Adamnite

An accessible, scalable, and secure blockchain development platform.

Archie Chaudhury

archchaudhury@adamnite.org

2022

Abstract

Since the advent of Bitcoin in 2009, cryptocurrencies and other digital assets have reshaped the store and transfer of financial value. Bitcoin was the first successful implementation of a decentralized currency supported by a peer to peer network, relying on no centralized authority or power to instill validity for the underlying asset. The creation Recently, digital assets have gained immense attention due to their use-cases beyond just currency transfers, with Non-Fungible-Tokens (NFTs) and other use-cases becoming strong markets in their own right. Yet, despite the importance of digital assets, the invention of Bitcoin also led to the creation of blockchain technology as a tool for data-storage, validation, and consensus. Blockchain technology, along with its close counterpart Distributed Ledger Technology (DLT), has reshaped the Internet; Decentralized Autonomous Organizations (DAOs), InterPlanetary File Storage (IPFS), and Decentralized Applications (DApps) are all part of the new Internet commonly referred to as “Web3”. Accordingly, Adamnite seeks to produce a fully permissionless blockchain platform that will allow developers across the world to confidently create Web3 applications. Utilizing a secure DPOS-based consensus mechanism, leveraging an efficient execution model for on-chain programs, and coming with an easy to use native high-level programming language, Adamnite will be more scalable and accessible than a majority of current blockchain solutions, thus accelerating the adoption of blockchain technology across both public and private sectors.

Table of contents

List of figures

Chapter 1

Background

1.1 History

Distributed Ledger Technology (DLT), the larger group of software that encompasses blockchain, has its roots in the early Roman Empire, which allowed its citizens to barter across the entirety of the empire using a record-keeping system. Any asset database that is shared across multiple nodes and does not rely on a central administrator can be defined as a DLT. DLTs, despite their potential, were never adopted enmasse in modern software due to the fear of malicious actors overtaking the system. This problem was summarized as "The Byzantine Generals Problem" by Leslie Lamport, Robert Shostak, and Marshall Pease in their 1982 paper of the same name. The paper describes an imaginary abstract situation in which the generals of the Byzantine army are separated while scouting an enemy base, and must use messengers to communicate with one another in order to decide on an appropriate plan of attack. However, because they are separated, traitorous generals who have infilitrated the Byzantine army may influence the decision, causing the generals to agree on a plan different from what their commanding general intended. All honest generals must agree upon the correct plan, regardless of the presence of malicious actors in the system. The adoption of DLTs in computational software was held back due to a variation of this problem: rather than referring to generals who are unable to reach agreement on a certain plan of action, the Byzantine Generals Problem as applied to DLTs refers to a group of indepedent computers being unable to reach agreement on the current state of some ledger.

Satoshi Nakamoto proposed a general solution to the Byzantine Generals Problems through Bitcoin, which used a new algorithm that depended on participants solving cryptographic problems in order to reach consensus on the state of a digital currency that settled on a distributed ledger. Bitcoin's initial success was a direct result of its unique integration of Proof-of-Work Consensus with the "Longest Chain Rule", which refers to the philosophy of assuming that the longest valid ledger is the most legitimate one. This implementation has come to be known as Nakamoto Consensus. In Bitcoin, Nakamoto Consensus specifically solved the Byzantine Generals Problem by incentivizing block validators, also known as miners, to validate the longest and most accurate chain. This ensured that all participants can have confidence in the validity of the current chain of transactions without having to look to a centralized authority. Bitcoin also solved the double-spending problem that had plagued previous digital currencies such as Digicash by introducing a timestamp record for each transaction; this ensured that no singular Bitcoin was “spent” multiple times by the same account. The idea of grouping these transactions into blocks, which were then validated through Nakamoto Consensus and stored on a public ledger, was revolutionary, and represented the first real use of what is now known as blockchain technology. The consensus mechanism also protects the network against malicious actors: an attacker will need to control over 50% of the computing power in the network in order to gain control of the blockchain (commonly known as a 51% attack). Bitcoin’s success paved the way for blockchain technology and DLT in general. In the years following Bitcoin’s release, multiple blockchains and their corresponding digital assets have been released, with each new chain focusing on a different method to improve upon Bitcoin’s original model. For example, Ethereum proposed a blockchain with a built-in Turing Complete programming language that allows for the creation of programmable contracts that can define transactions between two parties, store data, and tokenize other data. On the other hand, platforms such as Algorand and Solana have leveraged different consensus mechanisms and technical frameworks to create blockchains that are more scalable for enterprise use.

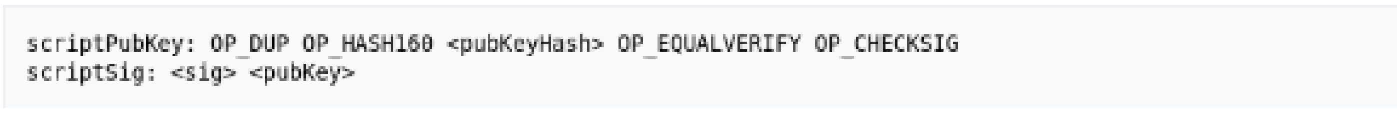
1.2 Going beyond the tokenization of assets

Bitcoin’s rise propelled the creation of new subsets of software and even the development of entirely unique sectors; for example, Decentralized Finance, or DeFi, rose entirely out of Bitcoin’s successful implementation of a decentralized currency with a shared state. Most importantly, the ability to make monetary transactions without a centralized authority was, and still is, revolutionary: users can store value, make payments, and more, all without reliance on a trusted third party. However, blockchain technology has applications beyond just financial transactions and asset management. For example, a decentralized governance application for voting could easily be built using smart contracts that are stored on a blockchain, with votes being recorded directly on the public ledger. Additionally, a storage service can also be built on the blockchain, enabling files and other forms of data to be encoded within the public ledger and to be shared with different participants in the peer to peer network. A more recent innovation are Non-Fungible-Tokens, or NFTs. NFTs are singular tokens representing a proof of ownership, and have seen numerous use cases in digital art, real-estate, and content management. Decentralized applications, or DApps, are applications that use smart contracts and ultimately settle on a ledger instead of a centralized group of servers. DApps are traditionally built on blockchains with a virtual machine that supports quasi-Turing complete execution, thus allowing for them to exist indefinitely. To support the growing number of applications, blockchains must be both scalable and secure when compared to their traditional counterparts.

Unfortunately, most alternative use cases are currently limited, at least in most current blockchains. In Bitcoin, computation on a large scale is intentionally limited, to maintain both decentralization and Bitcoin's main use case as a store of value at the base layer. This is largely due to Bitcoin's consensus model, which is perfect for a decentralized currency supported by a peer-to-peer network, but fails when used for higher-level applications. In particular, Bitcoin's utilization of POW means that individual transactions slow (on average, a transaction takes around 10 minutes)/ This not only makes Bitcoin unsuitable for enterprise use, but also places restrictions on the use-cases for the underlying blockchain technology. The amount of computational power needed to process blocks of transactions also makes Bitcoin somewhat restricted; governance and gaming applications may need to process 1000s of transactions per minute in a cheap and efficient manner, which is something Bitcoin cannot currently handle natively. Bitcoin also has no direct representation of state: while there is a mapping between accounts and balances, storage of data and native instructions is left to transactions.

1.2.1 Programming with Bitcoin Script

Development on Bitcoin is primarily done through two means: direct additions to the Bitcoin Core software that allow for scalablity , and implementation of the current Bitcoin Protocol with the Bitcoin Script programming language. Here, we will focus on the Bitcoin Script programming language as it is method for creating programs that leverage the Bitcoin blockcahin. Script is a stack-based programming language that allows for a transaction to have direct specifications with regards to how the receiver will be able to unlock the coins to be spent or transferred elsewhere. This is implemented through operations on Unspent Transaction Outputs (UTXOs) that essentially govern when and how a certain portion of the currency is made available. This allows for the creation of basic smart contracts that can be used to create applications that offer payouts based on certain requirements being met. These requirements can range from the completion of tasks that can be verified by the program to a simple exchange payout which dictates that you must send x amount of another asset in order to receive the corresponding amount of Bitcoin. Below, a code snippet of a Bitcoin Script program designed for a simple transaction to a public key address is defined.



```
scriptPubKey: OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
scriptSig: <sig> <pubKey>
```

Fig. 1.1 Source: Bitcoin Wiki

The opcodes *OP_DUP*, *OP_EQUALVERIFY*, and *OP_CHECKSIG* define functions for duplicating the top stack item, checking that all the inputs are equal, and checking that the top item of the stack is valid, respectively. The Script programming language comes with multiple opcodes that can be used to create unique contracts depending on the preferred use-case.

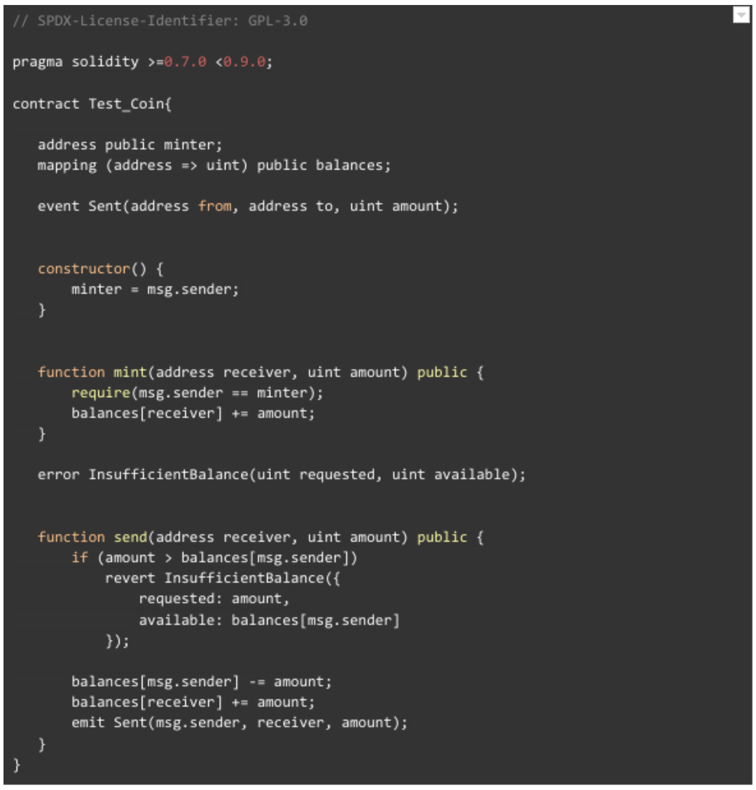
While Script can support a plethora of applications that connect directly to the Bitcoin blockchain, it lacks several key features that prevent it from being a long-term scalable solution in blockchain development. Script only focuses on UTXOs, thus limiting applications to only transactions on the blockchain. This prevents developers from creating smart contracts that can take into account or mutate on-chain data. While Script does allow for the creation of programs that can define data that settles through transactions, this is more optimized for side payment channels (such as the lightning network) rather than Script is also decidedly complex; its syntax and structure make it difficult to implement for more difficult use cases. For example, an average script for a MultiSig Transaction will require all users to send custom scripts in order to have the transaction function as intended. This leads to inefficiency when creating scripts that handle more complex operations and transactions.

1.2.2 Programming with Ethereum

The Ethereum Blockchain, described as “A Next-Generation Smart Contract and Decentralized Application Platform” in its initial white-paper, was meant to be a direct improvement over Bitcoin in terms of its scalability. Coming with a built-in Turing Complete programming language, the Ethereum Blockchain has become the main platform for DApp Development, with numerous developers and organizations using the platform to power their own blockchain base solutions. Programs written in Ethereum are significantly more scalable than their Bitcoin counterparts due to both their Turing Completeness and their capacity to process state; loops and on-chain data are frequently used to create programs that are both efficient and responsive.

Ethereum’s most popular programming language is Solidity, a high-level programming language based heavily on Javascript. Solidity is Turing-Complete, and any code written in

Solidity is meant to be transitioned to lower level byte code and run on the Ethereum Virtual Machine (EVM). A typical Solidity program is built around contracts, which are essentially classes that define various structures within a larger program. A contract can be used to create and define operations for a new asset, governance mechanism, or identity verification tool. A simple smart contract designed to create a new asset, taken from Solidity's Docs, is shown below:



```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;

contract Test_Coin{

   address public minter;
   mapping (address => uint) public balances;

   event Sent(address from, address to, uint amount);


   constructor() {
       minter = msg.sender;
   }


   function mint(address receiver, uint amount) public {
       require(msg.sender == minter);
       balances[receiver] += amount;
   }

   error InsufficientBalance(uint requested, uint available);


   function send(address receiver, uint amount) public {
       if (amount > balances[msg.sender])
           revert InsufficientBalance({
               requested: amount,
               available: balances[msg.sender]
           });

       balances[msg.sender] -= amount;
       balances[receiver] += amount;
       emit Sent(msg.sender, receiver, amount);
   }
}
```

Fig. 1.2 Solidity Code

Contracts support the use of functions, events, errors, etc. This allows developers to create complex smart contracts that respond directly to input. The contract above, for example, defines the event Sent, which signifies the transfer of the asset from one party to another.

Despite Ethereum's advantages, it still possesses several problems that act as a barrier to widespread institutional adoption. Namely, the high transaction fees in Ethereum, commonly referred to as Gas, mean that every smart contract must be carefully optimized to reduce deployment costs. Based on current network congestion, the deployment of a simple smart contract such as the one described above can cost upwards of 400 USD. Additionally, Ethereum programs, while significantly more high-level than their Bitcoin counterparts, come with their own nuances. Most notably, Ethereum code lacks a high degree of composability and reusability, thus creating an ecosystem where the same code is often deployed over and over again. While libraries in Solidity have provided a temporary fix, they still remain to be widely used. Internal functions can be called recursively; however, there are clear limits based on both the size of the stack and the underlying memory. Ultimately, programming on the Ethereum blockchain on an enterprise-level scale is extremely difficult for the average programmer due to its underlying intricacies, with gas optimization and a lack of significant modularity being the key reasons. This causes developers to make rudimentary mistakes, leading to applications that have fatal security vulnerabilities as a result of simple errors. These vulnerabilities result in significant financial loss every year as malicious actors take advantage of logical loopholes within the smart contracts driving these applications.

1.2.3 Alternative Methods and Chains

Currently, alternative blockchain platforms such as Algorand, Cardano, and Polkadot seek to make blockchain technology more scalable for both institutional adoption and DApp development. These platforms are often an improvement on existing solutions: Algorand and Cardano both reach on-chain consensus through a proof-of-stake implementation that drastically reduces transaction costs, while Polkadot offers interoperability, cross-chain transfers, and an efficient execution model based on WebAssembly (WASM). However, these chains have their own unique problems, with centralization, a lack of scalability, and a lack of security being several of the most commonly cited issues. Current blockchain solutions often also focus on developing infrastructure for a specific area or use-case, with most chains placing a large emphasis on DeFi. This leads to platforms that are well-suited to dictate asset transfers, but less apt for other use-cases such as data storage.

Furthermore, as with Ethereum and Bitcoin, the learning curve for most alternative chains are extremely steep. While a skilled developer should be able to execute automatic transactions with relative ease, creating complicated smart contracts or applications that manipulate on-chain data is much more difficult. In order to create a full-fledged DApp, developers often have to contend with low-level languages with little documentation, ensure that their application is secure, and find structural support for extraneous use cases. While community-based development (the creation of multiple SDKs on Algorand is a good example) has made blockchain development more accessible, they are still not integrated into the blockchain's core smart contract architecture. Although current blockchain solutions succeed in facilitating decentralized currencies, they remain difficult to use for standard application development. This significantly limits the adoption of blockchain beyond finance, as most developers will prefer the more efficient and easier legacy solutions. Furthermore, a majority of alternative chains are often dependent on EVM compatibility to generate meaningful adoption, and often settle for Solidity as their go-to high-level language, resulting in an ecosystem with little to no viable alternatives for developing smart contracts. EVM-dependency also makes blockchain development somewhat isolated; the EVM is fundamentally different from other popular virtual machines, making it difficult for both developers to learn its intrcracies and for enterprises to port legacy applications. While Polkadot (and Cosmos) allow for the usage of WASM, its mainly restricted for developing application-based side chains, rather than native contracts.

Chapter 2

Adamnite

Adamnite is meant to represent the future of blockchain development, allowing both professional developers and hobbyists to leverage blockchain technology without having to go through an arduous learning process or sacrificing a large amount of computational resources. Furthermore, a significant focus will be put on enterprise level use: the Adamnite blockchain should be faster, safer, more powerful, and cheaper to implement than existing blockchain solutions. Adamnite accomplishes this by introducing a blockchain development platform that is user/developer oriented while simultaneously optimized for large-scale application development. By providing a system that is more intuitive and easier to understand, Adamnite hopes to create a platform that encourages more developers and organizations to embrace blockchain development.

Adamnite focuses on three main goals:

Ease of Use: Adamnite's key appeal is its simplicity: its programming language and associated environment should be as easy to learn as Python, Javascript, etc. Abstraction and modularity will be key; the Adamnite blockchain should emphasize ease of development even if it comes at the cost of increased storage or bloat on-chain (however, we do predict that the marginal decrease in storage due to comosable contracts will offset any immediate storage cost). An example will be the use of libraries: developers using Adamnite should have the ability to download community-vetted libraries that will aid them when creating simple programs.

Security: Organizations and businesses should easily be able to implement Adamnite into their current framework without worrying about security or failure. This applies to both smart contract security, and settlement security: smart contracts developed on Adamnite should have guards that prevent unexpected behavior or exploits, and the blockchain should not halt due to coordinated economic or network attaks. The

Adamnite Blockchain should also remain accessible: anyone running a node should be eligible to validate different aspects of the blockchain or be selected in the leader election process to propose/validate new blocks.

Scalability: Apps built on Adamnite's blockchain utilizing multi-party smart contracts should be able to support numerous users and a generally high message throughput. A state-execution model, based on the popular WASM paradigm used by Polkadot and Cosmos, among others, should allow for both efficiency and potential interoperability with other development platforms. This execution model should be efficient while retaining both decentralization and storage efficiency.

2.1 Adamnite Transactions

Adamnite, like other distributed ledger technologies, uses a native digital asset to act as both an economic incentive for computers who are performing upkeep for the network and as a guard fee to prevent unmitigated use of the underlying network. Like Ethereum and other 2nd/3rd generation blockchain networks, Adamnite utilizes an account-based model, meaning that transactions and state-transitions are tracked in individual accounts instead of unspent transaction outputs as with Bitcoin. Adamnite has two account types: autonomous (controlled by on-chain code), and manual (controlled by an external party). All accounts have the ability to send and receive transactions, and contain the same standard fields:

1. Nonce: the amount of transactions the account has sent.
2. Balance: the total amount of NITE that the account owns
3. Code Hash: A hash of the code that controls the account. This field is empty if the account is manual.
4. Storage Hash: A hash that represents the state, or storage, for the account. This field is empty if the account is manual.

Adamnite's native currency is stylized as Nite, and is used to transfer value within the network. Transactions on the Adamnite network contain the same standard fields as any cryptocurrency platform:

1. The type of transaction
2. Sender's Public Adamnite Address

3. Amount of NITE being sent
4. Public Address of the recipient
5. Message, an optional field where additional data can be stored.
6. Message_Size, an arbitrary integer describing how large the message is in bytes.
7. ATE_Max, the maximum transaction fee of the transaction
8. Sender's signature

The type of transaction simply refers to whether the transaction is a payment transaction (transfer of NITE from one manual account to another) or an application call (from a manual account to a predefined autonomous account).

The net fee ATE_MAX is determined similarly to Ethereum's net gas price: each transaction, whether made by an external account or smart contract, is analyzed to determine the total amount of processing power or storage imposed on the blockchain. Like Ethereum's model, this is to prevent malicious players from taking advantage of the network by sending repetitive transactions that consume a lot of computing power. However, due to Adamnite's consensus model, net transaction fees are a lot lower than legacy chains, drastically increasing scalability and performance for on-chain applications.

2.2 Consensus Mechanism

Adamnite's blockchain and consensus protocol leverages a variation of Delegated Proof of Stake (DPOS). DPOS is a consensus mechanism in which participants vote on a group of validators, or witnesses, to represent them in the network. These validators are then given the ability to both create and approve new blocks. Rewards occur twice: validators who successfully propose a new block of transactions are rewarded for the work they expend on creating a new block and validating it, while active participants who regularly stake their tokens for the purpose of voting are rewarded an amount proportional to what they are staking. An important note is that in order to participate, a node simply needs to send a specific participation transaction (with no actual assets). This participation transaction will essentially communicate the individual node's preferences for the witnesses (a node is allowed to select more than one witness) for the next l blocks, where l is an arbitrary number.

2.2.1 Block Proposal and Agreement

A semi-formal definition of Adamnite's consensus mechanism follows. At the beginning of each round (a round consists of l blocks), all nodes have the opportunity to communicate their preferences for the witnesses. The witnesses who receive the most votes are then sorted into a witness pool of size m (pool A) which will remain consistent until the beginning of the $I+1$ round. Every kth block (k being an arbitrary factor of I), a set of n/m witnesses (with n being an arbitrary factor of m) are chosen randomly to act as validators for the next k blocks, weighed by factors such as the total number of votes they received (each NITE is one vote), their history/reputation as a witness, and their individual stake within the protocol. We define this new pool as pool B. Finally, for each block, a block proposer is selected randomly from pool B. Once a block is proposed, it must be confirmed by at least $2/3$ of pool B in order to be validated and added to the blockchain. This process repeats indefinitely, with pools A and B changing based on the witnesses that are elected by the network. If a witness is found to misbehave by, for example, encoding invalid transactions within their chosen block or trying to propose blocks on two different forks, then they are replaced by another witness from pool A. They also suffer a punitive loss to their reputation: their reputation score is cut by a fixed amount. This also results in the nodes who voted for them ultimately losing their rewards, and thus rendering them less likely to vote for that particular witness in the future.

Ultimately, Adamnite's consensus mechanism seeks to minimize block latency (the time it takes for one block of data to be agreed upon and uploaded to the network) while also maintaining a degree of security and decentralization. DPOS, by limiting both block production and validation to a small subset of democratically elected witnesses, allows for both mass scalability while allowing for widespread participation among the network. In that sense, Adamnite's consensus mechanism can be thought of as a democracy implemented on a public ledger, with the candidates being potential witnesses and voters being participants.

The Adamnite Blockchain also plans to use governance, similar to that of a Decentralized Autonomous Organization (DAO), to make protocol and structural changes to the blockchain. This is similar to the governance mechanism created by Algorand, where users vote on various proposals and have the opportunity to earn rewards in exchange. In Adamnite, governance will be used to select both delegators (individuals who oversee the governance of the network) and vote on legitimate proposals coming from the delegators. This can be used to both vote on both protocol-level decisions, such as the amount of rewards to give witnesses and their stakeholders, and development decisions such as the initial screening for improvement proposals.

2.2.2 Witness/Validator Selection

Adamnite's consensus mechanism maintains several key differences from traditional staking protocols. Adamnite blocks contain a list of all the individual storage messages for each transaction and the net storage size in bytes. Furthermore, a validation list (a copy of the public addresses of both the block proposer and validators, along with a cryptographic proof that they all were indeed in the set of validators chosen for that particular block) and the block number are also recorded. In order to validate a block, a witness W needs to first ensure that the previous block is valid, ensure that the current block has a reasonable timestamp $tn(tn-1 < tn < tn-1+10$, where t is recorded in minutes), ensure that each transaction is successful based on transaction parameters, ensure that the block number is valid, and finally ensure that the block proposer is a part of the pool of selected witnesses (previously defined as pool B). While there may be concerns over a malicious party specifically targeting the witnesses for a specific block based on both the public record of chosen witnesses and the inclusion of the validator list in each individual block, Adamnite's unique DPOS scheme should prevent this. Specifically, a cryptographic sortition scheme, similar to ones employed by Algorand and Witnet, is leveraged to select the individual from the top n addresses as determined by vote, where n is inherently dependent on the total number of participants in the ecosystem. This process is repeated every round. Specificially, the cryptographic sortition, by leveraging a verifiable random function (VRF) ensures that certain validators are not picked out and attacked, a problem that has plagued other blockchains.

$$h(sig(t, rand(i), M)) < K$$

In the above equation, h represents a Verifiable Random Function (VRF), the signature function takes in time, a random value, and the key M, and K represents the total number of votes the particular address received during the previous "election". This equation can be altered to take on additional weights: for example, for Adamnite's specific implementation will takes into account the stake of the account, and its reputation, as measured by its past behavior when selected as a witness. For each block, the probability that an address within n will be chosen to validate it is inherently dependent on the amount of votes that were allocated to it. The cryptographic sortition scheme should ensure that the identity of a block proposer, or even the group of witnesses acting to validate various blocks within the chain, is reasonably protected.

Chapter 3

Programming on Adamnite

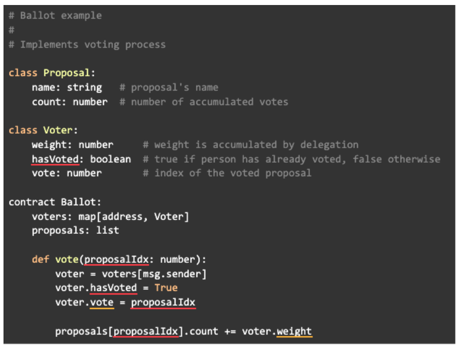
All transactions and autonomous contracts on Adamnite are executed by the Adamnite Virtual Machine (ADVM), which executes native byte code, defined as ADVM code. ADVM is a computational engine that is shared among all Adamnite nodes, and allows for programs written in a high-level source language to be executed on the blockchain. ADM code is Turing-Complete: it is meant to allow developers to have as much flexibility as possible when creating applications. Code written by developers in a high-level language is compiled to opcodes, which are then run on the ADVM. These opcodes can define simple transactions and payouts, or take into account external data to manipulate the overall state of the network. To achieve Adamnite's goal of both safety and modularity, the ADVM (and by extension, ADVM code) are implementations of the popular Web Assembly (WASM) paradigm. WASM is a general format for executing binary instructions in different environments. Its most popular use-case is for serving as a common standard for internet-based web applications. WASM was chosen for Adamnite because of its standardization and modularity: developers are more likely to be family with executing operations within WASM's framework (for example, standard arithmetic uses either 32 or 64 bits, rather than 256 bits as with the Ethereum Virtual Machine), and can create specific WASM binary modules as needed to supplement the core implementation. WASM's standardization also helps with on-chain code readability, thus allowing users to validate the low level byte code that is actually deployed on-chain.

3.1 A1 Programming Language

The ADVM will be the primary target of the A1 Programming Language, a high-level pythonic programming language designed for writing contracts and on-chain applications. A1's main advantage over other popular alternative smart contract programming languages comes from its ease of use, modularity, and assertion-based security: developers should be able to easily learn the basic syntax/semantics of the language, use vetted and predefined libraries to create complex programs, and declare various properties for checking during compilation. While modularity and ease of use enable developers to quickly create complex applications, property-based testing ensure that the individual contracts and programs that comprise these applications are secure. A1 is greatly inspired by the E programming language, a general-purpose programming language created specifically for distributed computing created by Mark Miller, Dan Bornstein, and others in 1997. E was also one of the first programming languages to introduce the concept of smart contracts by allowing developers to define programs that could execute agreements between multiple parties by supporting distributed computing.A1 extends the fundamental principles of E to create a programming language capable of defining high-level instructions that ultimately execute on a blockchain. A1 also fits in naturally with the ADVM due to its modular nature, which corresponds to WASM modules.

3.1.1 Modularity and Scripting

The development of the A1 ecosystem, and Adamnite as a whole, will be an iterative process; as more applications, libraries, and packages are built for the A1 Programming Language, the easier it will be for new developers to get started. A1's similarity to Python and other mainstream programming languages also means that more and more Adamnite's open-source nature also means that its standard library will become completely community-run over time, leading to a self-sustaining ecosystem that does not depend on a centralized authority or party for continued maintenance. Over time, developers may elect to use different versions of A1, leverage specific packages, or spin up different distributions for specific needs. In that sense, Adamnite's community will evolve to be that of not unlike Python or C++; its goal is to enable a shared community of developers to create functional programs such as the one below:



```
# Ballot example
#
# Implements voting process

class Proposal:
    name: string   # proposal's name
    count: number  # number of accumulated votes

class Voter:
    weight: number     # weight is accumulated by delegation
    hasVoted: boolean  # true if person has already voted, false otherwise
    vote: number       # index of the voted proposal

contract Ballot:
    voters: map[address, Voter]
    proposals: list

    def vote(proposalIdx: number):
        voter = voters[msg.sender]
        voter.hasVoted = True
        voter.vote = proposalIdx

        proposals[proposalIdx].count += voter.weight
```

Fig. 3.1 A1 Ballot Example

3.1.2 Verification for Security

On-going safety for smart contracts is where a specific part of Adamnite's proposed ecosystem comes into play: every one of Adamnite's programming languages will have a compiler with a verification engine that checks for logical invalidity and specific assertions defined by the developer. One of the primary goals for integrating this with the A1 programming is to integrate a verification engine within the compiler, thus allowing for checks on logical invalidity and assertions before compilation to WASM. An example follows. A developer wants to ensure that a smart contract will not allow a participant Alice to arbitrarily withdraw funds through a loop. They will simply need to declare this through an assertion in a separate script used specifically for compilation. This assertion can simply be a check of a single variable, or something more complex such as checking the interim value of a function's output. Whatever the case, the compiler will be able to check the assertion against the body of code for which it is defined and see if it holds. The compiler will also check for loop invariants, which are essentially conditions that are true at the start of every ith iteration of a loop. The compiler's security will be mostly centered around property-based testing, thus allowing for developers to check individual parts of their code to ensure that it functions as intended before deployment. These properties can be as simple as a binary statement comparing two variables to a complex knowledge assertion determining the information known to a party interacting with the contract. This is similar to the verification engine provided by Reach, a blockchain development platform that allows developers to write smart contracts for multiple platforms at once. However, for A1, the proposed verification engine will be built into the compilers itself, thus not requiring developers to learn a non-native language that they may be unfamiliar with. Furthermore, the compiler also supports the use of direct tests; developers should be able to write exploit scripts to test their own programs. Much like how QuickCheck enabled programs written in Haskell to be a lot more secure, A1's verification software will enable developers to easily write secure smart contracts on the Adamnite network. Making verification a key part of every program will allow for a streamlined and secure development process that will hopefully help create DApps that are less prone to exploits.

3.2 Storage and Execution

Adamnite utilizes a unique storage mechanism, as described earlier, to store information for autonomous accounts. While the core blockchain utilizes a binary merkle trie scheme (essentially the same as that described by Buterin and Ballet in EIP-3102 for storing account and state information), an off-chain database stores both the actual code and data for autonomous accounts. A hash (presumably computed using a secure one-way hash function) is then stored on the actual blockchain. This scheme helps Adamnite in two ways: it increases the efficiency of on-chain code execution (state changes due to interactions with code are uploaded in batches to the actual blockchain) and helps Adamnite to be more stateless by decreasing the size of the ledger that must be stored by clients and nodes, thus reducing the barrier for individuals to download a copy of the chain and thereby independently verifying the current state of the blockchain. A similar two-tier architecture has been proposed in Algorand, and implemented in Polkadot.

3.2.1 Structure

The binary merkle trie used to store the actual distributed ledger is a mapping between key values and their hashes, and mostly follows the standard structure. The off-chain database used to store contract code follows a more traditional key structure, mapping account addresses, files containing code, and storage. This works seamlessly with the WASM structure for the ADVM, as code and associated meta data can be easily uploaded to the database without the need for a complex deployment process. We assume that the off chain database can easily access state information of the actual blockchain, and the existence of clients/nodes to validate the contents of the database between various code executions and transactions. We also assume the existence of different types of clients designed to store the database itself and verify its contents. The binary trie itself stores account balances, the nonce, and for autonomous accounts, the hash of the code and storage. This is in stark contrast to the structure used by Ethereum and other EVM spinoffs, which often use multiple trie structures to store account information, leading to increased requirements for actually running nodes to verify the blockchain.

3.2.2 Contract Creation and Calls

The execution of on-chain code, or message calls, are handled through the off-chain database. We define a separate set of validators, elected through the same DPOS election used to select witnesses for the core blockchain. These validators not only validate any changes made to the database itself, but also execute calls made to autonomous accounts by executing the underlying code, and handle the creation of autonomous accounts. Unlike the core blockchain, the execution of code is optimized for speed rather than security or liveliness. A single leader is chosen among the group of validators to execute all code executions for the round. This leader produces a batch of state transitions within a certain period of time (equivalent to the execution of 10 blocks, for example) and optimizes for throughput.

Consensus on changes to storage and balances on the actual blockchain as a result of code execution is handled using a Proof of History (POH) scheme that is similar to Solana's implementation. The leader executes both the uploading of new code and calls to already existing code, proposes a set of transactions/changes to both the database and on-chain state, produces a cryptographic proof for each change attesting that time has passed, and communicates the set to the rest of the validators, who then check it. Because of the usage of POH, validators can check individual changes in the larger set, thus allowing the entire committee to reach consensus on a set of state changes much faster. Once a majority (2/3) of validators have approved the changes, they are submitted as an all or nothing batch to the current set of witnesses for the core blockchain, who then check it independently before updating the state. It is worth noting that the witnesses for the core blockchain will simply need to validate state-related checks (if any regular transactions have resulted in the one of the message executions being rendered invalid). We provide an example describing the liveness of a native on-chain application, from its creation to use by an user.

1. Developer writes application logic in the form of an A1 smart contract, and compiles it to ADM code (a WASM file, along with associated metadata).

2. Developer uploads ADM code to the off-chain database, thus creating a new autonomous account.

3. Off-chain validator certifies the creation of a new autonomous account, creates a new account in the database with a storage key based on the metadata and a code key based on the code. The autonomous account is created on the core blockchain after approval.

4. A user interacts with the autonomous account, thus submitting a transaction or message call.

5. The off-chain validators certify that the user is able to complete the execution by cross referencing with both the user's on-chain balance and the storage contents of the autonomous account in the database.

6. The off-chain validators create a state check that includes the user's required balance for the witnesses of the core blockchain, and submit it as part of an all or nothing batch.

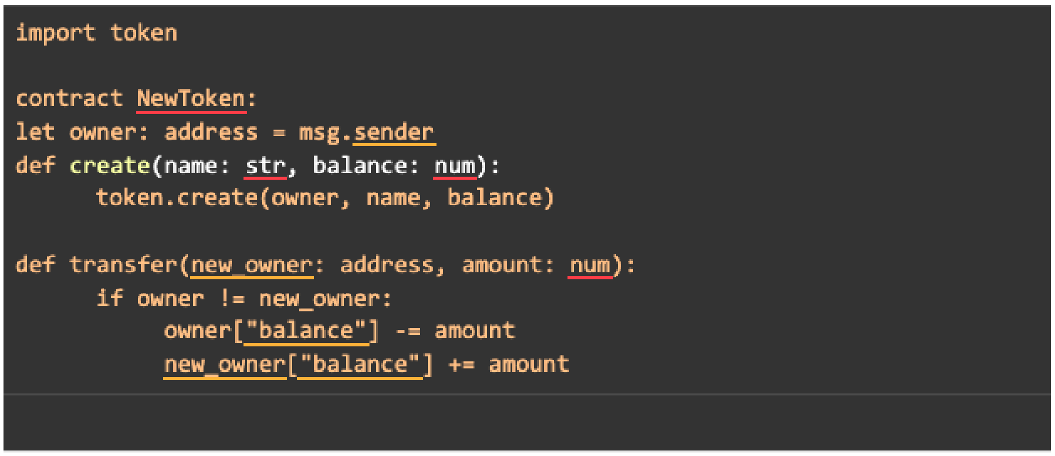
Ultimately, code execution in Adamnite is optimized for speed and efficiency, while base-level payments and consensus is optimized for liveliness and security. This allows for the creation of an ecosystem in which both high-throughput applications such as games and streaming applications can exist, without compromising the safety of base-level transactions that make up the integrity of the underlying system.

3.3 Potential Use Cases

We now turn to a discussion of potential use-cases for the Adamnite platform, centered around its efficiency and ease of use.

3.3.1 Asset Creation

Assets built on top of existing blockchains have grown rapidly in popularity, and now make up a significant portion of the cryptocurrency market. These assets can represent equity stake in a physical company, have some form of unique utility, or be used for governance in a DAO. Sub-Tokens, as they are often called, often serve to support or tokenize a different use case. On Adamnite, a new asset can easily be built directly onto the blockchain. The creation of an Adamnite-Sub-Token simply involves encoding standard token parameters such as the name, amount, and the address of the creator. Due to A1's modular structure, a developer can simply import a predefined token module and encode . An example of what Adamnite hopes to achieve with asset creation is given below in A1:



```
import token

contract NewToken:
let owner: address = msg.sender
def create(name: str, balance: num):
      token.create(owner, name, balance)

def transfer(new_owner: address, amount: num):
      if owner != new_owner:
            owner["balance"] -= amount
            new_owner["balance"] += amount
```

Fig. 3.2 A1 Token

3.3.2 Decentralized Finance

Adamnite's accounts could easily be leveraged to create smart contracts that allow for autonomous transactions to be processed directly on-chain. These smart contracts can be used to create DeFi applications that only send transactions based on certain requirements being met. For example, this could be used to create an exchange, where the transfer of an on-chain asset to an autonomous account results in that account paying out the equivalent value in a separate asset. Due to Adamnite's storage capabilities, a wide variety of data can be used for analysis in smart contracts, thus allowing for the creation of more diverse DeFi Applications. An example will be a banking application that lends assets as collateral, and establishes a credit history for a particular account based on its past transactions.

3.3.3 Decentralized Streaming

Decentralized Streaming has a variety of applications, from setting up communication hubs directly on the blockchain to enabling peer to peer communication without dependence on a centralized party. There have been implementations of decentralized streaming, such as LivePeer, that have seen success in recent years. On Adamnite, decentralized streaming applications can be easily built due to the modularity of Adamnite's underlying programming language. Developers will be able to leverage precompiled contracts to handle data storage and encoding Furthermlore, Adamnite's speed will also help these applications; streaming DApps that leverage Adamnite should have the same efficiency as their Web2 counterparts. Developers can also create a system in which data miners (not to be confused with on-chain validators) are rewarded for transcoding videos and "renting storage" for the purpose.

3.3.4 DAOs

A DAO, in its most simplest form, is a group of people who come together to make decisions regarding the rules or structure of an organization. Decisions are often made through votes through a standard governance process, with a computer program automatically making changes based on the outcome. Rules and decisions could extend beyond code; many DAOs make decisions on the allocation of internal capital, investments, and more. The key to implementing a basic DAO structure is to have mutable code that depends on the consensus reached by the members of an organization, and a basic governance model for determining said consensus. Governance models have also gained popularity in the blockchain community, with decentralized organizations and some blockchain networks using governance to enable the wider community to reach consensus on a particular proposal. A simple DAO or governance model can be implemented on Adamnite as follows:

1. Create separate blocks of code that only activate based on certain logical parameters. For example, there could be a block of code that is meant to transfer X amount of an asset from one account to another, and only executes when a satisfactory number of voters choose to approve the transaction.

2. Create smart contracts that define a new proposal and allow constituents to vote on whether or not it should be implemented. The actual creation of the smart contract is flexible, and can be done in multiple ways. A simple method is to simply create autonomous Adamnite Addresses representing each decision, and have these addresses register votes either through the transfer of an asset or the manipulation of some other on-chain data.

3. Implement the correct block of code based on the decision reached by the constituents.

Adamnite can also be used for governance methods by any community, thus allowing any group of people to make decisions directly on the blockchain.

Chapter 4

Conclusion

We now turn to a discussion of arbitrary research goals that Adamnite hopes to solve, either pre or post launch. These goals, for now, are centered around accessibility and privacy.

4.1 Looking Ahead

Making Adamnite more stateless or enabling clients/nodes intended for verifying Adamnite to act in a stateless manner, is a key future goal. Due to Adamnite's use of a binary merkle trie, the computational requirements for a node to validate the canoncial history of transactions is already lower than most of its counterparts. However, as more and more applications and users utilize the network, these requirements will increase over time. While alternative solutions such as moving to verkle tries (trie structures that use vector commitments rather than hashing) reduce the requirements for maintaining light clients/simple verification nodes, they don't reduce the size of the actual state. We plan to leverage the usage of zero knowledge proofs (ZKPs) to create a succinct blockchain that allows a validator to validate the latest block in order to validate the entire history of the blockchain. This will further reduce the barriers required for individuals to independently validate the network. It will also greatly decrease the overall size of the blockchain, thus making it more efficient from a computational perspective. A similar solution utilizing ZKPs has been implemented in the MINA blockchain, whose entire size is reported to be only 22kb. A more immediate way to reduce state is to have an expiration date for transactions submitted to the network, and to prune empty accounts with no balance or storage. A similar solution has been proposed by the authors of Vault, who theorized that introducing an expiration date for transactions and getting rid of unused accounts could greatly increase efficiency in a cryptocurrency network. Reducing the size of on-chain state also ultimately helps both developers and users to be censorship resistant: by verifying and connecting to the blockchain independently, participants in the network are less reliant on large-scale node operators who may act in their own self-interest.

Future security, especially for on-chain smart contracts, is another key research objective. While formal verification in the form of assertions helps prevent rudimentary errors, more complex attacks, especially those that are carried out by attackers who create an attack contract for the sole purpose of exploiting distributed logic within an external decentralized application. Within A1, we plan on solving this through the usage of distributed promises, a component of distributed computing originally found in the E Programming Language. An example of a promise is a "when do" statement, which is essentially a logical check that some action or transfer was completed before executing more logic. This can help prevent smart contract errors that arise from an exploit in logic rather than incorrect code. A prime example of this type of attack is reentrancy, a type of exploit where an attacker constantly calls a withdraw function on some type of staking contract, resulting in inaccurate payouts that allows the attacker to withdraw the funds locked in the contract over time. In recent years, more complex attacks with multiple participants have come into play, with more and more funds being lost due to smart contract exploits every year. Establishing a programming standard that prioritizes safety through the use of promises will greatly help in reducing these exploits.

4.2 Miscellaneous

The following is a discussion on several extraneous topics or questions that are separate from the core tenets of the network. Some subsections such as a defense of the underlying consensus protocol, are meant to answer potential questions, while others, such as the discussion on underlying economics, are meant to be a primer for non-technical decisions that nevertheless impact the protocol. For a more in-depth technical clarification on particular topics (such as the data storage mechnaism), more recent literature (such as the technical paper) should be consulted.

4.2.1 Why DPOS?

DPOS as a consensus mechanism is perceived to have two common flaws: security and centralization. These issues are hand-in-hand; critics of DPOS point that there must be active engagement in the voting process in order to prevent the same accounts from being selected repeatedly as witnesses, which therefore leads to centralization and security concerns, as a malicious attacker will be able to pinpoint the witnesses. Adamnite solves this problem in two ways. First, by introducing incentives for voting, Adamnite ensures that participants will be more likely to participate in the voting process. Second, by leveraging cryptographic sorition, Adamnite ensures that the pool of witnesses for a particular block is semi-random, with the initial pool of the highest vote earners being the only public information.

DPOS was specifically chosen as a consensus mechanism because of its speed, security, and reliability. DPOS has been shown to be faster and more efficient than both POW and POS, primarily because it utilizes a small number of witnesses to validate new blocks, rather than requiring a proof of computational power or consensus among a large number of addresses. Furthermore, DPOS is decentralized, as every token holder has the opportunity to participate in voting. Finally, witnesses have a large incentive to act honestly, as they could lose both their position (and their potential for earning validation rewards) at any time. Ultimately, Adamnite's DPOS consensus mechanism will help make it more scalable: by concentrating both block proposal and block validation in the hands of a few democratically elected validators, Adamnite will be faster than other chains while retaining a certain aspect of decentralization.

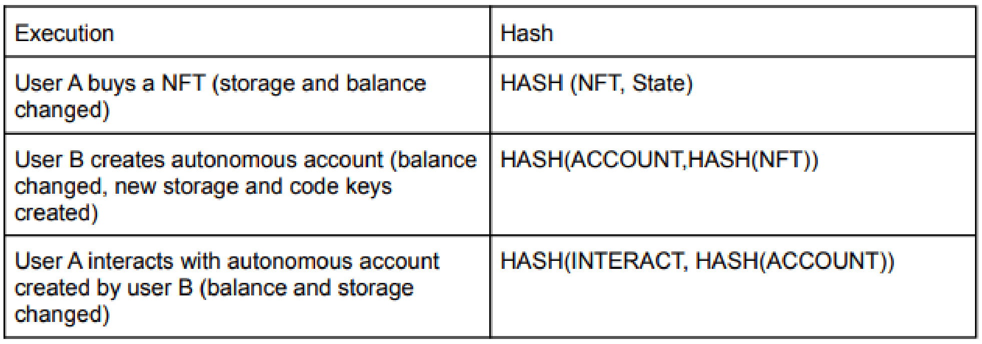
4.2.2 Governance

Adamnite plans to use governance as a way for participants to both vote on delegates and on proposals. Initially, governance in Adamnite will follow a traditional coin-vesting scheme, similar to the governance mechanisms utilized by Cardano and Algorand. Participants will be asked to lock a certain amount of NITE in order to vote on a proposal or on delegates. They will then be able to vote for a certain option. The option with the highest amount of NITE dedicated to it will be the one that is implemented. In the case of delegates, the top m addresses will be chosen, with m again depending on the total number of participants. This is ultimately a coin-voting mechanism, as an individual participant will be able to lock a larger amount of NITE to have a larger say. Although coin-voting does cause a certain degree of centralization, it is currently the most thought-out and widely implemented solution. In the future, Adamnite will likely transition to a different governance mechanism such as proof-of-participation in order to ensure both decentralization and security.

Governance proposals could range from a simple adjustment in the rewards structure to funding for on-chain development. Delegates will ultimately be in charge of approving and selecting new proposals, although anyone may create a new proposal for review. Proposals will be approved in a similar fashion to Bitcoin Improvement Proposals (BIPs), with the delegates actively engaging with both the individual who created the proposal and the wider community. The creator will then have the opportunity to revise their proposal before it is reviewed for approval. This creates an open process, which helps make the Adamnite network more decentralized.

4.2.3 Proof of History

The Proof of History (POH) protocol used to quickly reach consensus on changes to the state of the off-chain database is based on the repeated computation of a secure, one-way, and collision-resistant hash function. In this case, the leader chosen to execute code is analogous to the POH-generator in Solana's protocol. Following the execution of each state-transition that results in either a change in balance or storage for an account, the leader creates a hash of the data and signs with his private key. The usage of a cryptographic hash function allows for the other validators to prove that a certain amount of time has passed between two changes to the state; because we utilize a hash function, the change can be verified in a much shorter time than the time it takes to generate it. Hashes are ultimately appended to one another; this ensures state stability, as the validity of a future execution is ultimately dependent on the state that was derived from a previous execution. The proposed efficiency of this scheme is ultimately dependent on the limitation of execution to one efficient leader, with the other validators serving as a stamp of approval. The table below describes the execution of code and message calls, along with the generation of the corresponding POH for each change.



Execution	Hash
User A buys a NFT (storage and balance changed)	HASH (NFT, State)
User B creates autonomous account (balance changed, new storage and code keys created)	HASH(ACCOUNT,HASH(NFT))
User A interacts with autonomous account created by user B (balance and storage changed)	HASH(INTERACT, HASH(ACCOUNT))

Fig. 4.1 POH Table

The other validators can prove that a certain amount of time has passed between the computation resulting in the different states due to the amount of time it takes to compute each hash. The utilization of the previous hash creates a system in which changing the result of an execution will require changing the results of the computations before it, creating a cryptographic guarantee similar to that seen in mainstream Proof of Work chains.

4.2.4 Currency

The Adamnite blockchain will have its own built-in token, called nite. This will be used to process on-chain transaction fees, voting rewards, etc. There will also be smaller denominations of nite, analogous to cents in the USD Currency System and Satoshis in Bitcoin. As in Ethereum's initial white paper, these denominations are named after some of the most prominent contributors in cryptography and blockchain. They are defined as follows:

1:	micali
10^{10}:	sunny
10^{12}:	vitalik
10^{14}:	nite

Nite will have a permanently growing supply, with a mathematical function controlling the growth rate. The main argument for having a consistent growth rate, as opposed to fixed supply such as Bitcoin, is simply decentralization. Assets with fixed supplies are often concentrated in the hands of early adopters, thus preventing new individuals from being able to participate in the ecosystem. A growing supply ensures that the network is always ready to support new users. Token issuance will come from an account controlled by a delegated party, which could be a group of chosen delegates or an official organization with the sole purpose of growing the Adamnite network.

The growth function is also meant to control inflation. Currently, the proposed inflation rate is 10% at the minting of the genesis block, followed by a reduction to a long term inflation rate of 3%. These parameters, along with the annual reduction, can be altered by the broader community in governance proposals. Furthermore, the inflation rate is a maximum rather than an average or goal: the growth function does not account for tokens lost to burning, misplaced private keys, etc. Like the issuance models proposed by Etherem and Solana for their respective blockchains, the growth rate eventually reaches a constant that ensures that the network reaches a balance between providing economic incentivies to block producers/validators and preserving value.

4.3 Conclusion

Adamnite represents the future of blockchain development. By providing developers with the means to create fast, efficient, and secure applications, Adamnite sets the stage for a world in which blockchain technology is more widely used and preferred to legacy Web2 solutions.

Adamnite also serves to push peer to peer computing forward; Adamnite's storage capabilities and consensus system mean that individual nodes/witnesses are essentially rewarded for hosting on-chain data encoded by other developers using the platform. However, the most unique feature of Adamnite will be its development ecosystem. Developers, regardless of their prior experience with DLT or blockchain technologies, will be able to confidently build applications on Adamnite due to its intuitiveness and security. This will significantly reduce the barriers needed to start leveraging blockchain technology in day to day development.

The idea of a next-generation blockchain platform is not entirely new: protocols such as Ethereum have already made a significant contribution to blockchain adoption and innovation. Adamnite aspires to accelerate this trend by providing a platform that is at once more efficient, safer and easier to use than current solutions. Adamnite will be at the forefront of a world in which blockchain technology is used to its full potential, allowing anyone, anywhere to build powerful applications with the power of decentralization and distributed computing.