# The Concrete Architecture of Bitcoin Core


## Authors

Devin Pereira, Kanchan Shrestha, Alex Susilo, Adlai Bridson-Boyczuk, Cyrus Fung, Isaiah Wuthrich



## Abstract

In this report, we perform an in-depth examination of the concrete architecture of Bitcoin Core. The derived concrete architecture generally adheres to the layered style combined with peer-to-peer elements as outlined in our conceptual architecture; however, there were several major deviations that we analyze with the software reflection framework. The software reflection framework was also used to examine divergences between the conceptual architecture of Bitcoin Core's Mining subsystem with its concrete implementation. We also outline the derivation process for Bitcoin Core's concrete architecture and use our revised understanding of the architecture to re-examine the two use cases presented in our previous report.


## Introduction

As the reference implementation of Bitcoin, one of the most widely used and influential cryptocurrencies in the world, understanding Bitcoin Core at both a high and low level is essential to building a clear picture of how Bitcoin operates. This report focuses on the concrete architecture of Bitcoin Core, while comparing and contrasting it to the conceptual architecture that we derived in our previous report. While the overall architectural style remained unchanged, examination of the concrete architecture revealed some significant deviations from the conceptual architecture. One of these changes is the introduction of the Util subsystem, which contains functionality shared by multiple subsystems and tests. Another added component is App, which is responsible for most of Bitcoin Core's client-side functionality (such as its CLI and GUI) and now contains Node as a subsystem, which implements the core functionality for a single instance of Bitcoin Core. In general, our concrete architecture contains many more interactions that were not accounted for in our conceptual architecture, which were all analyzed using the software reflection framework.

To further our understanding of Bitcoin Core, we created a conceptual and concrete architecture for the Miner subsystem. Our conceptual architecture followed a pipe-and-filter architectural style with 5 subcomponents: Block Validation, Block Propagation, Block Solver, Block Creation and Transaction Selection. The concrete architecture generally followed the same structure and had the same amount of subcomponents, but with additional dependencies and a different naming scheme, where the subcomponents were called Miner, Merkleblock, Proof of Work, Consensus and Policy. Some of these derived subcomponents correspond to more than one subcomponent in the conceptual architecture.

We again applied the software reflection framework to examine these divergences and explain our reasoning.

Using the newfound understanding of both Bitcoin as a whole and the mining subsystem that software reflection gave us, we revisit the two use cases that were outlined in our original report: a Bitcoin transaction between two users, and a miner node solving a block. Finally, we finish off the report with a summary of our key findings, as well as a section on the limitations of our methodology and lessons learned from this report.

## Derivation Process

The derivation process included the use of Understand and the individual analysis of key files as well as researching documentation. When using Understand to first create our concrete architecture, it was necessary to map all files to their corresponding subsystems found in the conceptual architecture. For many of the files, file and folder names were sufficient to adequately assign them to a subsystem. Although, some files needed to be thoroughly analyzed to determine their function and relation to other systems to be accurately assigned a subsystem. Once all files were assigned, the dependencies were analyzed, gaps between the conceptual and the concrete architecture were found and then the appropriate changes were made. This was an iterative process, and the two steps of analyzing and adjusting were repeated many times as we refined our conceptual and concrete architecture. Some subsystems were merged with others and new subsystems were created as outlier files were found. Many of these outliers were grouped into the Util subsystem, which serves as a library of files that all other subsystems depend on. Once all files were assigned to appropriate subsystems, we reached a concrete architecture that accurately conveys the nature of Bitcoin Core.

# High Level Architecture

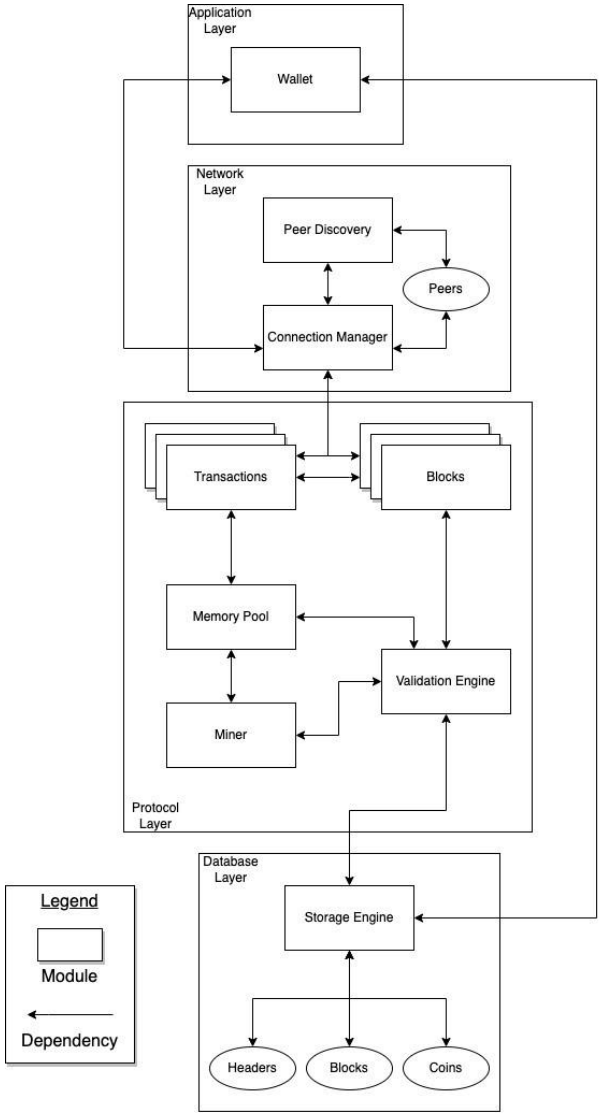


**Figure 1.** The conceptual architecture of Bitcoin Core

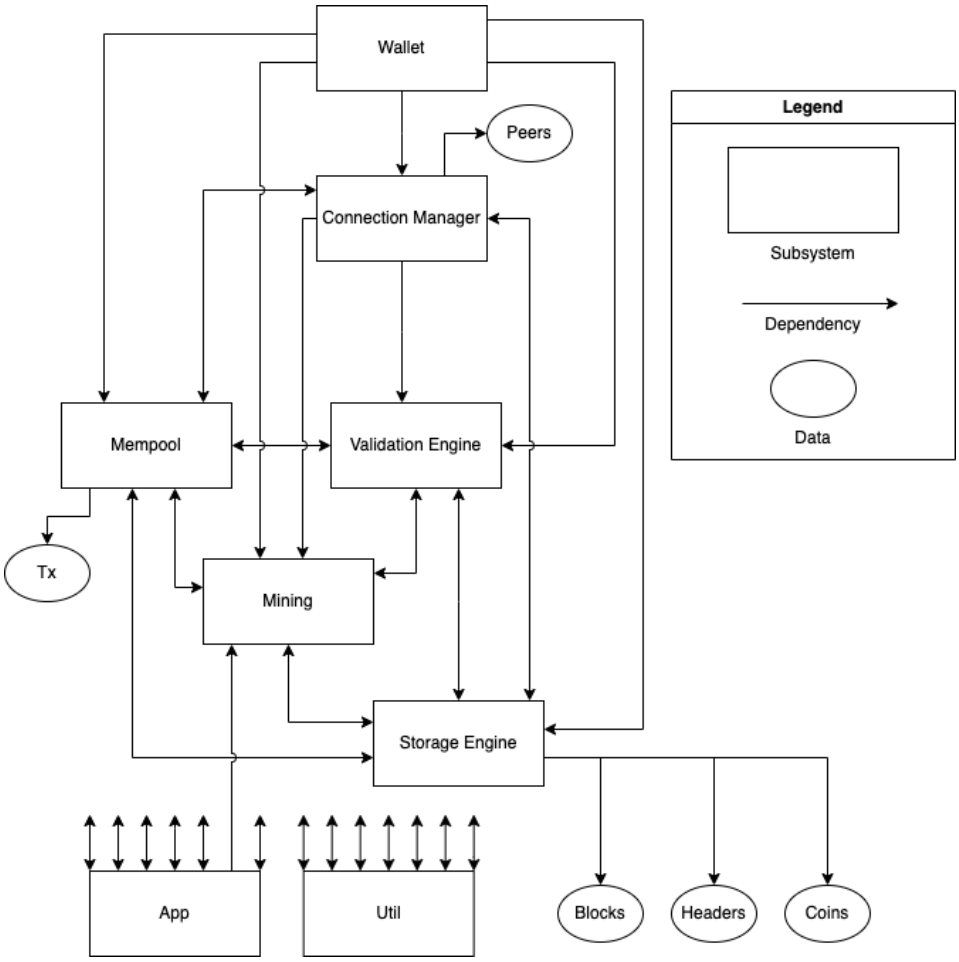


**Figure 2.** The concrete architecture of Bitcoin Core

# Description

## Mining

The Mining component is still responsible for adding blocks to the blockchain and creating new bitcoins. It receives transaction data and verifies their validity. Valid transactions are received from the Mempool component. Mining is also in charge of solving the PoW problem, then creating and transmitting the newest block in the blockchain.

The newly minted block consists of all the verified transactions picked from Mempool as well as the PoW. Once the new block is emitted from Mining to eventually make its way across the network, the mining process begins anew. Naturally, the miner is rewarded for their effort. When the other miners receive the new block, they will compare the block with their mempool, removing transactions that were included in the new block. They too start to work on mining with their own candidate block, a block without a valid PoW.

## Wallet

The Wallet component controls access to a user’s money, tracks balance, manages keys and addresses, and allows users to create and sign transactions. It stores the public and private keys used to access and spend a user's Bitcoin balance. Users can sign transactions with the keys, and thus prove they own the output of their transactions.

Additionally, Wallet interacts with the Connection Manager to synchronize the user's transaction history with the rest of the Bitcoin network. It employs the consensus rules to ensure that all transactions in the wallet are valid according to the rules of the Bitcoin protocol. Finally, the Wallet component is linked to the App component so developers can build custom applications that can interact with a user's Bitcoin wallet.

## Util

Util, a component which was missing in our conceptual architecture, provides shared functionality across different parts of the Bitcoin Core software. It includes a variety of utility functions for handling data serialization, network communication, cryptographic operations, and other core functionality. By consolidating this shared functionality into a single component, code duplication is reduced, and maintainability is improved. All other components depend on the functions defined in Util.

## Validation Engine

The Validation Engine subsystem continues to certify received transactions and blocks, ensuring that the bitcoin protocol is adhered to. In this module, many of the files are related to the aforementioned certifying, which it works on with the miner. Once the transactions are verified, they are sent to the mempool. The validation engine depends on the storage engine to supply the blocks of the blockchain.

## Storage Engine

The Storage Engine remains the primary component responsible for storing all data such as the current blockchain, the UTXO database and Wallet information on the disk. The module mainly consists of files relating to LevelDB and information such as the blocks and transaction indices. The Storage Engine relies on the Validation Engine for updating the stored block chain and many of the other subsystems depend on the storage engine as they use the data that it stores.

## Mempool

The Mempool subsystem is still responsible for storing all the validated transactions that are pending confirmation. Most of the files in this module are related to the management and storage of recently received transactions. This module depends on the Validation Engine for newly verified transactions and provides the Miner with the transactions used to form new blocks.

## App

The App sub-system encompasses the core application logic and functionality of the software. It serves as the main entry point for the Bitcoin Core software and provides the primary control flow that manages all the various sub-systems, such as the Wallet, Validation Engine, and Connection Manager.

One of the key features of the App sub-system is its modular design. This allows for the inclusion of pluggable features and extensions, such as support for alternative consensus rules, wallets, and scripting languages. This modular design also makes it easier to develop and maintain the software over time, as new features and enhancements can be added without affecting the existing codebase.

Another important aspect of the App sub-system is its UI components. These components provide users with a way to interact with the software through graphical and command-line interfaces. This allows users to monitor the status of their transactions, manage their Bitcoin wallets, and more.

## Connection Manager

The Connection Manager sub-system plays a critical role in the Bitcoin network by ensuring that all nodes are connected and able to communicate with each other. It establishes and maintains connections with other peer nodes using protocols such as the Bitcoin Protocol and the P2P Protocol. It also manages the transmission and receipt of network messages between nodes, ensuring that all nodes are kept up to date with the latest information.

Another key feature of the sub-system is its ability to handle synchronization of the Bitcoin blockchain across nides. Using the Block Propagation Protocol, it is ensured that all nodes have the same copy of the blockchain and that all transactions are valid according to the Consensus Rules. Additionally, the Connection Manager is responsible for managing the bandwidth usage of the Bitcoin Core software by prioritizing network traffic and limiting the number of established connections at any given time. This allows the software to remain efficient and responsible even under heavy network loads.

# Reflexion Analysis

## Mining

The salient change to the dependencies involving the Mining component was the addition of a new bi-directional connection with Storage Engine. There are many reasons for the dependency: stored blocks must contain information about the PoW defined in the Mining component, miners must have knowledge of the block structure described in Storage Engine, etc. Also, Mining employs many functions defined in the Util component to complete its operations giving rise to another dependency.

## Wallet

There were a significant number of changes in the links to and from the Wallet component. The forecasted bi-directional connections with Connection Manager and Storage Engine were found to be unidirectional, emanating from Wallet. While the Wallet component is required to contact the Connection Manager to initiate a transaction with another peer, the connection manager does not

require the balance of the wallet or any other data. That information is included in the blockchain stored by the Storage Engine component. Hence, the Storage Engine has no reason to depend on the Wallet component either.

## Util

Our view is that the dependencies with all other components should be unidirectional, emanating from Util. However, Util also depends on all other components to facilitate the common functionality. We view these dependencies as a flaw with the system architecture: a utility component should not depend on the components it is designed to service. This structure certainly does not fit well within the layered-architecture style. It may be that developers are cutting corners for simplicity.

## Validation Engine

The Validation Engine was found to have two new subsystems that are now dependent on this module. The Connection Manager was found to depend on the Validation Engine for information such as the deployment status of a block and to filter out request handling, while the Wallet depends on it for verifying outgoing transactions. There are also two new bidirectional dependencies with App and Util.

## Storage Engine

The Storage Engine subsystem was found to have five new bidirectional dependencies with Mempool, Mining, Connection Manager, App and Util. The Mempool provides the Storage Engine with unconfirmed transactions, the miner provides information for the stored blocks such as the PoW and the Connection Manager provides data on addresses of connected nodes, all this information is then stored on the disk. The Storage Engine was found to not be dependent on the Wallet anymore as the Wallet needs data on the stored blocks but does not have anything to provide the Storage Engine. Lastly, in our Conceptual Architecture, we had Blocks as a separate module as well as a data type under Storage Engine. This was remedied in our concrete architecture as the Blocks module was removed.

## Mempool

The Mempool received multiple changes in the form of four new bidirectional dependencies with the Connection Manager, Storage Engine, App, and Util. Recently verified transactions are supplied to the Connection Manager by the Mempool, while the Connection Manager supplies information about connected peers and their addresses. The Storage Engine works together with the Mempool to store the potentially large number of unconfirmed transactions. Also, transactions were found to be a data type which is a part of the Mempool subsystem instead of being a separate module.

## App

Upon analyzing the App sub-system in Bitcoin Core, it is evident that there is a significant gap between the conceptual and concrete architecture of the software. In fact, the App sub-system was not a part of the initial conceptual architecture. However, after thorough investigation, it was apparent that there was a need for a more robust and extensible architecture to support the growing Bitcoin ecosystem. The App sub-system features a bi-directional dependency with all the other sub-systems in the architecture making them dependent on each other. The sub-system ties together the various

components of the software into a cohesive and functional system, with its modular design to ensure that the software can continue to evolve and adapt to meet changing needs of the Bitcoin ecosystem.

### Connection Manager

Investigating the gaps between architectures, there were drastic differences regarding the Connection Manager sub-system. For starters, the sub-system is no longer connected to sub-systems such as Peer Discovery, Transactions, and Blocks. The component of Peer Discovery was transitioned to fall under the Connection Manger while Transactions and Blocks transitioned to forms of data stored in other sub-systems. The bi-directional dependency between Wallet was also transformed to a unidirectional dependency as it is no longer dependent on that sub-system. Moving on to the Connection Manager's new dependencies, it is now bi-directionally dependent with existing sub-systems such as Mempool and Storage Engine to replace the previous dependencies of Transactions and Blocks, as well as new components such as App and Utils. In addition, it also features new unidirectional dependencies with Validation Engine and Mining, making said sub-systems dependent on the Connection Manager.

## Low Level Architecture: Mining

### Overview

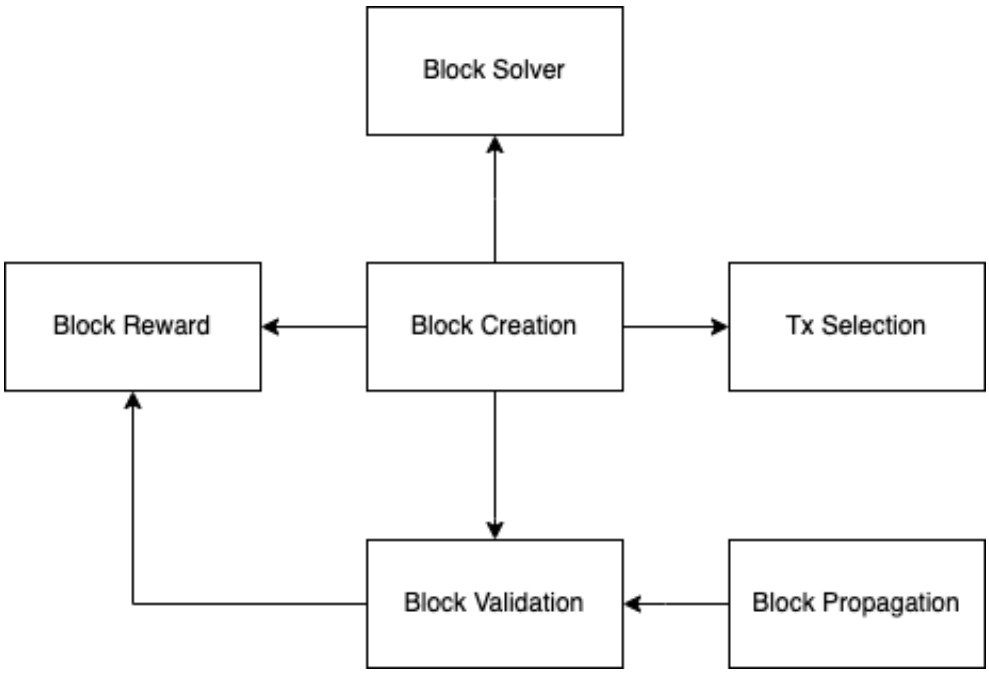


**Figure 3.** The conceptual architecture for Mining

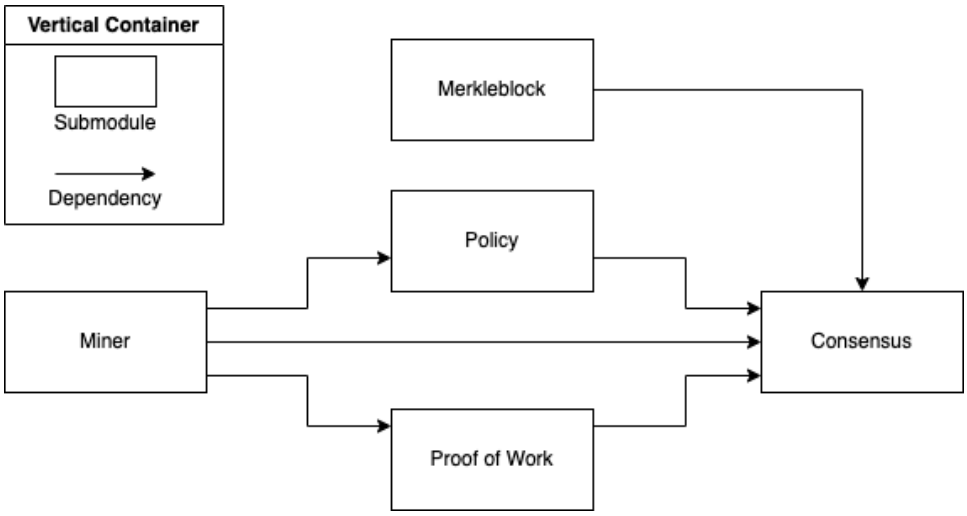


**Figure 4.** The concrete architecture for Mining

In order to gain a more complete understanding of Bitcoin Core, we examined the Mining subsystem and derived both its conceptual and concrete architectures. Mining is involved in the minting of new Bitcoins validating transactions and updating the state of the blockchain. The latter is especially crucial to Bitcoin's overall functionality as a decentralized cryptocurrency, since Bitcoin must be self-sustaining and inherently secure without the oversight of a central authority.

Based on internal developer documentation for Bitcoin Core that we found on GitHub, we created the following conceptual architecture. We then used Understand to make the concrete architecture, analyzing dependencies between files and their source code to build subsystems. In general, we found that both the concrete and conceptual architecture followed a pipe and filter style. For the conceptual architecture, incoming blocks flow from the Block Propagation and Block Creation submodules to Block Validation, Block Reward, Transaction Selection and Block Solver, each of which are responsible for a part of the Mining subsystem's functionality. After a thorough analysis of the source code, the concrete architecture we derived has the same number of subcomponents, but is organized in a slightly different manner, containing Miner, Proof of Work, Consensus, Policy and Merkleblock. Miner and Merkleblock are both source filters which receive data from outside the subsystem, with Consensus being the final filter. Miner pipes its outputs to Proof of Work and Policy, which in turn pass their outputs to Consensus, whereas outputs from Merkleblock are piped directly to Consensus.

# Components & Dependencies

## Miner

The Miner submodule is the base of the Mining subsystem, corresponding to the Block Creation submodule. Block Solver, Transaction Selection, Block Validation and the Block Reward submodules all depend on Miner, as the process of creating a new block requires the mining node to receive a list of transactions, validate them and solve the proof of work problem.

## Merkleblock

The Merkleblock submodule corresponds to Block Propagation and contains the definition of a Merkle block, a data structure that represents a partial Merkle tree. It allows for the secure recovery of both the Merkle root and the subset of transaction IDs that were included in the partial Merkle tree. By using a Merkle block, a full node in the Bitcoin network can efficiently validate that a transaction is included in a block without having to download, store and process the entire block. This submodule is one of the first filters in the subsystem, as it receives Merkle blocks which are then passed to Block Validation, otherwise known as Consensus.

## Consensus

The Consensus submodule is responsible for implementing the consensus mechanism of Bitcoin, which is the rules through which nodes reach agreement about the state of the blockchain. It contains the Block Reward and Block Validation submodules, as Consensus handles both the validation of incoming Merkle blocks and the minting of new Bitcoins as rewards for the miners. This can be seen in the concrete architecture, where data from Merkleblock, Policy and Proof of Work is piped into Consensus. Once consensus has been reached, the newly created block can then be added to the blockchain.

## Policy

The Policy submodule of the mining subsystem in Bitcoin Core enforces rules and constraints when creating new blocks and processing transactions in order to ensure the network remains secure and functional. It requires a valid PoW for each block and sets and enforces several rules related to the size and contents of a block (Ex. a maximum block size) so that network efficiency can be upheld. Beyond this, it also regulates transaction fee policies and their priority based on their fees. This incentivizes miners to process transactions and participate in the network.

## Proof of Work

The Proof of Work submodule is equivalent to the Block Solver submodule from the conceptual architecture. It is responsible for the implementation of the Proof of Work algorithm that is used by Bitcoin to secure the blockchain, and it verifies if block hashes satisfy the proof-of-work requirement. It also includes the definition for the header of a Bitcoin block. In the concrete architecture, it receives data from Miner and its output is piped to Consensus.

# Reflexion Analysis

## Miner

For the Miner submodule, there are no new dependencies. We noted that that this submodule still has outgoing connections to Policy (Tx Selection), Proof of Work (Block Solver), and Consensus (Block Validation). The connection to Block Reward was integrated with Consensus, and thus is maintained in the concrete architecture.

## Merkleblock

In the initial conceptual architecture, Merkleblock was categorized under Block Propagation. However, upon contrasting the conceptual and concrete architectures, it becomes evident that these two align with each other, with some additional components added in the concrete architecture. Furthermore, this alignment with the conceptual architecture is reinforced by the fact that Merkleblock, which belongs to Block Propagation, is still dependent on Block Validation, which is part of Consensus. Despite these consistencies, the conceptual architecture fails to account for several external dependencies, including an outgoing connection to Storage Engine and multiple incoming dependencies from the Wallet and Connection Manager modules.

## Consensus

The Consensus submodule has kept its old incoming dependencies from Miner (Block Creation), and Block Propagation (Merkleblock). However, it has a new incoming dependency from Policy (Tx selection). Policy rules depend on Consensus rules because they need to be consistent with them. If Policy rules contradicted the Consensus rules, then it may lead to a situation where some nodes consider a block or transaction valid while others reject it. By having Policy depend on Consensus, it ensures the blockchain stays consistent and secure.

## Policy

In our concrete architecture, the Policy submodule corresponds to Tx Selection in the conceptual architecture. However, it is important to note that Tx Selection refers solely to the selection of transactions to be included in a block, while Policy also determines the rules for what constitutes valid transactions and governs whether they can be broadcast to the rest of the network. Despite this divergence, the miner (Block Creation) continues to converge and remains dependent on Policy, while Policy now depends on several other submodules, including Consensus, Block Validation logic, and the transaction selection algorithm.

## Proof of Work

In our conceptual architecture, we found that the Block Solver submodule had no dependencies and was only depended on by Block Creation. However, in the concrete architecture, we found that it has a new dependency on Consensus. This is likely due to the algorithm needing to know the next block to work on, which can only be obtained from the information on block parameters found within Consensus.

# Use Cases & Diagrams

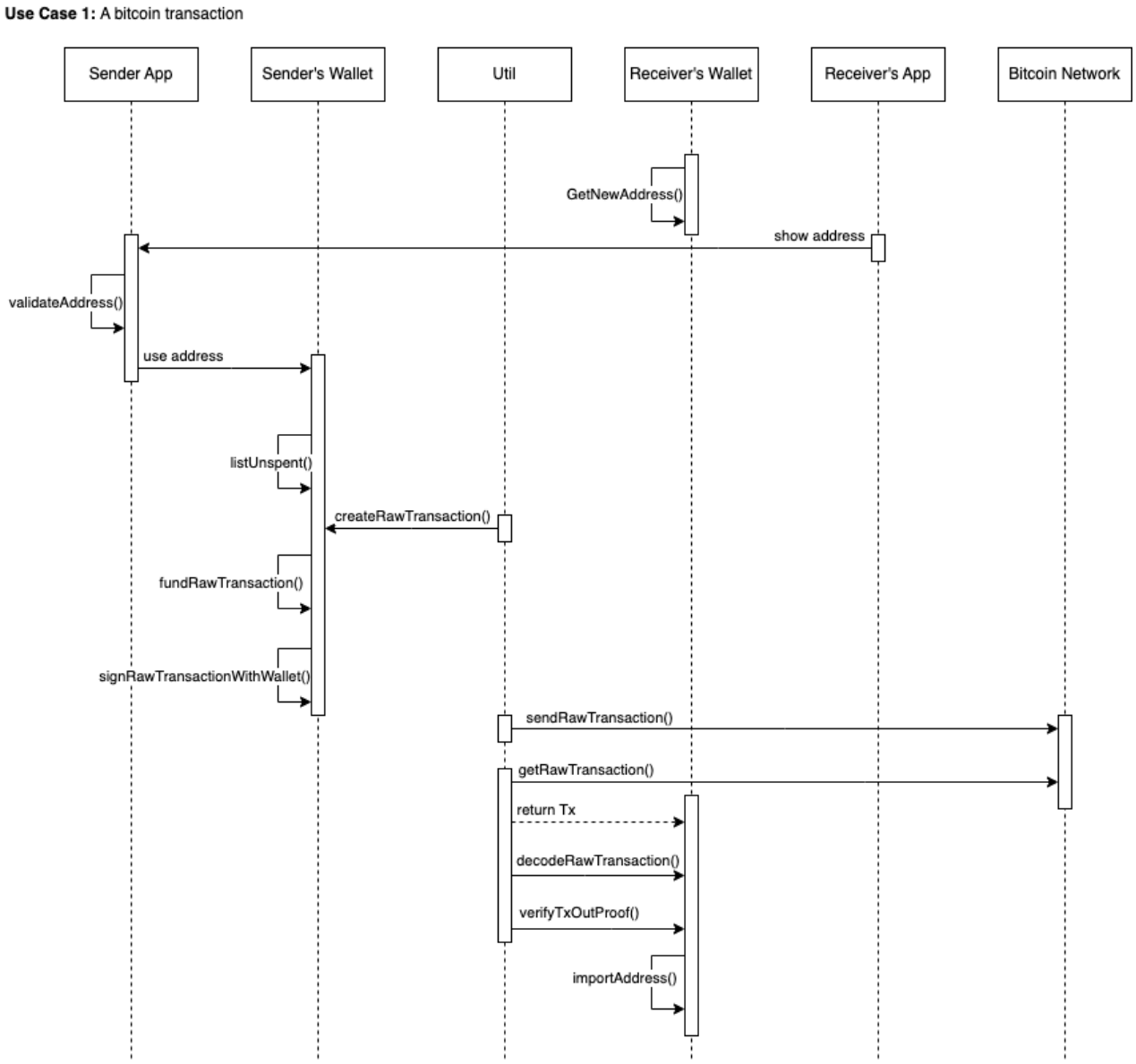


**Figure 5.** The use case of a bitcoin transaction

The first use case involves a bitcoin transaction between a sender and a receiver. The transaction starts with the receiver obtaining the address of their wallet  and displaying it to the sender through their app. The sender's app reads the address, usually through QR code scanning, validates it, and sends it to their wallet to be used. The wallet gathers its UTXOs (listUnspent), and creates a raw transaction funded by the sender's wallet, which then signs it. The transaction is then sent to the bitcoin network to be validated. The receiver's wallet eventually receives the transaction from the network and decodes it to display human-readable information about the transaction. The receiver's wallet then verifies the transaction output and adds it to their wallet (importAddress), allowing them to spend funds from the transaction output in the future.

The main difference between this use case and the conceptual architecture is the substitution of nodes with apps, as it is the primary source of communication between the two parties and how the wallet address is shared. Additionally, the Tx component was removed as it represents data rather than a module, and a Util component was added to incorporate some functions in this use case.

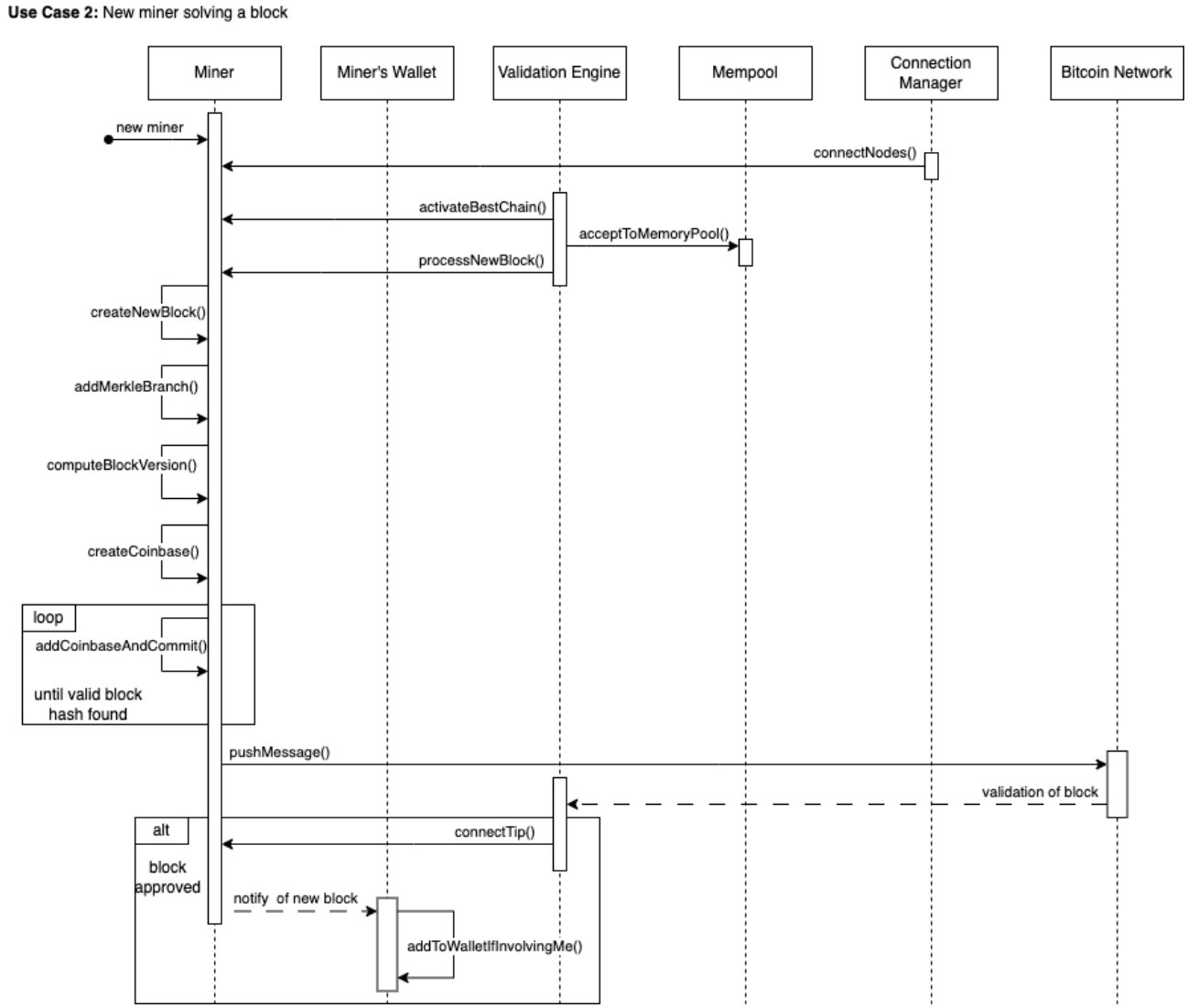


**Figure 6.** Use case for bitcoin mining

The second use case involves bitcoin mining using a new miner. When a brand-new miner joins the network, the connection manager connects it to other nodes on the network and validates the blocks in its local copy of the blockchain (activateBestChain). The miner then begins listening for new transactions in the mempool (acceptToMemoryPool), and processes new blocks to be added to its local copy of the blockchain. Next, the miner creates a new block by adding information to the block header (addMerkleBranch, computeBlockVersion) and generating the coinbase transaction. Once the block is set up, the miner adds the coinbase and starts the PoW algorithm to find a valid block hash (addCoinbaseAndCommit). If a valid hash is found, the miner propagates the block to the network (pushMessage). If other nodes validate the block, the miner adds the confirmed block to the blockchain

(connectTip), and notifies the wallet of a new block. Finally, the wallet adds the coinbase transaction outputs to the miner's balance (addToWalletIfInvolvingMe).

This diagram differs from the conceptual architecture's sequence diagram in a few ways. Blocks and transactions were removed since they represent data rather than modules, and the connection manager was added. Additionally, some events were combined, such as adding the coinbase transaction and mining in addCoinbaseAndCommit.

# Conclusion

In conclusion, making use of online resources, the Bitcoin repository, and the Understand software, our group was able to successfully examine, analyze, and contrast Bitcoin Core's concrete architecture. In this report, we performed a high-level reflexion analysis of 8 subsystems defined by the concrete architecture and delved deeper into the Mining subsystem's low-level architecture, also performing a reflexion analysis. In these analyses, we discovered that the overall architecture is consistent with our previous report being a pipe-and-filter architecture, however, there were several notable divergences that were outlined in this report. Beyond this, we explored limitations and lessons we encountered and used sequence diagrams to better illustrate two different use cases given our concrete architecture. Overall, spending time diving into the repository and code itself to derive the concrete architecture of Bitcoin Core was immensely useful to fully understand Bitcoin Core and how often Concrete and Conceptual architectures diverge from each other.

# Limitations and Lessons Learned

During the development of Bitcoin Core's Concrete Architecture, the team faced several limitations. One of the main limitations encountered was the complexity of the software. Bitcoin Core's architecture is nowhere near simple as it involves a variety of complex and intertwined components. Attempting to understanding the interdependencies between different sub-systems and use the Understand software, the team cross-referenced their architecture with the graphical representation of interactions between modules in order to piece together the final Concrete Architecture. Another limitation was the lack of clear documentation within the repository, making it challenging to identify the purpose of various modules and files as well as make those connections with other dependencies. This resulted in a significant amount of trial and error as the team worked to develop and refine the software architecture.

In terms of lessons learned, the team found it critical to take a modular approach to constructing Bitcoin Core's Concrete Architecture. This allowed for the separation of concerns and the ability to make changes to specific sub-systems and their dependencies without greatly affecting the overall architecture. Another lesson learned was the importance of continuous testing and validation. The ability to come up with a complete and correct Concrete Architecture first-try is quite rare. Thus, the team had to meet regularly and persist through trial and error to ensure that the architecture that they came up with was as accurate as possible for the complex software. Moving forward, the team will continue holding regular meetings in which they can communicate with each other and delegate tasks to collaborate effectively and efficiently.

# Data Dictionary

Transaction – A structure representing a transfer of value from one Bitcoin address to another

Coinbase Transactions – The first transaction of every new block will contain this. It holds the reward for the miner should they solve the block. The reward total is calculated by combining the transaction fees of all transactions in the block together with the reward of the current block height. The block height reward starts off at 50 bitcoin and is halved every 210,000 blocks. Currently, it is at 6.25 bitcoin.

Block – A structure that contains around 1900 transactions and the information necessary for miners

UTXO – An output of a Bitcoin transaction that represents a discrete amount of Bitcoin credited to an address

Blockchain – New blocks reference a "parent" block and thus act as a chain.

Node – A participant in the Bitcoin P2P network, which could be an instance of Bitcoin Core

# Naming Conventions

P2P – Peer-to-peer

TX – Transaction

GUI – Graphical user Interface

CLI – Command-line Interface

UI – User interface

PoW – Proof of work

App – The application component

RPC – Remote procedure call

# References (Everyone)


Asplund, Mikael, et al. "In-Store Payments Using Bitcoin." *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, 2018, https://doi.org/10.1109/ntms.2018.8328738.

"Chapter 10: 'Mining and Consensus'." *Chapter 10: 'Mining and Consensus' · GitBook*, https://cypherpunks-core.github.io/bitcoinbook/ch10.html.

"Transactions." *Bitcoin*, https://developer.bitcoin.org/devguide/transactions.html.

"What Are Bitcoin Blockchain Nodes? - Bitstamp Learn Center." *Bitstamp*, https://www.bitstamp.net/learn/crypto-101/what-are-bitcoin-blockchain-nodes/.

"Chapter 6: 'Transactions'." *Chapter 6: 'Transactions' · GitBook*, https://cypherpunks-core.github.io/bitcoinbook/ch06.html.