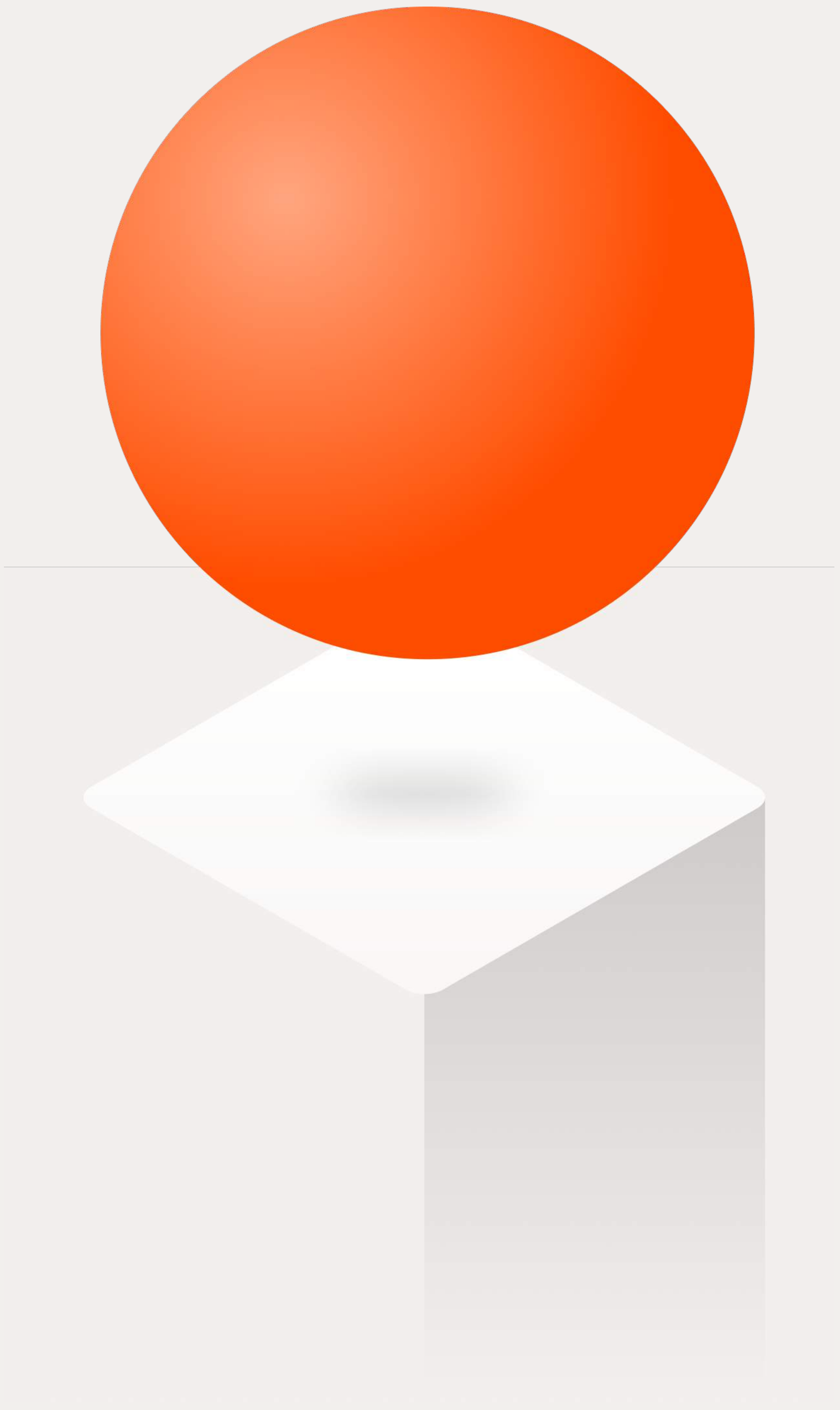
Archway

Technical Paper



TECHNICAL OVERVIEW

May 2023

DISCLAIMER

The material provided herein is for informational purposes only. It does not constitute an offer to sell or a solicitation of an offer to buy any interests in any other securities. Certain statements herein may constitute forward-looking statements. When used herein, the words “may,” “will,” “should,” “project,” “anticipate,” “believe,” “estimate,” “intend,” “expect,” “continue,” and similar expressions or the negatives thereof are generally intended to identify forward-looking statements. Such forward-looking statements, including the intended actions and performance objectives of Archway involve known and unknown risks, uncertainties, and other important factors that could cause the actual results, performance, or achievements of Archway in its development of the system, network, its components, and the tokens to differ materially from any future results, performance, or achievements expressed or implied by such forward-looking statements. No representation or warranty is made as to future performance or such forward-looking statements. All forward-looking statements herein speak only as of the date hereof. Archway expressly disclaims any obligation or undertaking to disseminate any updates or revisions to any forward-looking statement contained herein to reflect any change in its expectation with regard thereto or any change in events, conditions, or circumstances on which any such statement is based. You are not to construe this light paper as investment, legal, tax, regulatory, financial, accounting or other advice, and this light paper is not intended to provide the basis for any evaluation of an investment in an interest. The ARCH token will not be offered in the United States or to U.S. persons or to residents of certain other prohibited jurisdictions. Learn more here.
The information provided in this document is for general informational purposes only. It does not constitute, and should not be considered, a formal offer to sell or a solicitation of an offer to buy any security in any jurisdiction, legal advice, investment advice, or tax advice. If you are in need of legal advice, investment advice or tax advice, please consult with a professional adviser. The Archway protocol is under development and is subject to change. As such, the protocol documentation and contents of this website may not reflect the current state of the protocol at any given time. The protocol documentation, document, and website content are not final and are subject to change.

TABLE OF CONTENTS

ARCHWAY OVERVIEW

Archway is a Layer 1 Cosmos-SDK blockchain that enables decentralized application (dapp)[1] developers to capture the value they create for an underlying network.

The protocol distributes tokens to dapps based on the volume of activity they bring to the network, sharing a portion of network fees and newly minted tokens with validators and stakers. The network also allows developers to add premiums to their smart contracts.

Archway provides cross-language support for smart contracts with WebAssembly (WASM) and native bridges to other networks. Cross-chain communication and asset transfers are powered by the Inter-Blockchain Communication protocol (IBC) from the Cosmos SDK. Archway developers can deploy contracts to an established Proof-of-Stake (PoS) network, connecting them to users, assets, and communities from across the Cosmos ecosystem, and providing a viable alternative to building and maintaining an application-specific blockchain.

This document outlines the technical architecture of Archway. Archway includes customized SDK modules for Minting, CosmWasm smart contracts, Distribution, Staking, and Governance, as well as default Cosmos SDK modules for managing inflation and rewards. Since Archway is built on the Cosmos SDK, the Cosmos technology and modules will also be explained.

[1] Archway Lightpaper: An Introduction to Archway. Archway.io. https://archway.io/assets/Archway-Lightpaper.pdf

ARCHWAY ARCHITECTURE

Cosmos is a network of parallel, independent blockchains. Its SDK enables scalable, application-specific blockchains with Byzantine fault tolerant (BFT) consensus, that can interoperate with other independent blockchains in the Cosmos ecosystem.
Cosmos promotes the idea of self-sovereign blockchains, but in many cases, it makes sense to deploy a project first as a dapp. Launching and maintaining a blockchain requires infrastructure and resources. The Cosmos SDK simplifies this process, but developers must still incentivize a community of validators to secure their network. These efforts can be daunting for many early stage projects. Below is a comparison of building an application-specific blockchain versus building smart contract applications.

App Chain vs. Dapp

Some key differences between app chains and smart contracts are: customization level, performance, use cases, and technical requirements.

Developing an application-specific blockchain allows for complex, full stack customizations (e.g. Archway's x/rewards calculates every unit of gas used in contract executions). While they are generally more performant, because the native application-specific chain runs as a binary, there can be a lot of overhead in execution for smart contract code as it must always be loaded into a Virtual Machine.

App chains support a single purpose within their protocol (e.g. Stargaze is an application specific blockchain created to operate a decentralized NFT marketplace). Therefore, a potential hurdle for app chains is that they require deeper technical development and resources than developing smart contracts.

Cosmos blockchains are comprised of three core components: the Cosmos Software Development Kit (Cosmos-SDK), Tendermint Core and the Application Blockchain Interface (ABCI).

Cosmos SDK

The Cosmos SDK is a software development kit for building sovereign blockchains.[2] It is an open-source framework that allows developers to build custom blockchains that can natively interoperate with other Cosmos SDK blockchains.

Blockchains built with Cosmos SDK are commonly referred to as application-specific blockchains, or zones. Archway is a Cosmos SDK blockchain.

Tendermint Core

A blockchain can be divided into three layers: networking, consensus, and application. Before Tendermint was created, building a blockchain required developing all three layers, which represented a considerable amount of work.

The goal of Tendermint is to aid development by providing a standard engine to power the networking and consensus layers. With Tendermint, Developers only need worry about the application layer, over which they have full control. The interface that connects the state machine (application) to the underlying consensus engine is a socket protocol called Tendermint ABCI.

Tendermint is a Byzantine fault tolerant (BFT) consensus engine. Byzantine fault tolerance in distributed systems means messages can be broadcast between all parties, and reliable agreements can be formed, even when there are dishonest and adversarial parties involved. Tendermint is widely considered a gold standard for BFT consensus and Proof-of-Stake (PoS) blockchains.

A Tendermint blockchain is operated and maintained by a set of nodes. Nodes are the agents of a blockchain, and can be divided into two main categories: Validator nodes (the ones that decide on the next block), and non-validator nodes (the ones that can only execute blocks proposed by the validators).

A Tendermint blockchain can be divided into two main components: **the Consensus Engine** and **the State Machine**.

[2] High-level Overview. Cosmos SDK Documentation. https://docs.cosmos.network/main/intro/overview

1. Consensus Engine

The consensus engine is Tendermint Core, which handles:

a. Peer To Peer (P2P) messaging between nodes that propagates transactions, blocks, and signifies malicious behavior
b. Deciding which validator should propose the next block
c. Deciding on the ordering of transactions
d. Agreeing on a common time
e. Maintaining a mempool where transactions are stored while waiting to be included in a block

2. State Machine

The state machine is at the application layer, which handles:

a. The interpretation of state transitions (transactions)
b. Changes to the active validator set
c. How to punish validators in the case of misbehavior

Application Blockchain Interface (ABCI)

Tendermint offers a primitive abstraction layer called the ABCI, which allows developers to safely implement a blockchain.[3]

The ABCI interface defines a set of boundaries between duties of the state machine and duties of the consensus engine.

State machines define their specific application logic by implementing ABCI interface methods.

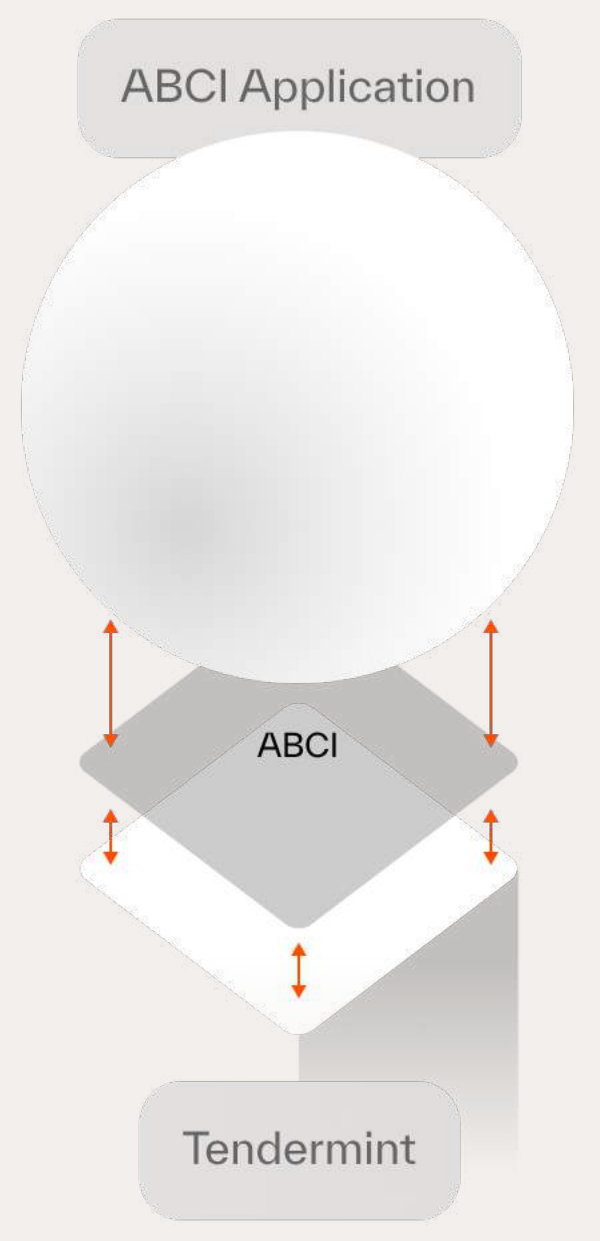


Figure 1

[3] Application Blockchain Interface. Tendermint. 2022. https://github.com/tendermint/tendermint/tree/main/abci

ABCI Interface Methods

The ABCI provides methods for procedures to be executed by Tendermint during the transition of state. Each Cosmos SDK module must define specific procedures to be compliant with the ABCI. Cosmos blockchains require implementing the following ABCI interface methods: **CheckTx**, **BeginBlock**, **DeliverTx**, **EndBlock**, and **Commit**.

CheckTx

CheckTx is a phase that acts as a guard for the node mempool. It signals to the state machine that a new transaction would like to be part of the mempool, and the state machine tells Tendermint Core to either insert the transaction into the mempool or reject it.

BeginBlock

BeginBlock signals to the state machine that a new block was proposed. It contains useful information for the state machine, such as agreed consensus time (BFT time), evidence of misbehavior of validators (if any exists), and the current block height.

DeliverTx

DeliverTx signals to the state machine to execute a specific transaction contained in a block. After the transaction is executed, the state machine reports information back to Tendermint Core, such as gas consumed, whether or not it was successful, and Events (e.g., objects containing metadata such as the type of Cosmos message that was executed).

EndBlock

EndBlock signals to the state machine that the execution of a block's transactions has finished. The state machine can optionally return a change to the validator set, or new consensus parameters (e.g., update the gas limit for future blocks).

Commit

During this phase, Tendermint signals to the application that computations which have occurred thus far need to be committed to state. The application produces a hash which represents the current state of the state machine. In order for **Commit** to succeed, all nodes must return the same hash value to ensure that state machine replication was successful. Enforcing the equivalency of hashes of the current state, across all nodes, signals successful determinism.

Modules

ABCI Interface Methods are implemented via **Modules** which contain the business logic of a Cosmos SDK blockchain, and define how the blockchain operates and interacts with other Cosmos chains. **Modules** can work together and execute at different phases of the ABCI interface process. They provide a less-primitive way of defining state machine functionality, and can either be default (e.g. Cosmos SDK modules) or custom (e.g. Archway modules).

Interacting With the ABCI Interface

In the following sections, we will analyze the anatomy of a Cosmos SDK module and highlight how each component interacts with the ABCI interface. The components we will analyze are: **BeginBlock**, **RunMsg**, **CheckTx**, **DeliverTx**, **EndBlock**, **AnteHandlers**, and **custom components**.

BeginBlock

Every module can define actions to perform during **BeginBlock**, which occurs before any transaction is processed for a block. A module implementing **BeginBlock** can perform preliminary actions such as minting tokens **(x/minting)**, tracking gas consumption **(x/tracking)**, and keeping track of validator misbehavior so that dishonest validators can be penalized later **(x/evidence)**. Developers can define the order of a module's **BeginBlock** functions. This set of functions is then executed together, in order, when Tendermint signals to the blockchain that it should perform **BeginBlock** actions.

RunMsg

The **RunMsg** procedure verifies that a Cosmos message has been properly registered for an application (e.g., by a module).

CheckTx

During **CheckTx**, Cosmos SDK creates a cache of the current state and simulates execution of the transaction. Each transaction's messages are forwarded to the module which handles that message type (e.g., **x/bank** handles **MsgSendCoins**, **x/staking** handles **MsgDelegate**, etc.).
If the execution is successful, Cosmos SDK reports to the Tendermint Core that the transaction is valid and can be included in the mempool. The created cache is then eliminated and discarded, and no state changes are applied yet.

DeliverTx

If **DeliverTx** is executing, it means that a transaction that was already in the mempool and has passed **CheckTx** verification. During **DeliverTx**, Cosmos SDK creates a cache of the current state. Execution then follows a similar verification process to **CheckTx**. If a transaction is successful it gets committed to the current state. If a transaction fails, the block will report it as failed. Any state changes made in failed transactions will be reverted to their original state.

EndBlock

Every module can define actions to perform during **EndBlock**. These actions will be performed after all the transactions of a block are executed. Some examples of **EndBlock** interactions can be found in the **x/staking** module, which returns changes to the validator set after it processes the new delegations, and Archway's **x/rewards** module, which processes rewards to be distributed to smart contracts.
The Cosmos SDK allows developers to define the order of each module's **EndBlock** functions. This set of functions is executed together, in order, when Tendermint Core signals to the Cosmos SDK blockchain that it should perform **EndBlock** actions.

AnteHandlers

AnteHandlers are functions that run after a transaction is decoded into a set of messages and authentication data, before the **RunMsg** phase. Multiple **AnteHandlers** are chained together by the Cosmos SDK into a single one, which runs each individual AnteHandler sequentially as defined by the developer. AnteHandlers can perform preliminary actions, but they don't execute messages. Examples of **AnteHandlers** can be found in the **x/auth** module, which verifies authentication data of a transaction, and Archway's **x/rewards** module, which splits transaction fees between validators, stakers, and smart contracts.

Custom components

Module methods are extensible and can be used to define custom components. Some examples of custom methods could be: a Genesis Import and Export feature; which modules implement to export or import the state into a human readable format (e.g. JSON); or, a Migrations feature wherein modules can define specific ways to migrate their state from one version to another during chain upgrades.

Anatomy of a Cosmos SDK transaction

A Cosmos SDK transaction is a set of bytes encoded using Protobuf, which can be decoded into a set of messages and authentication data. The Cosmos SDK allows every module to define a set of messages and handlers for those messages. A transaction message defines a state transition which changes the state based on the module's specific logic.

Examples:

1. **x/bank MsgSendCoins** allows users to move coins and tokens from one account to another
2. **x/staking MsgDelegate** allows users to delegate tokens to a validator
3. Archway's **x/rewards WithdrawRewards** allows users and contracts to withdraw their accrued dapp incentives

Relationship between RunMsg, DeliverTx and CheckTx

RunMsg is run during both **DeliverTx** and **CheckTx**.

The transaction bytes provided by Tendermint during **DeliverTx** and **CheckTx** are converted into a set of messages (**Msgs**) which are mapped into the modules' message handlers.

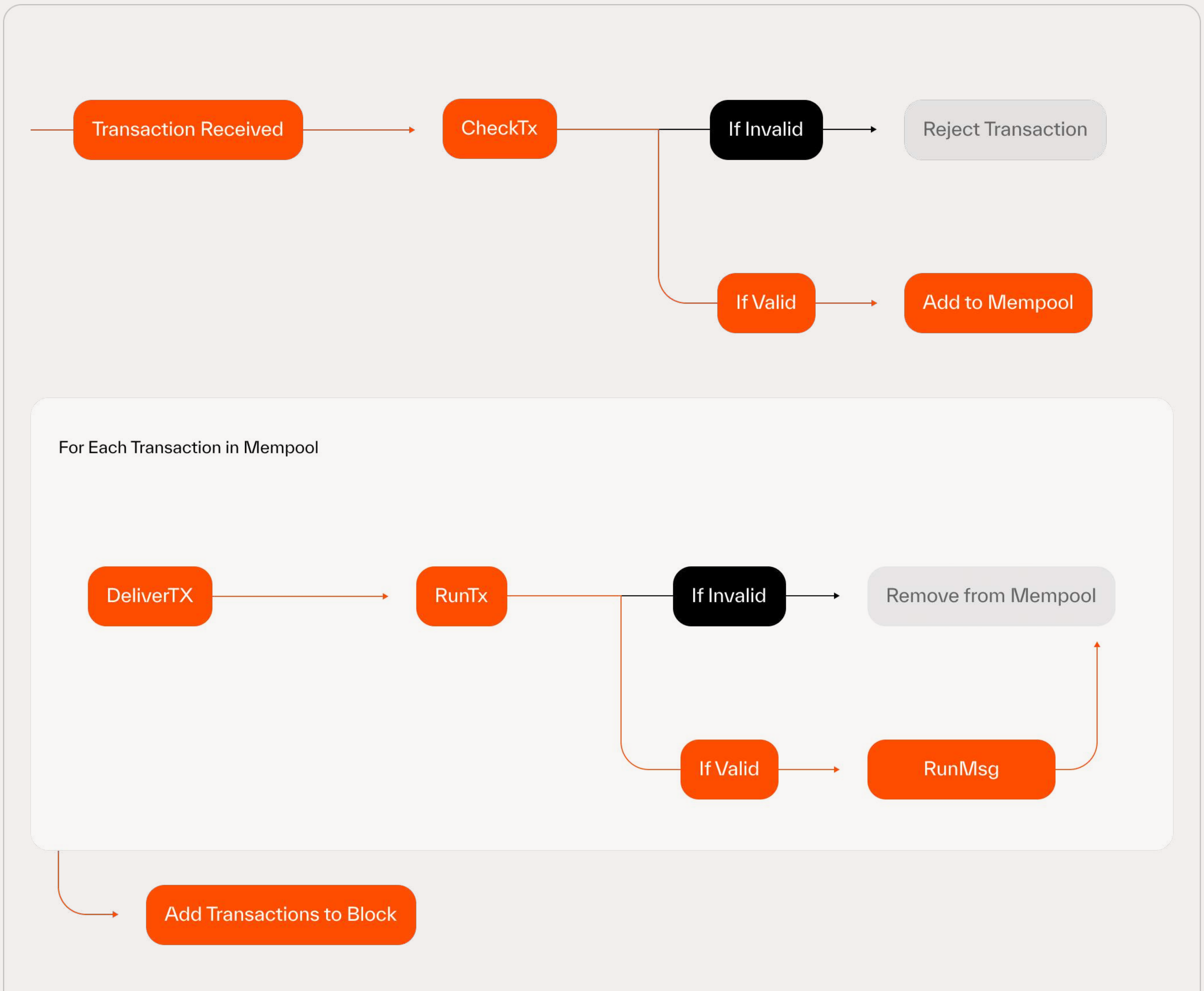


Figure 2

THE ARCHWAY MODULES

The Cosmos SDK comes with predefined modules for core blockchain functionality and Inter-Blockchain communication. These modules provide the basic set of functionalities of a blockchain (e.g. transaction authentication, Proof-of-Stake, accounting).

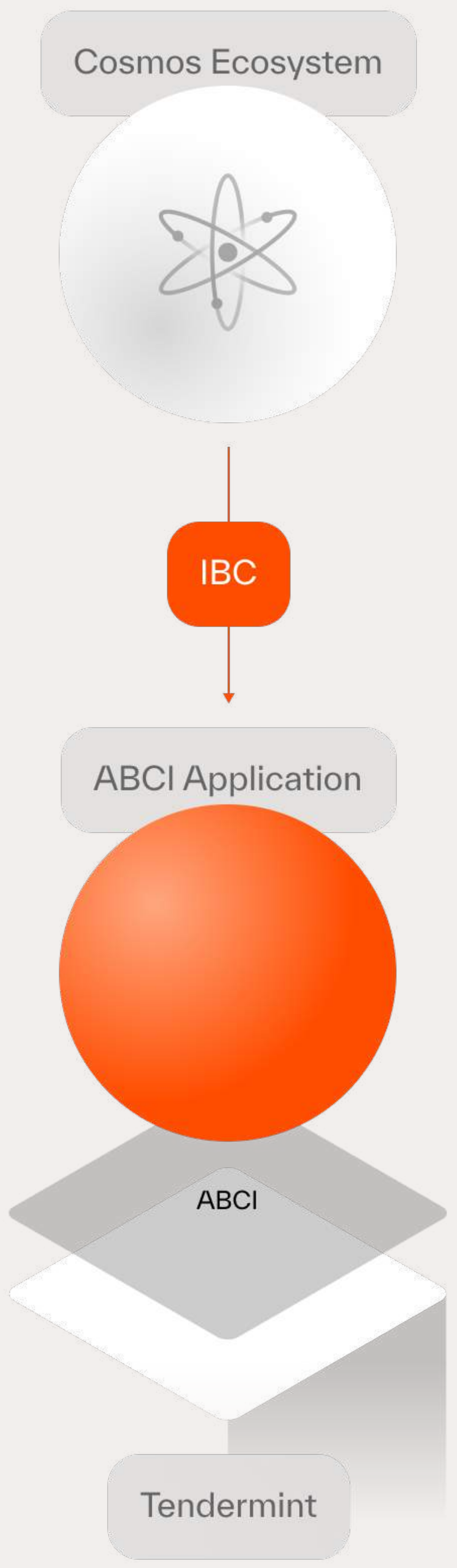


Figure 3

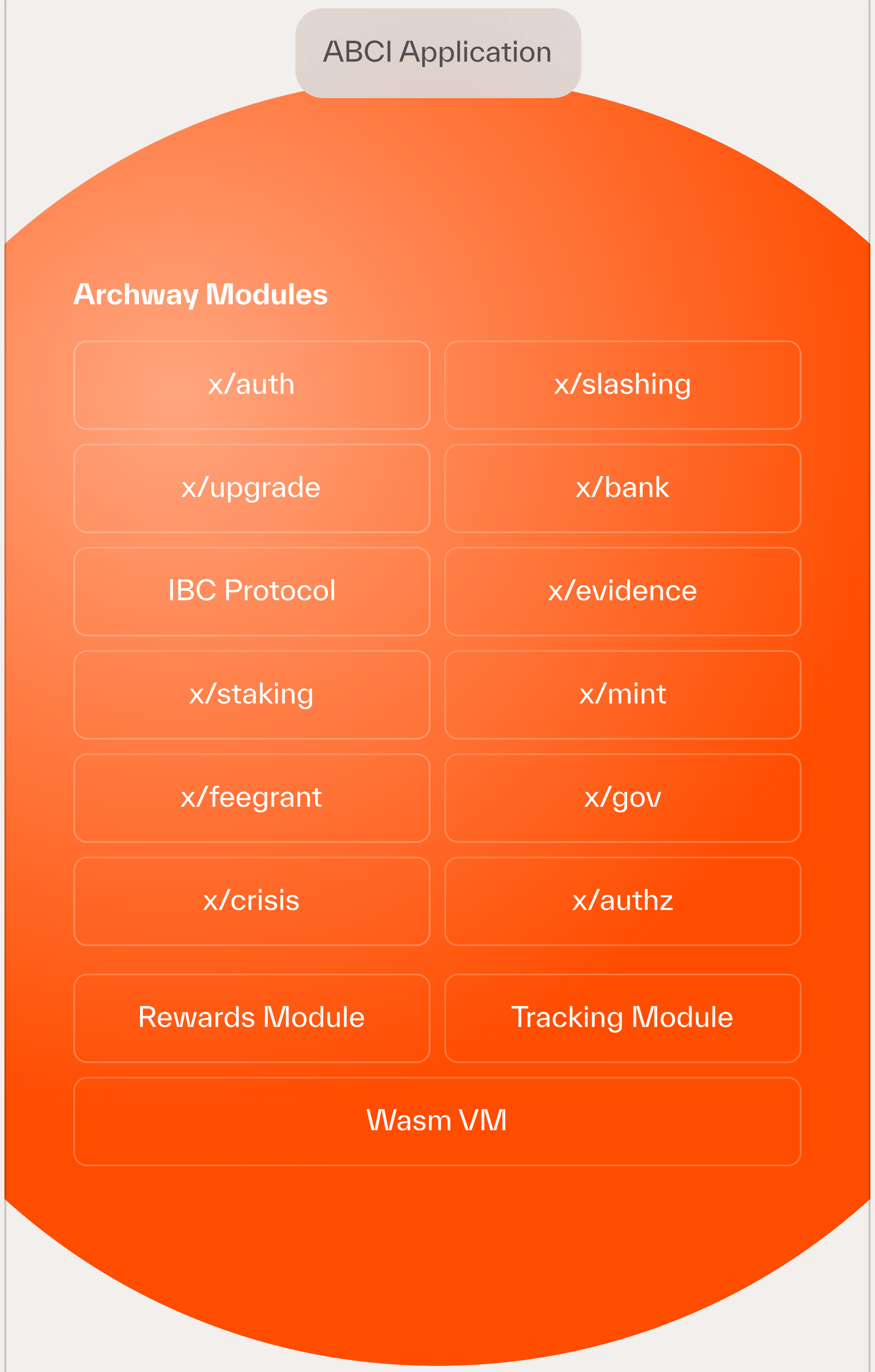


Figure 4

Archway's economical model depends on custom modules that extend the Cosmos SDK with functionality for disbursing ARCH tokens and tracking the gas consumption of smart contracts. This section explains both the Cosmos SDK modules that are included in Archway, as well as the custom ones. The SDK modules included are: Auth, Bank, Staking, Slashing, Minting, Governance, Crisis, Upgrade, Evidence, IBC, CosmWasm, Authorization and Grant (authz), and Feegrant.

The custom modules included are x/rewards and x/tracking.

Auth Module

The **x/auth** module handles accounts and authentication.[4]

Accounts

Accounts define identities in an abstract way. The identity is defined through what we call an address. Modules, smart contracts, and external users can have an account. External users have accounts mapped to public keys which are used for signature verification that ensures the sender of a transaction is the actual owner and not some impersonator. Modules and smart contracts can have accounts too, but they do not have public keys— instead, they only have an address which will never map to a public key. **x/auth**'s duty is to specify the transaction format and different account types for an application, as Cosmos SDK modules are agnostic regarding these formats. **x/auth** contains multiple **AnteHandlers** such as signature verification and fee deduction. Signature verification validates user signatures in a transaction. Fee deduction ensures the account sending the transaction has the funds to pay for it, and it also forwards fees to the Fee Collector module account. Accounts also provide access to other modules to read and modify accounts.

Fee Collector

Fee Collector is a special type of module account defined in x/auth as part of the Cosmos SDK design. It stores network fees from transactions to disburse them to validators. In the Archway protocol, fees stored by the Fee Collector are also a source of developer rewards (e.g. disbursed by x/rewards).

Bank Module

The **x/bank** module handles the blockchain's accounting. Each account is mapped from the **x/auth** module to the **x/bank** module with an associated balance. **x/bank** tracks and provides query support for account balances.[5]

x/bank handles moving assets in a safe way between different accounts. Moving assets is not restricted to transferring funds between accounts: it also includes actions such as delegations and undelegations. Vested accounts cannot move vested funds until the vesting criteria is met; however, they can still delegate, undelegate, and spend the staking rewards of vested funds.

[4] Cosmos SDK Auth Module. 2022. https://github.com/cosmos/cosmos-sdk/blob/2418c3ef2e6f74fd6e7575b743fc1da4b53ab972/x/auth/README.md

[5] Cosmos SDK Bank Module. 2022. https://github.com/cosmos/cosmos-sdk/blob/2418c3ef2e6f74fd6e7575b743fc1da4b53ab972/x/bank/README.md

x/bank also handles accounting for coins. Note that coins, in this context, are not the same thing as tokens. The term tokens refers to the native payment token of the SDK blockchain, but the term coins refers to assets created and managed on the blockchain by smart contracts. The difference between a coin (e.g. cw20) and a token (e.g. ARCH) in Cosmos SDK blockchains is similar to the distinction between Ether (ETH) and ERC-20 tokens on Ethereum.

Staking Module

The **x/staking** module manages changes to the validator set using Proof-of-Stake.[6]

Proof-of-Stake (PoS)

Proof-of-Stake is a model for participants to stake using tokens. The staked tokens act as collateral that can be slashed in the case of validator misbehavior. Staked tokens are commonly referred to as "bonded" tokens. Cosmos SDK uses this bonded stake to determine the active set of validators. **x/staking** tracks the supply of tokens bonded to the security of the network as well as the supply of unbonded tokens.

Validators and Delegators

There are two types of stakers:

1. Validators
2. Delegators

Validators are stakers with tokens bonded to the network: They are running a full node that participates in consensus. There is a limited set of validators that are selected, based on their stake in the network, to ensure the security of it.

Delegators are stakers with tokens bonded to an existing validator. These stakers are helping the validator be selected to sign blocks, known as the active set.

Both validators and delegators have the ability to unbond their tokens, but are subject to an unbonding period, during which they remain liable for misbehaviors committed prior to the completion of the unbonding transaction.

Notably, delegating to a validator enables a shared reward-risk scenario: The validator shares rewards with delegators, but if the validator misbehaves, the delegator will also be penalized.

[6] Cosmos SDK Staking Module. 2022. https://github.com/cosmos/cosmos-sdk/blob/2418c3ef2e6f74fd6e7575b743fc1da4b53ab972/x/staking/README.md

Validators have three possible states:

1. Unbonded
2. Bonded
3. Unbonding

Unbonded validators are not in the active set. They will need to bond a sufficient amount of tokens before they can sign blocks and earn yield.

Bonded validators have bonded a sufficient amount of tokens. They are part of the active validator set and can sign blocks and earn yield.

Unbonding validators are in the process of leaving the active set, either by choice, or because they no longer meet the criteria to remain in the active set. Reasons for no longer meeting the criteria to remain in the active set could be due to slashing (see Slashing Module below), jailing (missing too many blocks), or tombstoning (permanently removed from the active set for offences).

x/bank also handles accounting for coins. Note that coins, in this context, are not the same thing as tokens. Tokens refers to the native payment token of the SDK blockchain, but coins refers to assets created and managed on the blockchain, by smart contracts. The difference between a coin (e.g. cw20) and a token (e.g. ARCH) in Cosmos SDK blockchains is similar to the distinction between Ether (ETH) and ERC-20 tokens on Ethereum.

Slashing Module

The **x/slashing** module disincentivizes actions which put network security at risk.[7] The potential penalties include burning a validator's stake or removing their ability to vote on consensus for a period of time. There are two categories of infractions which are penalized: non-malicious protocol faults and liveness faults.

Non-malicious protocol faults

Non-malicious protocol faults could be due to misconfiguration or a faulty protocol update. To mitigate the impact of these faults, validators may get their funds slashed. However, there is a limit to how many infractions they will be penalized for. Validators receive slashing penalties only for the first infraction. Afterwards, they are tombstoned. Since more infractions may be found even while the node is still in jail, the validator must wait until the jail period expires. When the jail period is over, the node may rejoin the network by unjailing themselves, but will have to pay for the highest infraction reported while the node was in jail (this period of time is known as the slashing period).

[7] Cosmos SDK Slashing Module. 2022. https://github.com/cosmos/cosmos-sdk/blob/df4d6d1a4cd9fa0247f9db9378db857d95a1c1cb/x/slashing/README.md

Liveness faults

Liveness infractions can happen when a validator is not ready to verify a block. Contrary to non-malicious protocol faults, penalties for liveness faults are not capped since they cannot stack upon each other. Liveness faults are detected when the infraction occurs, and the misbehaving validators are immediately put in jail so that it is not possible for them to commit multiple liveness faults without unjailing themselves first.

Minting Module

The **x/mint** module is responsible for minting tokens and for managing inflation.[8] To incentivize stakers, the total supply of staking tokens needs to grow with inflation. **x/mint** allows customization strategies over the expansion of token supply, but the standard procedure is that the minting of tokens dynamically reduces or increases due to inflation every block. The goal is to keep a desirable balance between staked and liquid tokens.

Example:

1. Assume the total of staked tokens vs. liquid tokens is 50%. In this optimal scenario, inflation is 5% per year.
2. If the total amount of staked tokens drops to 45% (e.g., 55% of tokens are liquid), x/mint raises inflation to incentivize more accounts to stake until the target of 50% is reached again.
3. If the total of staked tokens rises to 55% (e.g., 45% of tokens are liquid), x/mint reduces inflation–and in consequence staking rewards–until the amount of staked tokens returns to 50%.

Governance Module

The **x/gov** module allows support for governance systems whereby token holders can vote on proposals on a one-vote-per-token basis.[9] While unbonded token holders cannot participate in governance voting; they can, however, submit proposals. Bonded participants cannot vote on proposals if they staked, or became a validator, after the proposal entered its voting period. In a scenario in which a participant has token bondings with multiple validators in the active set, earlier bondings will override later ones (e.g. in cases of voting discrepancies). The governance process is divided into three steps: Proposal Submission, Voting, and Execution.

[8] Cosmos SDK Minting Mechanism. 2022. https://github.com/cosmos/cosmos-sdk/blob/df4d6d1a4cd9fa0247f9db9378db857d95a1c1cb/x/mint/README.md

[9] Cosmos SDK Governance Module. 2022. https://github.com/cosmos/cosmos-sdk/blob/df4d6d1a4cd9fa0247f9db9378db857d95a1c1cb/x/gov/README.md

Proposal Submission

During proposal submission, a governance proposal is submitted with a deposit. Once the proposal reaches a minimum deposit requirement, it is considered submitted and the voting period can begin. The reason a minimum deposit is required is to avoid spam governance proposals.

Voting

During voting, bonded token holders can send a transaction to vote on the submitted proposal. Votes can have one of four values: **Yes**, **No**, **NoWithVeto**, or **Abstain**. The voting period is shorter than the unbonding period (~2 weeks) to avoid double voting by unbonding tokens.

Execution

After the voting period has expired, if the vote was successful and not rejected, the messages in the submitted proposal will be executed. Depending on the result of the proposal, deposits may be burned or returned to their owners. In the case of Approved or Rejected proposals (but not vetoed), deposits are returned to their respective owners. If more than 1/3rd of voters have voted **NoWithVeto**, the proposal is vetoed. In this case, deposits are burned from the governance module account and the proposal is removed from state.

Crisis Module

The **x/crisis** module allows other modules to define a set of Invariances.[10] Invariances are functions which must never fail. **x/crisis** ensures that no matter what state changes happen in the blockchain, some assertions always remain true.

Examples of invariances can be found in **x/bank** and Archway's **x/rewards** modules. **x/bank** defines an invariance check to ensure balances of accounts are never negative. **x/rewards** defines an invariance check which ensures the account balance of the **x/rewards** module is never lower than the outstanding rewards to be paid (e.g. **x/rewards** could not fully process payouts).

Invariance checks are run periodically since they can be extremely slow. A user can also manually trigger a specific invariance check by spending a specified amount of tokens (usually a large amount). If the invariance check fails, the chain goes into crisis mode and is halted. This is not ideal, but at the same time avoids further damage that could impact the blockchain. When x/crisis places a blockchain in crisis mode, human intervention is required to rescue the chain from its halted state.

[10] Cosmos SDK Crisis Module. 2022. https://github.com/cosmos/cosmos-sdk/blob/df4d6d1a4cd9fa0247f9db9378db857d95a1c1cb/x/crisis/README.md

Upgrade Module

The **x/upgrade** module administers software upgrades to a live blockchain.[11] **x/upgrade** works alongside an operating system process called Cosmovisor, which takes care of orchestrating certain parts of the upgrade such as downloading the binary designated for the upgrade and replacing the current binary. By stopping the node binary, **x/upgrade** makes it feasible to conduct software updates, thus stopping all network nodes and enforcing them to update to the binary selected for the network upgrade.

Upgrades are usually triggered by a governance vote. Once the governance vote passes, a halt height is decided. When the blockchain reaches the target halt height, the chain is suspended.

Upgrades can happen in one of the following three ways: binary replacement, binary replacement and state migration, or binary replacement and state replacement. In the first case, only the node binary changes, while the state remains unchanged. In the second case, the node binary is changed, and only a portion of the state is changed (e.g. migration). In the case of replacing both binary and state, a full state export, and then import, is performed alongside updating the node binary.

Evidence Module

The **x/evidence** module enables the submission and handling of arbitrary evidence.[12] It differs from the standard evidence handling (e.g., from Tendermint Core) handled by the consensus engine.
In Cosmos SDK, evidence is based around the evidence interface contract from **x/evidence**. Developers may define their own concrete evidence types that inherit this evidence interface. Submitted evidence is routed and then passed to a registered handler for the concrete evidence type created by the developers.

[11] Cosmos SDK Upgrade Module. 2022. https://github.com/cosmos/cosmos-sdk/blob/df4d6d1a4cd9fa0247f9db9378db857d95a1c1cb/x/upgrade/README.md

[12] Cosmos SDK Evidence Module. 2022. https://github.com/cosmos/cosmos-sdk/blob/df4d6d1a4cd9fa0247f9db9378db857d95a1c1cb/x/evidence/README.md

Evidence Proposal Submission

The most basic form of evidence must include logic to represent the following requirements:

1. A route to find the type of evidence
2. The type of evidence
3. A string representation of the evidence (e.g. a name)
4. A transaction hash corresponding to the evidence
5. Logic that validates the evidence at compile time
6. The chain height corresponding to the evidence

Evidence Registration

The evidence module must know all possible types of evidence, at run time, that it is expected to handle. Developers must define a router that registers routes to their evidence at compile time, and which allows the evidence module to use the evidence metadata to verify the information submitted. Once the evidence is registered, the router proceeds to find a corresponding handler for the evidence. This handler is responsible for executing the business logic for handling the evidence (e.g. validating).

Evidence Handling

During **BeginBlock** the Cosmos SDK transposes arbitrary evidence into a form of data known as an "equivocation". Whenever a valid equivocation is submitted in a block, that validator's stake is slashed an amount depending on the type of infraction committed. The penalties incurred are delegated to the **x/slashing** module.

Inter-Blockchain Communication Protocol

The Inter-Blockchain Communication protocol (IBC) enables cross-blockchain asset transfers between two sovereign blockchains that have both implemented the Cosmos SDK and ibc-go module.[13] The module implementation must meet certain requirements to interact with other IBC applications. They must bind to a port (or ports), include definitions and standardize an encoding format for packet data, use the default Acknowledgement struct provided by IBC (or optionally define a custom Acknowledgement struct), and they must implement the **ibc-go** interface. The following are important components of **ibc-go**: Clients, Client State, Client Height, Connections, Paths, Capabilities, Keepers, Packets, Channels, Timeouts, and Acknowledgement.

[13] Cosmos Inter-Blockchain Communication Protocol. 2022. https://github.com/cosmos/ibc-go

Clients

IBC clients track the consensus state of other blockchains, and have the tools to verify proofs against the client's consensus state. IBC clients can be associated with any number of target IBC blockchains.

Client State

An IBC client has a state that must contain chain-specific and light client-specific information necessary for verifying updates and upgrades to the IBC client.

Client Height

Client height is determined by two distinctions: revision number and revision height. Revision number is increased when hard fork events occur, and it also marks the revision of the blockchain (e.g. number of hard forks). Revision height marks the height of the chain within a given revision.
This system of revisions allows the IBC client to distinguish between different versions of the IBC target chain, and allows for comparison between the heights by using revision numbers to understand the nature of different nodes.

Connections

Connection Objects perform a connection handshake and are responsible for verifying the light clients of each counterparty. Once a connection is established it will handle all IBC cross state verifications, as the transacting blockchains do not talk to each other directly.

Paths

To communicate with another chain, a blockchain will commit state changes to a path reserved for the specific message and counterparty. Relayers read these paths, watch for updates, and submit data between the blockchains; e.g. blockchain A and blockchain B. This data, written on the path of blockchain A, is intercepted by the relayer and passed on to blockchain B in raw bytes, and the client of blockchain B is responsible for verifying the proof of this message.

Ports

IBC applications can connect to any number of ports that are uniquely identified with a specified ID (**portID**). When these ports are binded to, they return a capability object that restricts the execution nature of the port. It is the responsibility of the IBC Client to claim the capability that corresponds to it when the port is binded to.

Capabilities

The Cosmos SDK assumes the development of SDK modules could include faulty or malicious code. For this reason, the SDK utilizes an object capability model to ensure the reliability of modules by the granting of capabilities. A capability is described as a transferable right to perform operations on a given object. As a consequence of the object capability model, security properties can be established and maintained regardless of incoming unknown objects with faulty or malicious parts.

Capabilities allow for the provisioning, tracking, and authenticating of multi-owner privileges during runtime. Using capabilities, IBC can authenticate module actions so that modules remain within their encapsulated permissions.

Keeper

Keeper refers to a Cosmos SDK abstraction whose role is to manage access to the subset of state defined by various modules. Keepers provide developers with the ability to create module-specific memory stores that are scoped to their particular module. These scoped keepers can access a restricted subset of state defined by the module. Scoped keepers (also called sub-keepers), are bootstrapped at app initialization and cannot escape their module scope, ensuring that different modules cannot interfere with one another. If a module needs access to a subset of state defined in another module, a reference to the second module's keeper needs to be passed to the original module. Scoped keepers can create capabilities at initialization. Other modules must claim that capability in order to use it. Any module may check that a capability, with a particular name and that it has been associated with, does exist. Modules can fetch the capabilities they have already claimed, but remain unaware of any capabilities they have not claimed.

Scoped keepers have a persistent state and in-memory state. Their persistent memory maintains an incrementing index that maps a capability index to a set of capability owners in the form of tuples. The in-memory state tracks capabilities with two indexes: the forward index and the reverse index. The forward index maps the module name and capability tuples, to the capability name. The reverse index maps the module and capability name, to the capability itself.

Packets and Channels

Modules communicate with each other by sending packets over IBC channels which are established between two IBC ports. Similar to traditional TCP/IP, IBC Packets have identifiers for port (**portID**) and channel (**channelID**) which allow it to accurately route and know the source of packets.

There are two types of IBC channels: Ordered and Unordered. Ordered packets must be processed by the receiver in the order in which they were sent. Unordered packets can be processed in the order they are received.

There are four steps for opening a new IBC channel:

1. Blockchain A sends a **ChanOpenInit** message to signal a channel initialization attempt with Blockchain B.
2. Blockchain B sends a **ChanOpenTry** message to try opening the channel with Blockchain A.
3. Blockchain A sends a **ChanOpenAck** message to Blockchain B to mark its channel status as open.
4. Blockchain B sends a **ChanOpenConfirm** message to Blockchain A to mark its channel status as open.

There are two steps for closing an IBC channel:

1. **ChanCloseInit** initiates closing the channel from the initiating blockchain.
2. **ChanCloseConfirm** confirms closing the channel from the responding blockchain.

Timeouts

Considering the unpredictable nature of a distributed network, IBC must handle cases where packets either do not arrive in a timely fashion, or do not arrive at all. Packets must include a timeout height, or timestamp timeout, to constrain the execution of the packet. If a packet reaches its timeout, it can no longer be received by the destination chain. When this happens, a proof of packet timeout is submitted to the original chain which may perform custom logic as a response, such as rolling back the changes or retrying them.

Acknowledgement

Modules may also choose to execute custom logic upon processing a packet. This can be done both synchronously or asynchronously.

WebAssembly Smart Contracts

The **cosmwasm** module provides a runtime for WebAssembly smart contracts.[14] It includes methods for contract instantiation, execution, migration, querying, and state transitions. Execution may return a result or an error. In the case of an error, state transitions already performed will be reverted. **cosmwasm** is a part of CosmWasm, which is an ecosystem for Cosmos WASM smart contracts, consisting of a Virtual Machine, SDK, storage libraries, and the **cosmwasm** module.

WebAssembly (WASM)

WASM is a binary instruction format for a stack-based virtual machine. WASM is designed as a portable compilation target for diverse programming languages. Smart contracts can be written in any programming language that compiles to WASM, provided the binary output satisfies the **cosmwasm** interface.

Virtual Machine (VM)

A Virtual Machine is a computer system created using software in order to emulate the functionality of a physical computer. VMs run procedures with their own set of operating system instructions. The CosmWasm Virtual Machine represents an abstraction of the Wasmer engine, which is a WebAssembly runtime that enables lightweight containers to run in any environment. Wasmer engines are mainly responsible for two things: they transform the compilation code to create a binary, and they load those binaries so they can be executed by a caller (e.g., pushing code into executable memory). In CosmWasm, the Wasmer engine has been modified to execute code for smart contracts in a deterministic fashion.

Smart Contract

Smart contracts are represented on the blockchain by WASM bytes stored on-chain at a specific identifier (codeId). This custom set of bytes runs the functions of the WASM contract and has its own unique state. Once the codeId has been instantiated, it gets a contract address on the blockchain. Any codeId may be instantiated, at any moment, with a custom initial state. Smart contracts are dynamically uploaded, unlike Cosmos SDK modules, which are loaded during compile time. Smart contracts and modules can interact with each other.

[14] WebAssembly Smart Contracts for Cosmos SDK. 2022. https://github.com/CosmWasm/cosmwasm

There are two steps required for deploying a smart contract are:

1. Upload (e.g., **archwayd tx wasm store**)
2. Instantiate (e.g., **archwayd tx wasm instantiate**)

During Upload, WASM bytecode–which has been optimized by the CosmWasm optimizer–is stored on the blockchain without a state, and a **codeId** is assigned to the bytecode. During Instantiation, an instance of the bytecode stored at a given **codeId** is assigned its contract address and may be initialized with a state. Once a contract is Instantiated, its methods can be consumed. Consumable operations of a contract, or "entry points", are methods that can be called by accounts, contracts, and modules. They **include**: **Instantiate**, **Query**, **Execute**, **Sudo**, and **Migrate**.

Instantiate is an entry point and message for initializing the contract and its state.

Query is an entry point and message type for reading the contract's state.

Execute is an entry point and message type for performing state transitions.

Sudo is an entry point for granting privileged access to a contract, it can only be called by modules.

Migrate is an entry point and message type for moving state between different versions of a contract.

Smart contracts also possess three important properties, which are: Portability, Immutability, and Extensibility.

Portability refers to the fact contracts can be launched on any blockchain that implements **cosmwasm** (e.g. can be ported between chains).

Immutability means message formats cannot be changed (which would cause breaking changes and require a migration of their state).

Extensibility means contracts can implement custom messages and message types, and can extend default messaging and message types, without having to worry about underlying protocol changes (e.g. network upgrades).

Actor Model

cosmwasm uses the actor model for distributed systems. This model defines actors as primitives in concurrent computations. During contract operations the execution is an actor that has exclusive rights to the internal state of the smart contract. Actors cannot call each other directly, but they can trigger calls to one another upon completion of an execution (e.g. synchronously). By enforcing that internal state is restricted to the scope of an actor, CosmWasm VM serializes state transitions ensuring atomic execution and guaranteeing determinism.

Resource limits

A "fork bomb" is an attack in which some process is invoked to continuously replicate itself. To protect against denial of service attacks, wherein malicious actors upload fork bombs to consume resources of the node (e.g. causing the network to halt), the CosmWasm Virtual Machine constrains the resources of a node. VM constraints placed on nodes will limit the following resources:

1. Memory capacity
2. CPU usage
3. Disk usage

Memory capacity is limited to 32MB. This allows for most computations (barring a few notable examples, e.g. zero knowledge circuits).

CPU usage is constrained by the Wasmer runtime. The runtime calculates prices of operations before they are executed, which allows deterministic gas consumption regardless of the environment (e.g. transactions consuming too much gas can be rejected prior to their execution).

Disk usage is constrained by gas costs. All disk access is determined by the Cosmos SDK **KVStore** (e.g., key-value store, handled by scoped keepers). The **KVStore** enforces gas consumption which prices out malicious parties, as transactions require a gas payment proportional to the computational effort required by the execution.

Authorization and Grant Module

The **x/authz** module handles granting privileged access on behalf of a granter to a grantee.[15]

A grant allows the grantee to execute Cosmos SDK messages on behalf of the granter. Granters can always revoke privileges they have granted to any grantee.

Authorization

The Cosmos SDK includes some standard authorizations which can be granted to grantees, such as: Generic Authorization, Send Authorization, and Stake Authorization.

Generic Authorization gives the grantee unrestricted permissions to execute Cosmos SDK messages on behalf of the granter.

Send Authorization sets a spending limit on the grantee funds which can be spent by the grantee.

[15] Cosmos SDK Authorization and Grant Module. 2022. https://github.com/cosmos/cosmos-sdk/blob/main/x/authz/README.md

Stake Authorization allows the grantee to delegate, undelegate, or redelegate on the behalf of the granter, and sets a limit on the amount of funds the grantee can delegate, undelegate, or redelegate.

Cosmos SDK also allows developers to define custom authorizations. Custom defined authorizations must include the following:

1. They must process stateless validation of SDK messages.
2. They must determine whether a grant permits, or does not permit, executing SDK messages.
3. They must return a qualified SDK message service method that can accept or reject a request.

Grant

Grants can be read from the blockchain application state.

They are identified by combining the granter address bytes, grantee address bytes, and authorization type. This prevents granters from having multiple grants for a single authorization type at the same time.

Whenever a grant is created, it goes through a grant queue, which allows for the pruning of grants from the application state.

Fee Grant Module

The **x/feegrant** module allows accounts to grant fee allowances.[16] Fee allowances permit another account, the granter account, to pay for the fees of the grantee. Grantees will not need to maintain their own account balance for paying network fees (e.g. gas fees for submitting a transaction). Just like **x/authz**, **x/feegrant** uses a granter model. In this model, a granter permits an allowance to the grantee, but granters always retain the right to revoke any fee grants which they have granted.

Allowances

The Cosmos SDK defines three standard types of allowances. These types are: Basic Allowance, Periodic Allowance, and Allowed Message Allowance.

Basic Allowances permit the grantee to use a fee from the granter's account. This grant is constrained by a time limit and spending limit. If either of these constraints is violated, the allowance will be removed from state.

Periodic Allowances are a repeating type of fee allowance which operate continuously for a period of time, after which they are no longer enforceable and will be removed from state. This allowance type may also have a reset time and spending limit.

Allowed Message Allowance is an allowance type which limits spending of the granter's funds to a specific set of SDK messages defined by the granter (e.g. messages to a specific dapp). This allowance may be basic or periodic in nature, following its individual constraints.

Cosmos SDK also allows developers to define custom allowances. Custom defined allowances must include logic to perform:

1. Stateless validation of the SDK message
2. Retrieval of the grant expiration deadline
3. An Accept method containing any logic necessary to enforce the constraints of the custom allowance

[16] Cosmos SDK Fee Grant Module. 2022. https://github.com/cosmos/cosmos-sdk/blob/main/x/feegrant/README.md

Rewards Module Exclusive

The **x/rewards** module enables Cosmos SDK blockchains to calculate and distribute ARCH tokens to smart contracts that bring transaction volume to the network.[17] The module also introduces the concept of a minimum consensus fee, which sets the lower bound of a transaction fee.

Decentralized Application (dapp) Incentives

Decentralized Applications, or dapps, are applications deployed to the network as WASM smart contracts. Developers can choose for their dapps to receive tokens by sending a transaction with a custom message type (**MsgSetContractMetadata**), which will be processed by **x/rewards**. Tokens sent to dapps are proportional to gas consumed by transactions made to a smart contract on a per block basis.

The more gas a contract uses, the greater the amount of tokens will be that the contract will receive.

Tracking of gas consumption is handled by the **x/tracking** module.

Dapps receive a portion of the transaction fees they generate and a percentage of inflation.

Transaction fees

Dapps receive a portion of network transaction fees, which are calculated based on this formula:

$$ContractFeeRewards = (TxFees \times TxFeeRebateRatio) \times \frac{ContractTxGasUsed}{TxGasUsed}$$

Where:

1. **TxFees** are transaction fees paid by a user (e.g., an account)
2. **TxFeeRebateRatio** is an **x/rewards** module parameter that defines the ratio to split fees between the Fee Collector and the Rewards module accounts ([0..1))
3. **ContractTxGasUsed** is the total gas used by the contract within this transaction
4. **TxGasUsed** is the total gas used by all contracts within this transaction

[17] Archway Rewards Module. 2022. https://github.com/archway-network/archway/tree/main/x/rewards

Inflation Incentives

A portion of the per block inflation minted by the x/mint module is given to dapps based on their usage.

The formula for calculating this is:

$$\mathit{ContractInflationRewards} = (\mathit{MintedTokens} \ \times \ \mathit{InflationRewardsRatio}) \ \times \ \frac{\mathit{ContractTotalGasUsed}}{\mathit{BlockGasLimit}}$$

Where:

1. **MintedTokens** is the amount of new tokens minted per block by the **x/mint** module
2. **InflationRewardsRatio** is an **x/rewards** module parameter that defines the ratio to split inflation between the Fee Collector and the Rewards module accounts ([0..1))
3. **ContractTotalGasUsed** is the total gas used by the contract within this block
4. **BlockGasLimit** is the maximum gas limit per block (e.g., a consensus parameter)

Minimum Consensus Fee

Minimum Consensus Fee is a network parameter created to guard the network from malicious developers or block producers. A malicious actor could be someone creating artificial network activity (e.g. spam transactions) to receive larger developer incentives. The minimum consensus fee guards against this type of attack by enforcing a price for one gas unit such that any incentives received are unprofitable compared to the cost of sending the initial transaction to the contract.

The formula for calculating the minimum transaction fee of the network is:

$$\mathit{MinimumTxFee} = \mathit{MinConsensusFee} \ \times \ \mathit{TxGasLimit}$$

Where:

1. **TxGasLimit** refers to the minimum gas required by Cosmos SDK to execute a transaction

The minimum consensus fee is updated on a per block basis within the Rewards module. First, the inflation incentives are updated (e.g., based on dynamic inflation of the network), and then the minimum consensus fee is updated. The formula for calculating this is:

$$InflationBlockRewards = (MintedTokens \times RewardsRatio)$$

$$MinConsensusFee = \frac{InflationBlockRewards}{BlockGasLimit \times TxFeeRebateRatio - BlockGasLimit}$$

Where:

1. **TxFeeRebateRatio** is an **x/rewards** module parameter that defines the ratio to split fees between the Fee Collector and the Rewards module accounts ([0..1))
2. **InflationRewardsRatio** is an **x/rewards** module parameter that defines the ratio to split inflation incentives between the Fee Collector and the Rewards module accounts ([0..1))
3. **MintedTokens** is the amount of new tokens minted per block by the **x/mint** module
4. **BlockGasLimit** is the maximum gas limit per block (e.g., a consensus parameter)

Contract Premiums

Contract premiums allow smart contract developers to define a custom flat fee for interacting with their smart contract.

Contract premiums can be used to cover hidden costs of a smart contract, for example a NFT marketplace which delivers goods can use contract premiums to cover delivery costs.
The reasons for using contract premiums over using x/wasm funds are:

1. Fee predictability: Contract Premiums define a standardized way to define contract custom fees and can be used by wallets to predict fees
2. Rewards on Msg Fail: When using Contract Premiums rewards will be distributed even when the contract msg execution fails. Using the x/wasm funds way would not reward the developer if the msg execution failed due to bad input by the user.
3. Rewards withdrawal: Contract Premiums sends all the rewards to the configured rewards address. Using the x/wasm funds option would send all the funds to the smart contract unless custom transfer logic is implemented.

4. Easier regulatory compliance: Using contract premiums, developer receives the rewards only when they explicitly request to withdraw (similar to how staking rewards works). Using x/wasm funds to receive the funds, which happens immediately, might complicate the tax situation based on the jurisdiction.
5. One configuration to rule them all: Once set, Contract Premiums are applied to all Msg Executions exposed by the contract, as opposed having to be configured for every msg.

Tracking Module Exclusive

x/tracking is a custom Archway module.[18] It enables the tracking of gas consumption, on a per transaction basis, of CosmWasm **Execute** and **Migrate** operations in smart contracts.

Contract Operation Objects

Transactions could have multiple operations for one or more contracts (e.g. a contract calling the Execute entry point of another contract). In order to persist this information, a contract operation object is created. This object is pruned whenever dapp incentives are disbursed.

Transaction Info

Trackable transactions must have two unique identifiers: a height, and a gas value; which are represented in an abstraction called **Transaction Info**.

Block height refers to the height of the blockchain at which the execution was performed, the gas value is a total represented by the sum of gas consumed in all contract operations of the transaction. The formula for calculating this gas value is:

$$TotalGas = GasSDK + GasVM$$

Where:

1. **GasSDK** is the total gas used by the transaction outside of the WASM Virtual Machine
2. **GasVM** is the total gas used by the contract within the WASM Virtual Machine

[18] Archway Tracking Module. 2022. https://github.com/archway-network/archway/tree/main/x/tracking

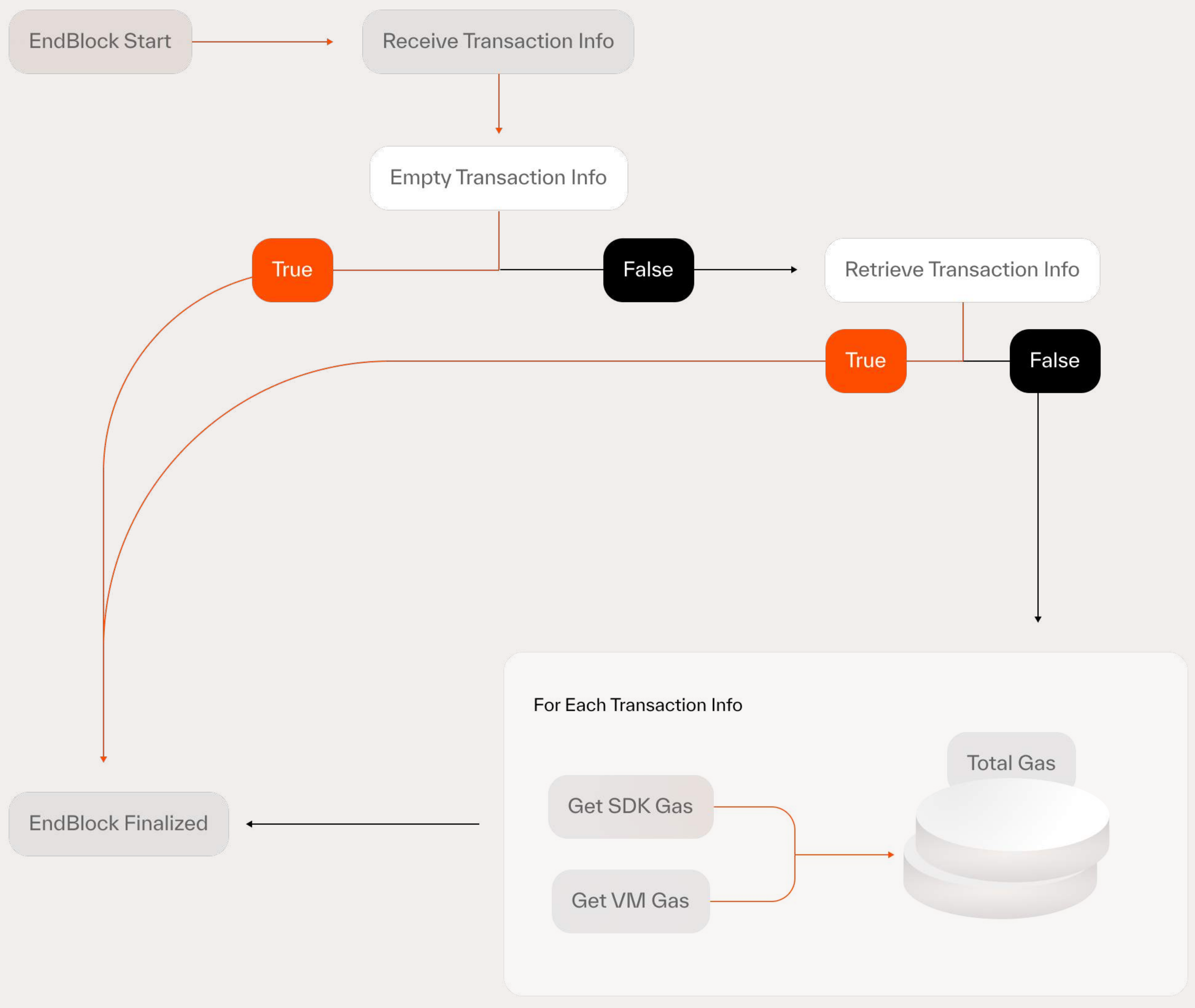


Figure 5

Tracking Engine and Gas Processor

Archway uses a modified version of the CosmWasm Wasmer Engine. These modifications enable a custom gas processor called the tracking engine. The tracking engine keeps a record of each contract's **Instantiate**, **Execute**, and **Migrate** operations. The gas processor intercepts smart contract operations and handles tracking the gas usage of contract operations.

Tracking the gas consumption of contract operations in a transaction involves several steps.

First, the transaction is received by an **AnteHandler** in **x/tracking**. Next, a **Transaction Info** is created. In the third step, the gas processor creates a contract operation object. Lastly, **EndBlock** finalizes gas tracking for the block. To finalize gas tracking, **EndBlock** retrieves the **Transaction Info** object created in the block, and then it retrieves the contract operation object linked to in the **Transaction Info** object. For each **Transaction Info** in the block, three distinct gas values are tabulated. The values tabulated are the total gas used by the transaction outside of the CosmWasm VM (e.g., **GasSDK**), the total gas used by the contract within the CosmWasm VM (e.g. **GasVM**), and the final gas tracking value (e.g. the sum of **GasSDK** + **GasVM**) which represents the total gas consumed by the smart contract in the transaction.

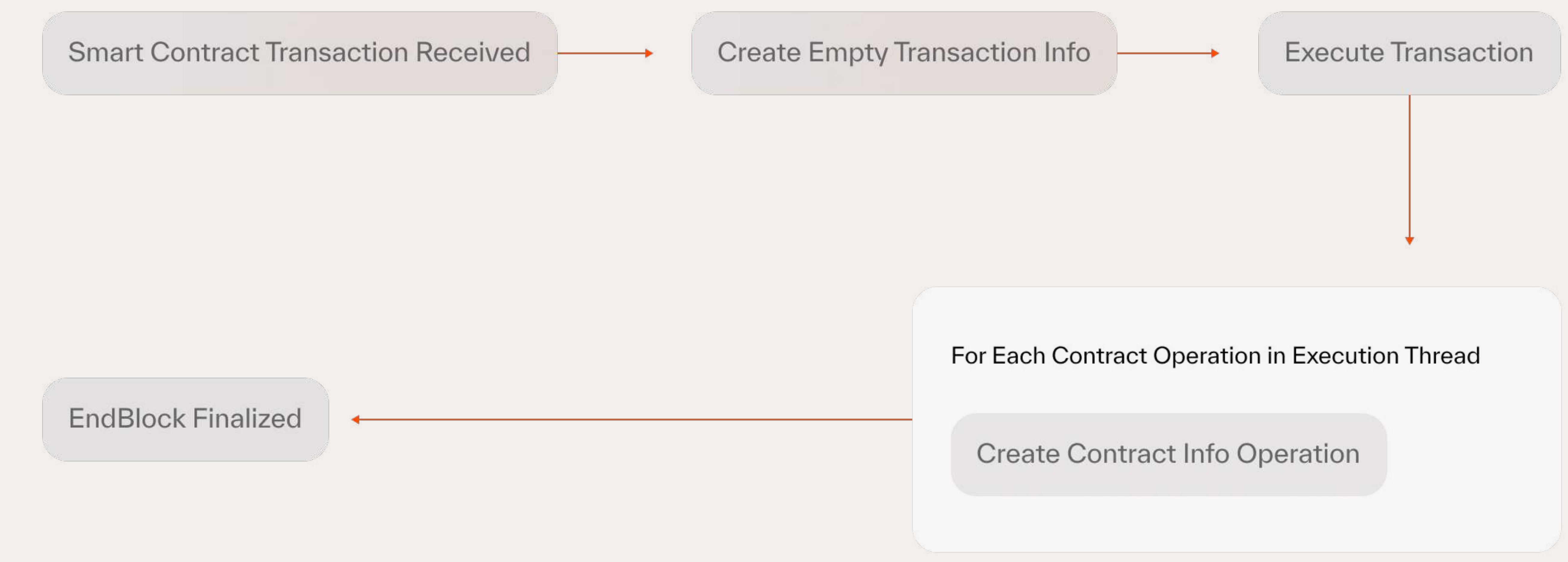

Smart Contract Transaction Received
Create Empty Transaction Info
Execute Transaction
For Each Contract Operation in Execution Thread
Create Contract Info Operation
EndBlock Finalized

Figure 6

CONCLUSION

Archway is expanding upon the framework of Cosmos SDK to enable consistent revenue streams for builders of dapps and web3 tooling. The additions of the x/rewards and x/tracking modules allow for the utilization of the network to directly benefit those who build its utility. Future additional module integration will be streamlined due to the efficient modular nature of the Archway blockchain.

As the economic innovation driving Archway's development blossoms, Archway will remain at the cutting edge of economic sustainability, technical advancement, and security. Not all applications need their own blockchain, and Archway's technical and economic infrastructure will simplify and readily scale smart-contract development and deployment.

By creating opportunities for independent developers to earn perpetual revenue from the utilization of their dapps and tooling, Archway aims to cultivate a brilliant developer community devoted to advancing the technology and user adoption of web3.

REFERENCES

Application Blockchain Interface. Tendermint. 2022. https://github.com/tendermint/tendermint/tree/main/abci

Archway Lightpaper: An Introduction to Archway. Archway.io. https://archway.io/assets/Archway-Lightpaper.pdf

Archway Rewards Module. Archway. 2022. https://github.com/archway-network/archway/tree/main/x/rewards

Archway Tracking Module. Archway. 2022. https://github.com/archway-network/archway/tree/main/x/tracking

Cosmos SDK Auth Module. Cosmos. 2022. https://github.com/cosmos/cosmos-sdk/blob/2418c3ef2e6f74fd6e7575b743fc1da4b53ab972/x/auth/README.md

Cosmos SDK Authorization and Grant Module. Cosmos. 2022. https://github.com/cosmos/cosmos-sdk/blob/main/x/authz/blob/2418c3ef2e6f74fd6e7575b743fc1da4b53ab972/x/auth/README.md

Cosmos SDK Authorization and Grant Module. Cosmos. 2022. https://github.com/cosmos/cosmos-sdk/blob/main/x/authz/README.md

Cosmos SDK Bank Module. Cosmos. 2022. https://github.com/cosmos/cosmos-sdk/blob/2418c3ef2e6f74fd6e7575b743fc1da4b53ab972/x/bank/README.md

Cosmos SDK Crisis Module. Cosmos. 2022. https://github.com/cosmos/cosmos-sdk/blob/df4d6d1a4cd9fa0247f9db9378db857d95a1c1cb/x/crisis/README.md

Cosmos SDK Evidence Module. Cosmos. 2022. https://github.com/cosmos/cosmos-sdk/blob/df4d6d1a4cd9fa0247f9db9378db857d95a1c1cb/x/evidence/README.md

Cosmos SDK Fee Grant Module. Cosmos. 2022. https://github.com/cosmos/cosmos-sdk/blob/main/x/feegrant/README.md

Cosmos SDK Governance Module. Cosmos. 2022. https://github.com/cosmos/cosmos-sdk/blob/df4d6d1a4cd9fa0247f9db9378db857d95a1c1cb/x/gov/README.md

Cosmos SDK Minting Mechanism. Cosmos. 2022. https://github.com/cosmos/cosmos-sdk/blob/df4d6d1a4cd9fa0247f9db9378db857d95a1c1cb/x/mint/README.md

Cosmos SDK Slashing Module. Cosmos. 2022. https://github.com/cosmos/cosmos-sdk/blob/df4d6d1a4cd9fa0247f9db9378db857d95a1c1cb/x/slashing/README.md

Cosmos SDK Staking Module. Cosmos. 2022. https://github.com/cosmos/cosmos-sdk/blob/2418c3ef2e6f74fd6e7575b743fc1da4b53ab972/x/staking/README.md

Cosmos SDK Upgrade Module. Cosmos. 2022. https://github.com/cosmos/cosmos-sdk/blob/df4d6d1a4cd9fa0247f9db9378db857d95a1c1cb/x/upgrade/README.md

High-level Overview. Cosmos SDK Documentation. https://docs.cosmos.network/main/intro/overview

Inter-Blockchain Communication Protocol. Cosmos. 2022. https://github.com/cosmos/ibc-go

WebAssembly Smart Contracts for the Cosmos SDK. Cosmos. 2022. https://github.com/CosmWasm/cosmwasm

Archway

medium.com/archwayhq

discord.com/invite/archwayhq

twitter.com/archwayhq

github.com/archway-network