAddress); //changed now
    } catch (error) {
      console.error("Error connecting to Lace Wallet:", error);
    }
  } else {
    console.error("Lace Wallet not found");
  }
}
```

3. Add into the client-side JavaScript code the following to get a reference to the deployed Redeemer 42 script:

```
const redeemer42Script = {
  code: applyCborEncoding(
```

```

"581e010100255333573466e1d2054375a6ae84d5d11aab9e3754002229308b01"),
  version: "V3"
};

const redeemer42Addr = resolvePlutusScriptAddress(redeemer42Script, 0);

```

4. Then to actually send funds we'll use the following code (that is explained below the code):

```

1 async function sendFunds(amount) {
2   console.log(`Sending funds: ${amount}`);
3   const tx = new Transaction({ initiator: wallet })
4     .setNetwork("preprod")
5     .sendLovelace({ address: redeemer42Addr }, amount)
6     .setChangeAddress(walletAddress);
7
8   console.log('Building transaction... ');
9   const txUnsigned = await tx.build();
10  console.log('Transaction built... Signing transaction... ');
11  const txSigned = await wallet.signTx(txUnsigned);
12  console.log('Transaction signed... Submitting transaction... ');
13  txHashAssetUtxo = await wallet.submitTx(txSigned);
14  console.log(`Transaction submitted... Asset UTXO hash:
$ {txHashAssetUtxo}`);
15 }

```

Line numbers 3-6 sets the required parameters for the transaction including: passing in a reference to the wallet we're using to send funds, the network (i.e. preprod), the script address and the amount of Lovelace to send, and the change address.

In line number 9, 11 and 13, we build the transaction, sign it and submit the transaction respectively.

5. We also add a 'Send Funds' button to call the added functionality to send 3,000,000 Lovelace (3 Ada).

For reference, the full updated index.html file follows:

```
<button id="connectWallet" onclick="connectWallet()">Connect Wallet
```

```

</button>
<button id="sendFunds" onclick="sendFunds('3000000')">Send Funds</button>
<!-- added now -->

<script src="js/mesh.bundle.js"></script>

<script>
const { BrowserWallet,
        Transaction,
        resolvePlutusScriptAddress,
        applyCborEncoding,
        MeshTxBuilder,
        BlockfrostProvider,
} = window.Mesh;

const redeemer42Script = { //added now
    code: applyCborEncoding(
"581e010100255333573466e1d2054375a6ae84d5d11aab9e3754002229308b01"),
    version: "V3"
}
const redeemer42Addr = resolvePlutusScriptAddress(redeemer42Script, 0);
//added now

let wallet; //added now
let walletAddress; //added now

let txHashAssetUtxo; //added now

async function sendFunds(amount) { //added now
    console.log(`Sending funds: ${amount}`);
    const tx = new Transaction({ initiator: wallet })
        .setNetwork("preprod")
        .sendLovelace({ address: redeemer42Addr }, amount)
        .setChangeAddress(walletAddress);

    console.log('Building transaction... ');
    const txUnsigned = await tx.build();
    console.log('Transaction built... Signing transaction... ');
    const txSigned = await wallet.signTx(txUnsigned);
    console.log('Transaction signed... Submitting transaction... ');
    txHashAssetUtxo = await wallet.submitTx(txSigned);
    console.log(`Transaction submitted... Asset UTXO hash:
${txHashAssetUtxo}`);
}


```

```

async function connectWallet() {
    if (window.cardano && window.cardano.lace) {
        try {
            let lace = await window.cardano.lace.enable();
            wallet = await BrowserWallet.enable('lace'); //changed now
            walletAddress = await wallet.getChangeAddress(); //added now
            console.log("Connected to Lace:", walletAddress); //changed
            now
        } catch (error) {
            console.error("Error connecting to Lace Wallet:", error);
        }
    } else {
        console.error("Lace Wallet not found");
    }
}
</script>

```

After running the NodeJS server and refreshing the page (i.e. refreshing localhost:3000), you should see the added button 'Send Funds':

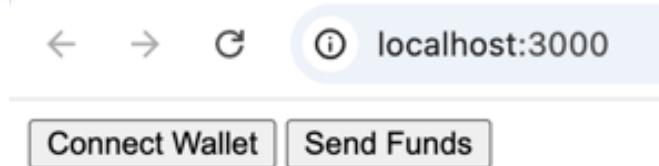


Figure 25. Added 'Send Funds' button.

Upon clicking 'Send Funds' the wallet should pop-up asking that you confirm to sending 3 Ada. It may take a while until the transaction is part of a block—you can check your wallet's transaction history and also search in a Cardano preprod network block explorer for your wallet's address for a successful transaction (at the associated date/time).

Deploying a reference script for the Redeemer 42 example

To add functionality that deploys a reference script (discussed in [Chapter 8](#)) for the Redeemer 42 example follow these steps:

1. We will make use of an RPC provider, which allows for querying of parameters from the blockchain. We'll make use of the BlockfrostProvider provided by mesh SDK, but you could also look into using other providers:

```
const provider = new BlockfrostProvider('<YOUR API KEY>');
```

2. Then we can include the burn address where to associate the reference script to:

```
const burnAddr =
'addr_test1wr4mrzsjwa6pquu0m6480mq06kpxsht80d4nfh56dcak6lsejdm28';
```

3. We add a function that will handle deploying the reference script as follows:

```
1 async function deployRefScript() {
2     console.log('Deploying reference script');
3     const txBuilder = new MeshTxBuilder({
4         fetcher: provider
5     });
6
7     console.log('Getting Wallet UTXOs... ');
8     const utxos = await wallet.getUtxos();
9     console.log(`Retrieved [${utxos.length}] Wallet UTXOs`);
10    console.log('Building reference script transaction... ');
11    const unsignedTx = await txBuilder
12        .txOut(burnAddr, [{ unit: "lovelace", quantity: '3000000' }])
13        .txOutReferenceScript(redeemer42Script.code,
14            redeemer42Script.version)
15        .changeAddress(walletAddress)
16        .selectUtxosFrom(utxos)
17        .complete();
18
19    console.log('Signing transaction... ');
20    const signedTx = await wallet.signTx(unsignedTx);
21    console.log('Transaction signed... submitting transaction... ');
22    txHashRefUtxo = await wallet.submitTx(signedTx);
23    console.log(`Transaction submitted... Reference Script UTXO hash:
```

```
    ${txHashRefUtxo} );
23 }
```

In Line 3-5, we create an instance of a transaction builder that we will use in a few lines.

Line 8 gets the wallet's UTXOs (unspent transactions) that could be used (for the transaction that will be submitted).

Lines 11-16 builds the transaction to deploy the reference script, and then the transaction is signed and submitted (on lines 19 and 21).

4. Finally, we'll add a button to test out the deploy reference script functionality:

```
<button id="deployRefScript" onclick="deployRefScript()">Deploy Reference Script</button>
```

The whole code should now look something like this:

```
1 <button id="connectWallet" onclick="connectWallet()">Connect Wallet
  </button>
2 <button id="sendFunds" onclick="sendFunds('3000000')">Send Funds
  </button>
3 <button id="deployRefScript" onclick="deployRefScript()">Deploy Reference Script</button> <!-- added now -->
4
5 <script src="js/mesh.bundle.js"></script>
6
7 <script>
8 const { BrowserWallet,
9     Transaction,
10    resolvePlutusScriptAddress,
11    applyCborEncoding,
12    MeshTxBuilder,
13    BlockfrostProvider,
14 } = window.Mesh;
15
16 const redeemer42Script = {
17   code: applyCborEncoding(
```

```

    "581e010100255333573466e1d2054375a6ae84d5d11aab9e3754002229308b01") ,
18   version: "V3"
19 }
20 const redeemer42Addr = resolvePlutusScriptAddress(redeemer42Script,
21   0);
21
22 const provider = new BlockfrostProvider('<YOUR API KEY>'); //added
23   now
24
25 const burnAddr =
26   'addr_test1wr4mrzsjwa6pquu0m6480mq06kpxsht80d4nfh56dcak6lsejdm28';
27   //added now
28
29 let wallet;
30 let walletAddress;
31
32 let txHashAssetUtxo;
33
34 async function deployRefScript() { //added now
35   console.log('Deploying reference script');
36   const txBuilder = new MeshTxBuilder({
37     fetcher: provider
38   });
39
40   console.log('Getting Wallet UTXOs... ');
41   const utxos = await wallet.getUtxos();
42   console.log(`Retrieved ${utxos.length} Wallet UTXOs`);
43   console.log('Building reference script transaction... ');
44   const unsignedTx = await txBuilder
45     .txOut(burnAddr, [{ unit: "lovelace", quantity: '3000000' }])
46     .txOutReferenceScript(redeemer42Script.code,
47       redeemer42Script.version)
48     .changeAddress(walletAddress)
49     .selectUtxosFrom(utxos)
50     .complete();
51
52   console.log('Signing transaction... ');
53   const signedTx = await wallet.signTx(unsignedTx);
54   console.log('Transaction signed... submitting transaction... ');
55   txHashRefUtxo = await wallet.submitTx(signedTx);
56   console.log(`Transaction submitted... Reference Script UTXO hash:
57   ${txHashRefUtxo}`);
58 }

```

```

55 async function sendFunds(amount) {
56     console.log(`Sending funds: ${amount}`);
57     const tx = new Transaction({ initiator: wallet })
58         .setNetwork("preprod")
59         .sendLovelace({ address: redeemer42Addr }, amount)
60         .setChangeAddress(walletAddress);
61
62     console.log('Building transaction... ');
63     const txUnsigned = await tx.build();
64     console.log('Transaction built... Signing transaction... ');
65     const txSigned = await wallet.signTx(txUnsigned);
66     console.log('Transaction signed... Submitting transaction... ');
67     txHashAssetUtxo = await wallet.submitTx(txSigned);
68     console.log(`Transaction submitted... Asset UTXO hash:
69     ${txHashAssetUtxo}`);
70 }
71
72 async function connectWallet() {
73     if (window.cardano && window.cardano.lace) {
74         try {
75             lace = await window.cardano.lace.enable();
76             wallet = await BrowserWallet.enable('lace');
77             walletAddress = await wallet.getChangeAddress();
78             console.log("Connected to Lace:", walletAddress);
79         } catch (error) {
80             console.error("Error connecting to Lace Wallet:", error);
81         }
82     } else {
83         console.error("Lace Wallet not found");
84     }
85 </script>

```

Indeed, if this DApp were to be deployed by the developer, they may facilitate the deployment of the reference script—and not require the user to actively choose to deploy the reference script via the interface.

After re-running the NodeJS server and refreshing the page (i.e. refreshing localhost:3000), you should see the added button 'Deploy Reference Script':



Figure 26. Added 'Deploy Reference Script' button.

Claiming back funds from the Redeemer 42 example

To claim back funds follow these steps:

1. We add a button to initiate claiming back of the funds:

```
<button id="claimFunds" onclick="claimFunds()">Claim Funds</button>
```

2. We add a function to help retrieve back UTXOs that we'll make reference to when initiating the transaction to claim back funds, as follows:

```
async function getUtxo(scriptAddress, txHash) {
  const utxos = await provider.fetchAddressUTxOs(scriptAddress);
  if (utxos.length == 0) {
    throw 'No listing found.';
  }
  let filteredUtxo = utxos.find((utxo) => {
    return utxo.input.txHash == txHash;
  });
  return filteredUtxo;
}
```

4. Finally, we add the functionality to initiate the transaction to claim back funds, as follows:

```
1 async function claimFunds() {
2     console.log('Claiming funds');
3     console.log(`Retrieving Asset UTXO [${txHashAssetUtxo}] from [
4 ${redeemer42Addr}]`);
5     const assetUtxo = await getUtxo(redeemer42Addr, txHashAssetUtxo);
6     console.log(`Retrieving Script UTXO [${txHashRefUtxo}] from [
7 ${redeemer42Addr}`);
```

```

    `${burnAddr}]`));
6   const refScriptUtxo = await getUtxo(burnAddr, txHashRefUtxo);
7   const redeemer = { data: BigInt(42) };
8
9   console.log('Find collateral UTXO');
10  const walletUtxos = await wallet.getUtxos();
11  const collateral = walletUtxos.find(utxo => utxo.output.amount
12    .find(asset => asset.unit === "lovelace" && BigInt(asset.quantity) >=
13      BigInt(5000000)));
14
15  console.log('Building claim funds transaction...');

16  const tx = new Transaction({ initiator: wallet, fetcher: provider
17    })
18    .setNetwork("preprod")
19    .redeemValue({ value: assetUtxo,
20      script: refScriptUtxo,
21      datum: undefined,
22      redeemer: redeemer})
23    .sendValue(walletAddress, assetUtxo)
24    .setCollateral([collateral])
25    .setRequiredSigners([walletAddress]);
26  const txUnsigned = await tx.build();

27
28  console.log('Signing transaction...');

29  const txSigned = await wallet.signTx(txUnsigned, true);
30
31  console.log(`Transaction signed... submitting transaction...`);

32  const txHash = await wallet.submitTx(txSigned);
33  console.log(`Transaction submitted... Claim Funds hash: ${txHash}
34 `);
35 }

```

In Line 4, we retrieve back the initial asset UTXO in which we sent funds.

In Line 6, we retrieve back the reference script UTXO.

In Line 7, we define the redeemer value (42) to send to the script.

In Lines 10 and 11, we find a UTXO that can be used as collateral. Though this step may be automatically undertaken for us in non-browser environments, we need to explicitly determine the collateral to be used when using a browser interface like BrowserWallet.

In Lines 14-23, we build the transaction, then in Line 26 we sign the transaction, and submit the transaction in Line 29.

The whole updated code follows:

```
1 <button id="connectWallet" onclick="connectWallet()">Connect Wallet
2 </button>
3 <button id="sendFunds" onclick="sendFunds('3000000')">Send Funds
4 </button>
5 <button id="deployRefScript" onclick="deployRefScript()>Deploy
6 Reference Script</button>
7 <button id="claimFunds" onclick="claimFunds()>Claim Funds</button>
8 <!-- added now -->
9
10 <script src="js/mesh.bundle.js"></script>
11
12 <script>
13 const { BrowserWallet,
14     Transaction,
15     resolvePlutusScriptAddress,
16     applyCborEncoding,
17     MeshTxBuilder,
18     BlockfrostProvider,
19 } = window.Mesh;
20
21 const redeemer42Script = {
22   code: applyCborEncoding(
23     "581e010100255333573466e1d2054375a6ae84d5d11aab9e3754002229308b01"),
24   version: "V3"
25 }
26 const redeemer42Addr = resolvePlutusScriptAddress(redeemer42Script,
27   0);
28
29 const provider = new BlockfrostProvider('<ENTER API KEY>');
30
31 const burnAddr =
32   'addr_test1wr4mrzsjwa6pquu0m6480mq06kpxsht80d4nfh56dcak6lsejdm28';
33
34 let wallet;
35 let walletAddress;
36
37 let txHashAssetUtxo;
```

```

31
32 async function getUtxo(scriptAddress, txHash) { //added now
33     const utxos = await provider.fetchAddressUTxOs(scriptAddress);
34     if (utxos.length == 0) {
35         throw 'No listing found.';
36     }
37     let filteredUtxo = utxos.find((utxo) => {
38         return utxo.input.txHash == txHash;
39     });
40     return filteredUtxo;
41 }
42
43 async function claimFunds() { //added now
44     console.log('Claiming funds');
45     console.log(`Retrieving Asset UTXO [${txHashAssetUtxo}] from [
46         ${redeemer42Addr}]`);
46     const assetUtxo = await getUtxo(redeemer42Addr,
47         txHashAssetUtxo);
47     console.log(`Retrieving Script UTXO [${txHashRefUtxo}] from [
48         ${burnAddr}]`);
48     const refScriptUtxo = await getUtxo(burnAddr, txHashRefUtxo);
49     const redeemer = { data: BigInt(42) };
50
51     console.log('Find collateral UTXO');
52     const walletUtxos = await wallet.getUtxos();
53     const collateral = walletUtxos.find(utxo => utxo.output.amount
54         .find(asset => asset.unit === "lovelace" && BigInt(asset.quantity)
55             >= BigInt(5000000)));
56
56     console.log('Building claim funds transaction... ');
57     const tx = new Transaction({ initiator: wallet, fetcher:
58         provider })
59         .setNetwork("preprod")
60         .redeemValue({ value: assetUtxo,
61             script: refScriptUtxo,
62             datum: undefined,
63             redeemer: redeemer})
64         .sendValue(walletAddress, assetUtxo)
65         .setCollateral([collateral])
66         .setRequiredSigners([walletAddress]);
65     const txUnsigned = await tx.build();
66
67     console.log('Signing transaction... ');
68     const txSigned = await wallet.signTx(txUnsigned, true);

```

```

69
70     console.log(`Transaction signed... submitting transaction...`);
71     const txHash = await wallet.submitTx(txSigned);
72     console.log(`Transaction submitted... Claim Funds hash: ${txHash}
73 `);
74
75 async function deployRefScript() {
76     console.log('Deploying reference script');
77     const txBuilder = new MeshTxBuilder({
78         fetcher: provider
79     });
80
81     console.log('Getting Wallet UTXOs... ');
82     const utxos = await wallet.getUtxos();
83     console.log(`Retrieved [${utxos.length}] Wallet UTXOs`);
84     console.log('Building reference script transaction... ');
85     const unsignedTx = await txBuilder
86         .txOut(burnAddr, [{ unit: "lovelace", quantity: '3000000'
87     }])
88         .txOutReferenceScript(redeemer42Script.code,
89         redeemer42Script.version)
90         .changeAddress(walletAddress)
91         .selectUtxosFrom(utxos)
92         .complete();
93
94     console.log('Signing transaction... ');
95     const signedTx = await wallet.signTx(unsignedTx);
96     console.log(`Transaction signed... submitting transaction...`);
97     txHashRefUtxo = await wallet.submitTx(signedTx);
98     console.log(`Transaction submitted... Reference Script UTXO hash:
99 ${txHashRefUtxo}`);
100 }
101
102
103
104
105
106
107
108

```

```

109     const txSigned = await wallet.signTx(txUnsigned);
110     console.log(`Transaction signed... Submitting transaction...`);
111     txHashAssetUtxo = await wallet.submitTx(txSigned);
112     console.log(`Transaction submitted... Asset UTXO hash:
113     ${txHashAssetUtxo}`);
114
115 async function connectWallet() {
116     if (window.cardano && window.cardano.lace) {
117         try {
118             let lace = await window.cardano.lace.enable();
119             wallet = await BrowserWallet.enable('lace');
120             walletAddress = await wallet.getChangeAddress();
121             console.log("Connected to Lace:", walletAddress);
122         } catch (error) {
123             console.error("Error connecting to Lace Wallet:", error);
124         }
125     } else {
126         console.error("Lace Wallet not found");
127     }
128 }
129 </script>

```

After re-running the NodeJS server and refreshing the page (i.e. refreshing localhost:3000), you should see the added button 'Claim Funds':

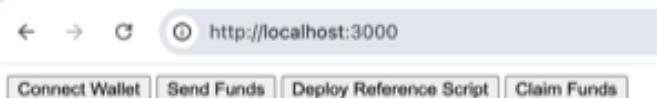


Figure 27. Added 'Claim Funds' button.

The above provides the full DApp implementation. To run through the example DApp you should first ensure your wallet is connected by pressing 'Connect Wallet'. Thereafter, pressing the 'Send Funds' button should send a transaction within which you send funds to the Redeemer 42 script. Once confirming the transaction in your wallet it is ideal to wait to see that the transaction is confirmed. Thereafter, you can press the 'Deploy Reference Script' button to the deploy the reference script—again,

after confirming the transaction in your wallet you should wait to see that the transaction is confirmed. Finally, you can claim back funds by pressing the 'Claim Funds' button—and yet again after confirming the transaction in your wallet you should wait to see that the transaction is confirmed. In the final transaction, you should see that funds were received into your wallet (i.e. the initial funds sent into and locked in the redeemer script were sent back to you).

Indeed, the DApp is barebones, and serves the purpose to demonstrate of how we to interact from client-side JavaScript with Cardano on-chain scripts.

26.3. Decentralized Web Storage

The DApp architecture introduced in Section [Decentralized Application Architecture](#) relies on a traditional centralized web server to deliver the initial HTML/JS web-page content, and then provides guarantees to users through the interaction with on-chain scripts (smart contracts). Yet, relying on a centralized web server to deliver the initial HTML/JS web-content may not be suitable for certain applications and/or it may be desirable that some web content is not dependent on a centralized web server.

Different solutions have been proposed for decentralized web storage (including IPFS, Arweave, Filecoin, Storj, and others) that vary in cost, persistence, latency and reliability—we therefore suggest that readers interested to make use of decentralized storage to explore different alternatives. Yet one common feature of decentralized web storage is that the resources (e.g. html pages, images, etc) are 'content-addressable'—i.e. the resource's unique identifier directly represents the content. Typically, the hash of the resource's data is used as the unique identifier to refer to the specific resource. Content addressable unique identifiers/references can be thought of as providing a system that allows for resources to be retrieved based on the actual content itself, rather than where it is located

(i.e. a filename on a specific server). Using such a system for web resources:

- guarantees the integrity of resources (since the hash of the content must match the unique identifier—that can always be checked);
- minimises data storage requirements for resources with same content;
- allows for decentralizing from relying on a single specific server to host and serve the specific resource—any peer in the network that hosts the resource can serve it.

26.4. DApps and UI/UX Issues

While DApps promise to decentralize many multi-party digital services, without a doubt there are still several challenges that must be overcome for their mass-adoption—particularly for the non-tech-savvy. We now discuss some challenges (that DApps on all blockchains face) and potential future directions to overcome such challenges:

- **Wallet setup woes:** New users may find it daunting to install a wallet and store the wallet's seed phrase. Various directions to circumvent some of these issues have been proposed including "account abstraction", use of "ephemeral keys", and use of "passkeys".
- **Switching between user interfaces:** Users may find it hard to deal with switching between DApp web pages, wallet pop-ups, and block explorers. That being said, confirming actions in wallets is akin to how users confirm online purchases with internet banking apps. As wallets become more integrated into browsers and mobiles, and as wallets provide users with information that is more digestible (without having to use a block explorer), user experience should also reach similar levels to internet banking apps.
- **Gas costs:** for non-cryptocurrency related DApps, especially those that a user may not interact with often, users may find it troublesome to both purchase and cover required gas costs. Solutions to this may

include feeless/gasless transaction models.

- **Smart contract/on-chain script code and errors may be opaque:** Even though on-chain code is available for all to see, and some may be able to viewed in an intuitive visual block format, understanding code logic is often beyond what many non-tech users are capable of. Furthermore, errors that are often displayed to users require technical knowledge to understand. Different avenues are being investigated that may eventually help non-tech users to understand both the scripts they are confirming to interact with and any errors reported.
- **Losing access to keys:** Whilst not exactly a barrier to using DApps, some users may not feel comfortable using DApps associated with a private key/seed phrase that they may lose. Different approaches are being investigated to circumvent users taking full responsibility on keeping their keys safe including: social recovery, shared wallets, and multi-sig wallets.

[1] The content likely also makes use of CSS code, but this detail can be ignored unless you want to dig deeper into web page design.

[2] Most web browsers allow users to use 'Developer Tools' that are built into the web browsers themselves, that allow you to manipulate web pages once they are in your browser.

[3] It may help to consider that when using a web site you are the client, and this is why it is referred to as client-side code, since the code is executing on your laptop. Really though the terminology comes from ;client-server' architectures (which has resemblances to the analogy provided).

[4] Faucets are the term typically used for services that send test cryptocurrency.

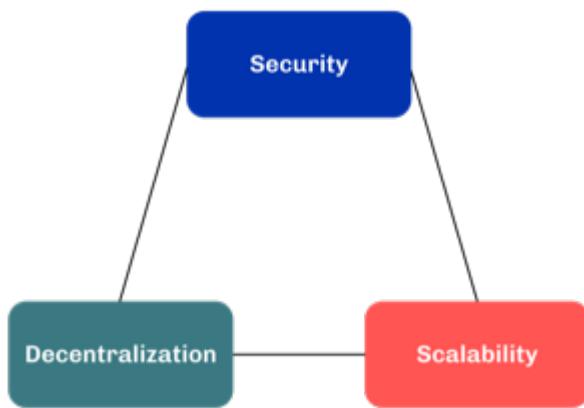
[5] Also see github.com/LukaKurnjek/ppp-plutusV3-plinth/blob/main/off-chain/meshjs/Week02/redeemer42-ref-script.ts

Chapter 27. Looking forward

27.1. A Scaling Vision

27.1.1. The Blockchain Trilemma

Imagine, a robust and decentralized network with thousands of interconnected nodes participating in forming consensus on an ever-evolving ledger-state. While this describes the Cardano network, it (like many decentralized or distributed networks before) finds itself constrained at the hands of the Blockchain Trilemma.



The Blockchain Trilemma is the Iron Triangle principle applied to decentralized networks. This concept generally asserts that any of the three intertwined principles may not be changed or affected without affecting the others in some way. In the world of blockchain, the three principles comprising this triangle are Security, Scalability, and Decentralization.

- **Security:** In simple terms, this refers to the ability of the blockchain to resist bad actors. Increased security typically requires a higher level of decentralization, but may have a negative effect on scalability.

- **Scalability:** This refers to the ability of a decentralized network to increase overall throughput. This principle is usually at direct odds with decentralization, as the more the consensus or other actions of the protocol are distributed, the longer things take to happen in a secure manner.
- **Decentralization:** The distribution of function away from the control of a single or several actors. The more decentralized a system is, the more secure, especially with well-spread geographic distributions and secure consensus mechanisms. (This is an involved topic on its own and significant efforts exist to measure decentralization for cryptocurrencies).

While Cardano arguably has robust and efficient consensus at its base layer (Layer 1 in other words), in order to truly become the financial backbone of the world economy, all possible scaling options must be explored.

27.1.2. Scaling

There are two fundamental ways in which decentralized systems typically scale: **Vertical** and **Horizontal**.

Vertical Scaling

Vertical scaling refers to the increase in the performance of a system by adding resources or modifying, streamlining, and adjusting the existing system. Cardano accomplishes this through a variety of on-chain methods:

- **Block Size Increases:** The larger the block, the more transactions it carries.
- **Pipelining:** Also called diffusion pipelining, is a modification to the consensus layer that allows for faster block propagation by allowing nodes to pre-notify their downstream peers of an incoming block and allowing those nodes to pre-fetch the new block body.

- **Plutus Upgrades:** General plutus performance improvements including reference inputs, plutus datums and script sharing.
- **Cardano Node Enhancements:** Continual improvement to the node, such as lowering memory consumption and the spreading out of staking reward computations across each epoch allow the network to more easily scale long term.
- **Input Endorsers:** Improves block propagation time by allowing transactions to be separated into pre-constructed blocks, improving propagation efficiency and allowing for higher transaction throughput.

It is worth mentioning that increasing hardware requirements is necessary for some vertical scaling methods, such as increasing block size. This comes with tradeoffs of course, the cost can be lessened with node and plutus performance enhancements, allowing for even greater possibilities of scaling.

Horizontal Scaling

Horizontal scaling refers to increasing the performance and potential throughput of a decentralized network by adding systems that operate in parallel to the main network with the purpose of processing transactions and computations via external methods. Unlike vertical scaling, horizontal scaling is only limited by the methods introduced, how they work, and how widely they are implemented. These solutions are referred to as Layer 2 solutions. Here are some examples:

- **Payment Channels:** This type of protocol typically allows two parties to transact with each other through an off-chain protocol. Bitcoin's Lightning Network is a collection of payment channels where payments may be routed between channels, creating a useful and fast payment network that offloads transaction volume from the main chain.
- **State Channels:** A state channel is a generalization of a payment channel that extends the capabilities beyond payments to include script

execution and complex logic handling all off-chain.

- **Side Chains:** A side chain essentially allows the transferring of assets “off” of the main chain to a separate chain with its own protocol and consensus rules. This is usually accomplished by locking assets on the Layer 1 with smart contracts and creating an alternative on the side chain.
- **Rollups:** Rollups typically provide a way to move transaction processing and state off-chain while still representing the off-loaded transactions on-chain.
 - **Optimistic:** Verification of the rolled up transactions are performed on-chain by independent validators. Validation disputes may occur and must be resolved on-chain.
 - **Validity (Zero Knowledge):** Proofs are generated off-chain alongside the processing of transactions and are used on-chain to verify correct computation

27.2. Enter Hydra

Hydra is a family of state channel scaling solutions on Cardano that provides the network and applications running on it with the ability to do significant amounts of work off-chain in a secure manner.

One of the methods utilized in pursuit of the Hydra vision is the use of isomorphic state channels, which was brought to the public in IOG's 2021 paper *Hydra: Fast Isomorphic State Channels*. State channels allow multiple parties to take a small chunk of the ledger from the main chain, work on it out-of-band, agree on the changes and bring that chunk of changed ledger state to the main chain. State channels operate in a similar manner to Bitcoin's Lightning network, but instead of leveraging the full feature set of EUTxO, the Lightning Network is relegated to payments only, limited by Bitcoin's UTXO accounting model. Lightning payment channels only allow two parties to connect to a single channel, but with the

implementation of HTLCs (Hash Time-Locked Contracts) it is possible to route payments between payment channels.

Hydra's implementation of state channels on the other hand, offers the ability for multiple parties to open isomorphic state channels, granting those parties use of the full EUTxO feature-set, which includes smart contracts, script execution, and other expressive functions as well as simple payment transactions. The isomorphic nature of these state channels allows for nearly frictionless transfer between the main ledger and the state channels mini-ledger, since both can use the same level of expressiveness.

It is important to note that the use of state channels is one of a number of possible scaling methodologies that can be used. For example, the Hydra Tail protocol will likely use a form of either optimistic or zero-knowledge rollups instead of isomorphic state channels.

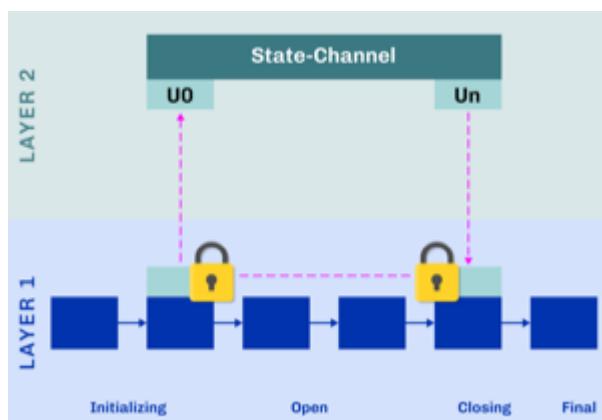
27.2.1. Hydra Head

Being the first implementation of the Hydra protocol, Hydra Head bears strong resemblance to the idea of fast isomorphic state channels presented in the original Hydra paper. Translational research led to the development of the [Hydra HeadV1 Specification](#) and the Hydra Head protocol, which is open source and can be found, used, built and developed on its [github repository](#).

This protocol allows multiple users to agree on a set of rules, open a Hydra head with, import UTxOs into the head, transact, execute scripts, or do other work off the main chain, close the head, finalize the state and bring it back to the main chain. The opening and closing of a Hydra head each take a transaction, but there are no real limits to the amount of transactions that may take place in the Hydra head once opened. It is also important to note that though the mini-ledger used by a Hydra head follows the same ledger rules as the main chain, Hydra heads do not use Ouroboros consensus, and instead use an optimistic consensus model in

which all clients must agree and update state with every transaction. The protocol contains built-in safeguards that allow any participant to close the head at any time. The total consensus algorithm requires all participants to remain online and reachable for the head to remain open. If one becomes unreachable, the Head will close and the state will be returned to the main chain.

The advantage of leveraging EUTxO to securely transact offline without using Ouroboros consensus is that the decentralized consensus no longer has to be paid for the same way, meaning that the parties using a Hydra head can agree on transaction fees that fit what they are doing, even potentially setting them to zero. Without requiring global consensus it is possible to transact at zero or low fees, and the constraints on throughput are significantly less restrictive if a low number of clients is assumed.

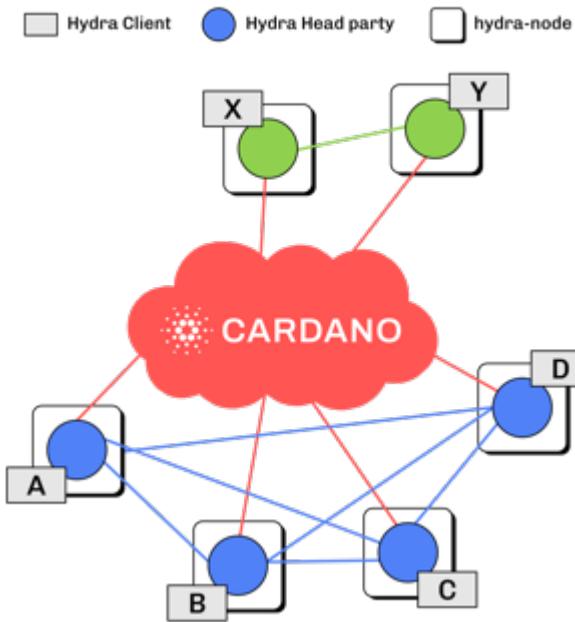


The Hydra Head protocol is one part of a much bigger idea in regards to scaling with state channels on Cardano. A wide variety of Hydra implementations are possible and will be implemented based on the use-case and need, and serve as parts of a greater vision where Hydra allows Cardano and many of its applications to scale tremendously.

Here are some potential future topologies.

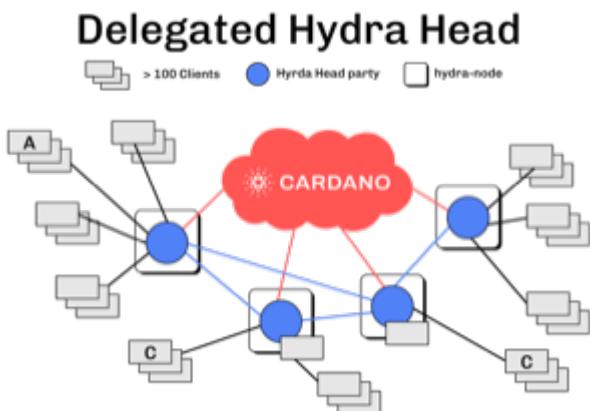
27.2.2. Basic Hydra Head Network

Basic Hydra Head



The idea of the basic Hydra Head network involves multiple hydra nodes connected together to form a Hydra head state channel between the connected nodes. The figure above shows two separate Hydra head networks, the blue and the green, with two and four participants respectively. Eventually, it will be possible for multiple Hydra nodes to be opened on a single hydra node.

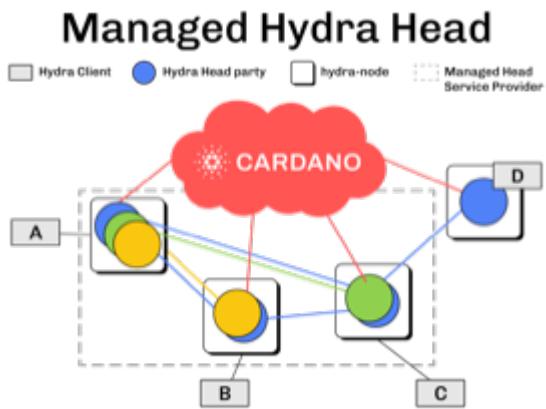
27.2.3. Delegated Hydra Head Network



The Delegated Hydra Head model involves a separation of the Operator and Client nodes. In this scenario, Operators possess the Hydra keys used by the protocol to sign snapshots on the Head as well as the keys used to progress the Head state machine on the main chain. The Clients will still be in possession of the payment keys associated with any of the client

UTxOs submitted to the head, but in this scenario it will be possible for hundreds or even thousands of clients to interact with a single state channel. A downside here is that the clients will have to trust at least a single operator, with the upside being many clients directly interacting with the state channel. This model could be best explained as running Hydra Head as a side-chain.

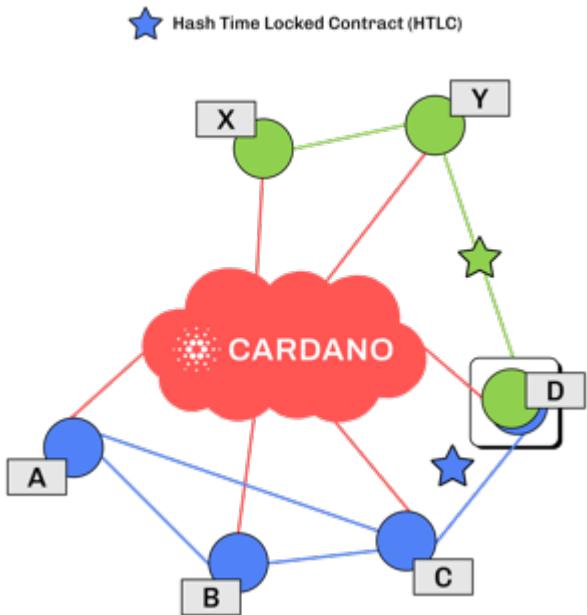
27.2.4. Managed Hydra Head Network



Rather than each Hydra Head requiring each participant to host an instance of Hydra node, Hydra nodes will be able to support multiple Heads per node. Here, a Managed Head Service Provider would host Hydra nodes as a service, allowing clients to connect to the Hydra Heads via API (Application Programming Interface) while clients still control the Hydra keys. The client in this case would be analogous to a “light node” that checks into the infrastructure maintainer, or in this case the Managed Head Service Provider.

27.2.5. Hydra Head Network

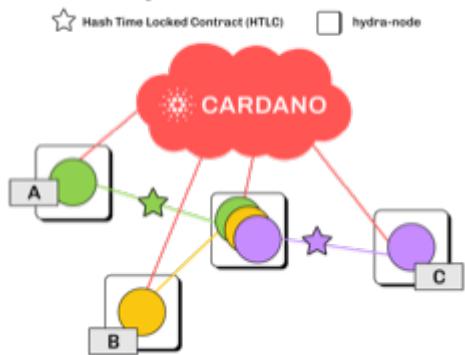
Hydra Head Network



The Hydra Head Network gives Basic Hydra Head Networks the ability to connect with each other through individual Hydra nodes. This will require Hydra nodes to be able to manage more than one Head per node, and HTLCs (Hash Time Locked Contracts) or adaptor signatures may be used to perform swaps between networks and forward payments of fungible assets. This is very similar to how the Lightning Network works on Bitcoin.

Hydra Head networks can also be assembled in arrangements resembling other network topologies.

Star-Shaped Head Network

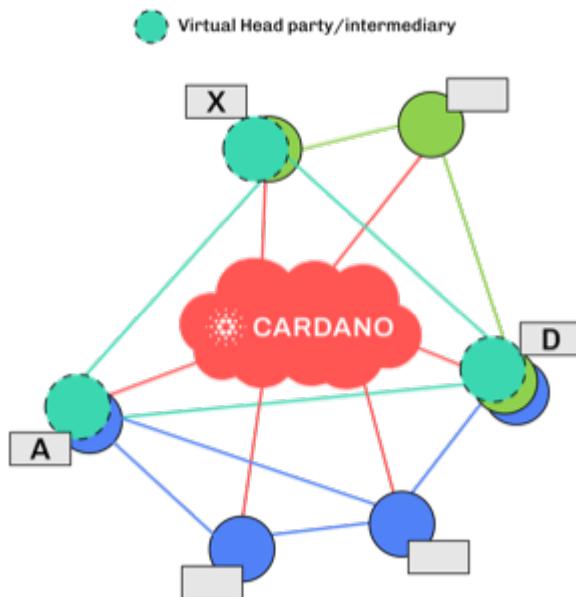


An example being the Star-Shaped Hydra Head Network. In this case, a central Hydra node server (the hydra-node in the diagram with the green, yellow, and purple circles inside of it) would act as a Hydra Head aggregation point in which Client nodes (A, B, and C) would open channels

with the central server independently. The Hydra Server could route transactions between state channels, and potentially use HTLCs in case the destination head is not currently active. The server node will need to have high operational uptime whereas the client nodes can attempt to connect as needed.

27.2.6. Inter-Head Hydra Network

Interhead Hydra

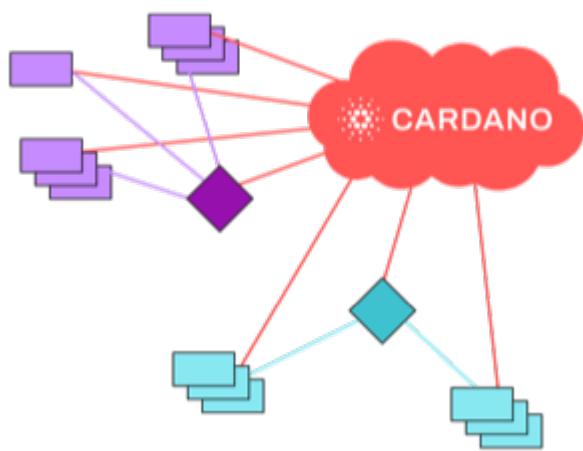


Though the topologies listed so far will significantly help Cardano and many of its applications scale in a variety of ways, the concept of Interhead Hydra (Layer 3) takes it a step further and considers virtual Hydra Heads running on top of regular Hydra Heads allowing even more potential for out-of-band computation and consensus.

27.2.7. Hydra Tail

Hydra Tail

Tail Client Tail Server

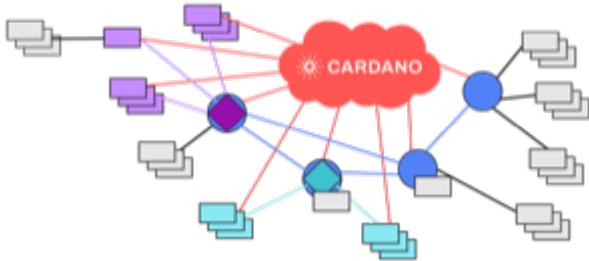


The concept of the Hydra Tail does not use the Hydra Head protocol at all. Instead, it uses a server-client architecture, where the server remains online and maintains a ledger state waiting for client connections. Unlike the Hydra Head protocol, the Hydra Tail protocol is asymmetrical, with the Tail Server assuming most responsibilities. This asymmetry allows Tail Clients to be low-powered and unreliable (i.e. smartphones, personal computers, etc.) and can connect to the Tail Server when needed. There are methods to prevent the server from bad behavior such as putting collateral on the main chain and a challenge-response-protocol on the mainchain where clients may dispute server claims. The Hydra Tail protocol may also use zero knowledge proofs alongside optimistic consensus in the form of rollups.

27.2.8. Combined Hydra Head and Tail Network

Combined Head/Tail

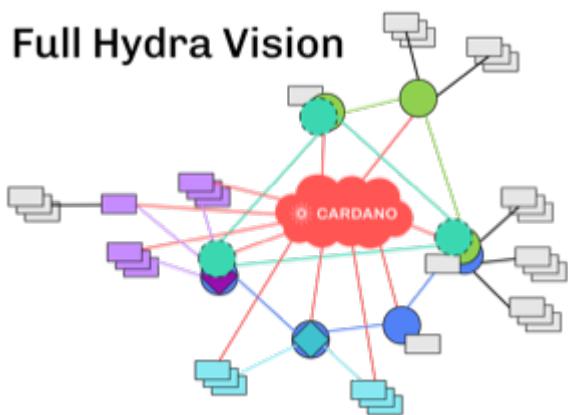
Tail Client Tail Server



Another potential way to accommodate high-throughput application

demands is the combining of the Hydra Head and Hydra Tail protocols. Hydra head networks could be accessed through Tail Servers for clients that are unable to run a reliable hydra node for example, with Hydra nodes potentially serving as the Tail servers themselves.

27.2.9. Full Hydra Vision



The concept of Hydra as a scaling solution goes well beyond a multi-party state channel that leverages the expressive EUTxO feature set to offload transaction volume from the mainchain and increase throughput for applications and services. It will eventually be possible to see Hydra evolve into a fully heterogenous Layer 2 solution that fits a multitude of use-cases. Once developed it will be possible to observe multiple large, interconnected and incredibly high-throughput Hydra networks with the ability to connect to each other, all running on top of Cardano.

27.3. Mithril

27.3.1. Mithril in a nutshell

Mithril is a protocol and a network designed for proof-of-stake blockchains that aims to provide lightweight access to blockchain data with high security and decentralization (see the [Mithril security \(advanced\)](#) section). While the Cardano blockchain offers robust security, starting a new node, syncing it with the network, or exchanging data can be slow and resource intensive – requiring 24GB of RAM, 150GB (and growing) of storage, and over 24 hours for initial synchronization. Mithril,

developed for Cardano as part of the Basho development phase, focuses on optimization, scalability, and interoperability. Practical use cases for Mithril include data synchronization for light and full-node wallets and data exchange with layer 2 solutions (such as bridges, sidechains, rollups, and state channels).

The Mithril protocol design is based on the following core ideas:

- Leveraging stake to efficiently certify data within a large committee of signers
- Ensuring a transparent setup without increasing trust requirements. Security is guaranteed through a trusted Genesis phase for Mithril, similar to Cardano's.

Mithril achieves its goals using a stake-based threshold multi-signature (STM) scheme developed by the Input | Output (IO) research and engineering teams and presented in the [Mithril research paper](#). Simply put, the scheme generates a certificate to prove that a minimum representative portion of signers – selected based on their stake – has signed a specific piece of data (here referred to as the message).

Unlike other schemes where communication complexity increases exponentially with the number of signers, Mithril employs a lottery mechanism to select a subset of signers. The probability of selection is proportional to the stakeholders' stake, ensuring efficient participation. The selected signers produce signatures that an aggregator combines into a Mithril multi-signature, which can only be produced when a predefined threshold (quorum) of the total stake has contributed. No interaction is required between signers, and the communication complexity is sublinear. A verifier can then efficiently check the multi-signature without relying on the aggregator that provides the multi-signature.

The Mithril network is the practical implementation of the Mithril protocol. Network nodes take the aforementioned roles: *signer*,

aggregator, and *verifier* (also known as the client), as well as other security-related roles such as the relay.

27.3.2. The Mithril protocol (advanced)

Mithril enables a multi-party signing process by holding a pre-defined number of lotteries (**m**) run by all signers when signing a message. A multi-signature is generated once the number of unique winning lottery indices from the signers reaches a threshold (**k**) – the quorum. The probability of a signer winning a lottery is proportional to their stake. It is influenced by a tuning parameter (**f**), which represents the success probability assigned to the maximum stake weight or the total protocol stake. Note that this process can lead to a single signer winning multiple lottery indices for the same message.

The Mithril protocol has three phases:

- **Protocol establishment**, during which the protocol parameters for all the Mithril participants (signers and aggregators) are selected
- **Initialization**, during which the set of participating Mithril signers generate and broadcast keys
- **Operation**, during which the set of participating Mithril signers create and broadcast individual message signatures, and Mithril aggregators produce message multi-signatures.

Protocol establishment phase

In this phase, three important protocol parameters are selected:

- **m**, the number of lotteries a single signer can participate in to sign a message
- **k**, the minimum required number of unique lottery indices gathered from individual signatures to create a multi-signature
- **f (or phi_f)**, a tuning parameter that adjusts the probability of signers winning a lottery based on their stake.

The protocol's security directly depends on the values of these parameters, which are carefully chosen to ensure a negligible chance of an adversary forging a valid multi-signature. Details on the relationship between these parameters and protocol security are provided in the [Mithril security \(advanced\)](#) section.

All network participants, including signers and aggregators, must operate using identical protocol parameters. This protocol establishment phase is conducted only once.

Initialization phase

This phase involves initializing each signer and registering their cryptographic keys. First, the signer retrieves the protocol parameters to generate a new key pair (verification key and signing key) along with a proof of possession (PoP). The PoP proves that the signer owns the secret key associated with the verification key being registered. The verification key and PoP are broadcast to all parties for registration, which takes place within a predefined period – one epoch (five days on Cardano). The initialization phase repeats at the start of each new epoch. The keys generated during this phase are used for signing two epochs later.

For the protocol to work, all signers and aggregators must reach consensus on the key registration process, which includes the verification keys and the associated stake of the signers (referred to as the Mithril stake distribution). Specifically, consensus must be achieved among a subset of signers representing sufficient stake in the protocol to ensure the quorum of signatures can be met.

Each party then computes a condensed representation of the key registration process: the **aggregate verification key** (AVK). The AVK, derived as the root of a [Merkle tree](#) created from the verification keys and stake of the registered signers, is used to create individual signatures and ultimately verify the aggregated multi-signature.

Operation phase

In the operation phase, signers generate individual signatures, aggregators combine individual signatures into multi-signatures, and verifiers confirm the validity of these multi-signatures.

During this phase, operations run in cycles triggered deterministically on each party by the computation of a beacon, which depends on the type of data to be signed. For example, a beacon might correspond to the transaction set or UTXO set of the chain, signed every 20 blocks. Using this beacon, each signer computes the associated message, which includes the aggregate verification key, and attempts to sign it.

To do so, signers participate in a series of lotteries (**m**) against the message to be signed and their stake (conducted in parallel), creating a list of winning indices. If this list is not empty, signers combine it with the cryptographic signature of the message (generated using their signing key) into the individual signature. This individual signature is then broadcast to the network for aggregators.

Upon receiving individual signatures from the signers, the aggregator combines them into a multi-signature, provided that the quorum is reached. The quorum is reached when the valid list of winning lottery unique indices, extracted from the individual signatures, meets the threshold (**k**). Such a multi-signature is then sealed into a Mithril certificate along with the AVK.

Finally, a verifier can confirm the certificate's validity by verifying its multi-signature against its AVK. This verification requires a single cryptographic pairing operation, which is fast.

The diagram below illustrates how a signer can create an individual signature with Mithril. You can learn more about this process in the [Mithril: a stronger and lighter blockchain for better efficiency](#) blog post.

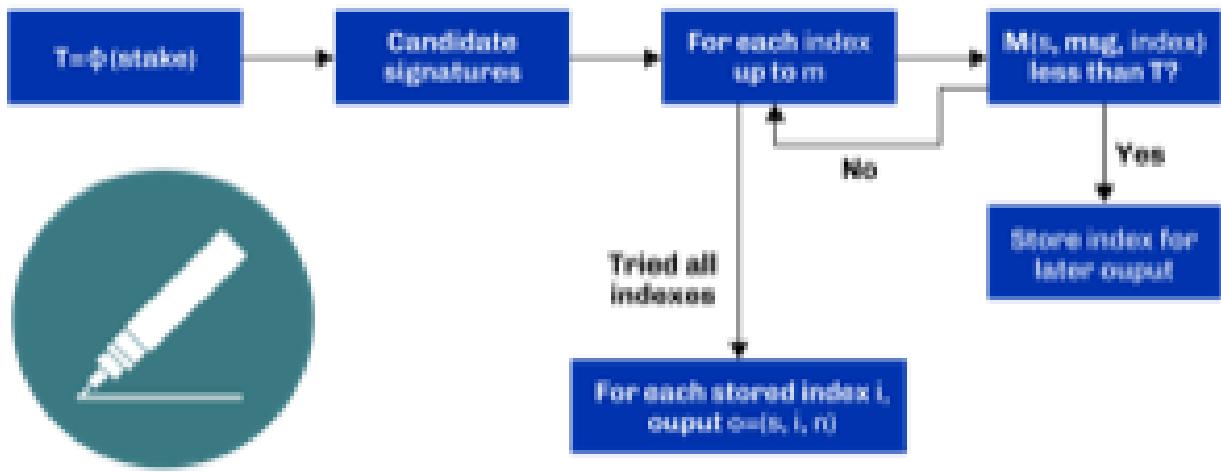


Figure 49. Mithril individual signature creation

27.3.3. The Mithril network

Nodes

The Mithril network consists of multiple nodes, each with different roles, to implement the signature and aggregation processes of the protocol:

- The **Mithril signer** is responsible for producing individual signatures
- The **Mithril aggregator** collects individual signatures from the **signers** and aggregates them into a multi-signature
- The **Mithril client**, both a library and a node, retrieves artifacts (eg a Merkle proof or an archive file) and verifies that they are genuinely signed by a Mithril multi-signature. The node version runs in a WASM-compatible browser or as a standalone binary with a command-line interface.

Certified data types and their use cases

Mithril nodes are jointly able to sign and create Mithril multi-signatures for any information that can be computed deterministically by each of them individually (the nodes need to compute the same message to ensure successful aggregation). A specific framework has been implemented for Mithril nodes to support the creation of new types of certified data without impacting the core protocol code. This provides flexibility and enables swift implementation. Below is a list of data types currently

implemented:

- **Cardano database:** the internal database of the Cardano node is certified, enabling fast bootstrapping (a full node can be restored in 20 minutes!).
- **Cardano transactions:** the Cardano transactions set is certified, allowing for the certification of a subset of transactions. This allows light wallets to have a certification layer for transactions associated with an address, eliminating the need to trust a third-party provider or run a full Cardano node. The verification can even occur in the browser. This also supports the implementation of ‘light clients’ for bridges, enabling to monitor transactions on layer 1 smart contracts without running a full Cardano node. It is also applicable to state channels such as Hydra or rollups.
- **Cardano stake distribution:** the Cardano stake distribution is certified at the transition to a new epoch. This enables the verification of validator node stake in a bridge without running a full Cardano node to access this data.
- **Mithril stake distribution:** the Mithril stake distribution (the stake of the stake pool operators (SPOs) involved in the protocol and their verification keys) is certified at the transition to a new Cardano epoch. This mandatory information must be signed at each epoch to secure the Mithril protocol, and is achieved through the **Mithril certificate chain**.

The certificate chain

The **Mithril certificate chain** is the component that certifies the **Mithril stake distribution** used to create multi-signatures. Its primary purpose is to prevent adversaries from executing an [eclipse attack](#).

Without the certificate chain, the stake distribution can't be trusted. A malicious actor could relatively easily create a fake stake distribution and use it to produce a valid multi-signature, which would be embedded in a

valid but non-genuine certificate. This certificate could be served by a dishonest Mithril aggregator node, leading an honest Mithril client to trust a non-genuine information.

To certify the Mithril stake distribution used to create a multi-signature, the distribution must be verified as having been previously signed in a certificate from the previous epoch. Then, one can recursively verify that the earlier certificate is valid in the same manner. The first certificate in the chain has a special role, which is discussed below. Also, the certificates are chained in such a way that traversing them results in only one certificate per epoch, enabling fast verification.

The first certificate in the certificate chain is known as the **genesis certificate**. Validating the stake distribution embedded in the genesis certificate is only possible by manually signing it with a private key linked to a widely accessible public key, called the **genesis verification key**. The use of these specific keys ensures the integrity and security of the initial stake distribution and subsequent transitions within the blockchain network. You can read more about the certificate chain design and its verification algorithm in the official documentation. A link is provided in the [Additional resources](#) section.

Decentralization and peer-to-peer (P2P) networking

Ultimately, the **Mithril network** is designed to be fully decentralized. However, the first implementation is centralized, as full decentralization introduces complex subjects that are currently active areas of research for the Mithril team:

- **Signer registration:** this protocol phase requires that a vast majority of the signers and aggregators compute the Mithril cryptographic operations on the same Mithril stake distribution. Currently, this is being achieved with a centralized broadcast mechanism of signer registrations by an aggregator. In a decentralized setup, an implementation of this process could be very similar to the consensus

reached by block producers in a blockchain over a P2P network. Implementing this is complex, which is why it is still under development.

- **Signature diffusion:** this protocol phase requires the diffusion of signatures from signers to aggregators over a network. This is currently achieved with a centralized mechanism on an aggregator. In a decentralized setup, the signature diffusion needs to rely on a P2P network layer. [CIP-137](#) (Decentralized message queue) has been proposed to leverage the Cardano network layer to operate the diffusion of signatures, incorporating new mini-protocols.

27.3.4. Mithril security (advanced)

Security of the cryptographic protocol

Mithril security relies on the underlying security of the STM scheme. The protocol leverages threshold multi-signatures, which enable the aggregation of multiple individual signatures into a single compact signature, depending on the distribution and control of stake among the participants. The Mithril protocol relies on the following:

- **Threshold multi-signature:** a scheme in which individual signatures from multiple participants are aggregated into a single signature if the total stake of the participants exceeds a certain threshold.
- **Stake-based eligibility:** the protocol ensures that only participants with sufficient stake are eligible to sign messages. This eligibility is determined [pseudorandomly](#).
- **Aggregation and verification:** individual signatures are aggregated into a single multi-signature, which can then be verified efficiently.

The Mithril protocol realizes the **ideal functionality of a stake-based threshold multi-signature scheme**. This means that an adversary cannot create a valid multi-signature unless they control a significant portion of the total stake.

The Mithril research paper mathematically demonstrates the protocol's security: it formalizes the security guarantee by showing that the protocol realizes the ideal functionality under specific conditions, relying on a computational hardness assumption and the collision resistance of a hash function. The proof involves a series of hybrid games comparing the real protocol with an idealized version to show that the adversary's probability of success is negligible. The security proof is supported by some lemmas:

- **Sampling property:** demonstrates that the probability of an adversary winning enough lotteries to form a multi-signature is negligible.
- **Individual signature verification failure:** ensures that the probability of a non-eligible user producing a valid signature is negligible.

The Mithril protocol is designed to be secure against a wide range of attacks, including those by adversaries with significant computational power, control a significant portion of stake, or attempts to manipulate the protocol through multiple identities or old stake. The security goals are as follows:

- **Integrity:** ensure that only valid and legitimate participants can generate multi-signatures, and that these multi-signatures accurately reflect the consensus of the stakeholders.
- **Resistance to Sybil attacks:** ensure that the influence in the protocol is proportional to the stake held, making it difficult for an adversary to gain control through multiple fake identities.
- **Forgery resistance:** prevent adversaries from creating valid forged individual signatures or multi-signatures.
- **Long-range attack resistance:** ensure that the old stake cannot be used to create an alternate certificate chain capable of overwriting the current one.

To ensure robust security and efficiency of the Mithril protocol, **protocol parameters** must be cautiously selected - the choice of these parameters

directly influences the trade-offs between security and efficiency:

- **Higher k and m values:** these increase security but require greater computational and communication resources. This is suitable for high-value transactions or networks with significant stake concentration.
- **Lower k and m values:** these optimize efficiency and are suitable for more distributed networks where the risk of attack is lower.

Security of the certificate chain

The **certificate chain** certifies the **stake distribution** used to create multi-signatures, embedding them in certificates that are chained together to establish trust. Its primary purpose is to prevent adversaries from executing an [eclipse attack](#). The verification process is recursive, and a chain is considered valid if, for each certificate in the chain (at least one certificate per epoch is required as the stake distribution changes at every epoch):

- The certificate itself is valid (the multi-signature is valid, and the certificate hash matches its content).
- The AVK representing the stake distribution used to create the multi-signature is either:
 - Signed by a Mithril multi-signature in a certificate from the previous epoch, which is referenced in the current certificate
 - Signed by a genesis signature. In this case, the certificate is called the **genesis certificate** – a one-time manual signature signed with a secret key. The corresponding verification key (the genesis key) is widely accessible to verifiers.

SPOs participation

Mithril security relies heavily on the participation level of SPOs and the adversarial assumptions of the underlying stake-based Cardano network. Achieving the protocol's full security requires participation representing

nearly the entire Cardano stake.

Threat model

Mithril security is also tied to its implementation, which is currently available in the [Mithril repository](#). The Mithril team has published a [threat model analysis](#), which assesses the assets involved in the Mithril network, identifies potential threats to these assets, and outlines their mitigations to better understand the impact on the SPO infrastructure and the Cardano chain.

27.3.5. The present and future of Mithril

The full implementation of the Mithril protocol is a work in progress, with new capabilities and enhancements being progressively added. The first version of the protocol was released on the Cardano mainnet in July 2023, with a set of pioneer stake pools as participating signers, enabling the certification of snapshots of the Cardano blockchain. Mithril currently operates in a centralized setting, where the aggregator, operated by the Mithril team at IO, takes additional responsibilities, such as broadcasting signing keys and orchestrating the signing process. Decentralization, increased stake pool participation, the addition of certificates for new types of data, and porting the Mithril client to new platforms and programming languages are some of the features on the development roadmap. Furthermore, while the current implementation targets the Cardano blockchain, it is designed to work in any stake-based environment beyond Cardano.

27.3.6. Additional resources

- [Mithril: Stake-based Threshold Multisignatures \(research paper\)](#)
- [Mithril repository](#)
- [Mithril documentation](#)
- [Mithril network architecture](#)

- Mithril protocol phases
- Certificate chain design
- Protocol security
- Bootstrap a Cardano node
- Run a Mithril signer node
- Threat model analysis
- Decentralized message queue CIP
- Mithril Explorer.

Index

A

Algorithmic stablecoins, 422
Amaru, 434
arbitrage traders, 429
automated auditing, 426

C

Cardano testnet, 451, 451
CIP-52, 441
CIP-57, 441
CIP-68, 422
client-side, 466, 468
Coindesk, 425
compiler, 434, 436, 436, 445
concurrency, 428
CSS, 491

D

DApps, 455, 456, 466, 490, 490, 490, 490, 491, 491
Decentralized Application Architecture, 455
Decentralized Applications, 452, 455
decentralized exchange, 461
determinism, 428, 428, 431
DEX, 461

distributed systems, 427

E

Emurgo, 297, 425, 432
ERC-20 tokens, 432
EUTXO, 433
Evolution-SDK, 439

F

Faustus, 445, 445
Formal methods, 437
Formal verification, 438, 440, 440
front-running invasion attacks, 430, 431

G

GeniusYield, 438, 439
GHC core, 436

H

HTML, 453, 456, 457, 466, 468

I

IPFS, 457, 489
Isabelle proof assistant, 447

L

lambda calculus, 435

liquidity providers, 430, 430

M

manual auditing, 426

Marlowe CLI, 446, 449

Marlowe Playground, 312, 450

Marlowe Runtime, 448, 449, 449,
449, 449

MBO), 434

mempool, 430, 431

Minswap, 461, 461, 461, 462, 462,
464, 465, 465

MLabs, 438, 439, 439, 442

N

network layer, 436

NFTs, 422

NodeJS, 466, 467, 467, 467, 467, 467,
471, 472, 478, 482, 488

O

order book model, 430, 430

Ouroboros, 427, 427

P

PIR, 436

Plutus Core, 434, 436, 436, 436, 439

R

Redeemer 42, 474, 474, 474, 474,
475, 475, 475, 478, 478, 483, 488

reentrancy attack, 431, 431

Rekt leaderboard, 425

S

script information, 370

server-side, 454, 454, 455, 455, 455,
467

smart contract auditor, 433, 433

SPOs, 427

static analysis, 447, 450

System F, 435, 435

T

Tether), 429

The EUTXO Handbook, 433

transaction output reference, 375

Turing-complete, 443, 443, 443

Tweag, 439, 439, 441, 443, 443, 447

U

UTXO contention, 442

V

vesting script, 403

W

web applications, 452, 455, 455, 456

web server, 452, 453, 453, 453, 453,
454, 454, 454

Web3, 425, 425, 429

webpack, 470, 470, 470, 471, 472

Well-Typed, 439