Cosmos Dev Series: Cosmos Blockchain Upgrade

zeroFruit · Follow
Published in Web3 Surfers
14 min read · Jul 4, 2022

Photo by Zhanjiang Chen on Unsplash

In this article, we are going to talk about the Cosmos blockchain upgrade. On the Cosmos-SDK documentation, this process is called 'In-Place Store Migrations'

Although there are some documents related to the In-Place Store Migrations, I feel that those are quite high-level explanations, and the knowledge is scattered. As a blockchain engineer in the team, I want to understand the details of the upgrade process because error while upgrading the chain is critical to the network as well as to the community. So I want to make sure how Cosmos-SDK-based blockchain can be upgraded in detail and what components work together for the blockchain to be upgraded.

For understanding In-Place Store Migrations of Cosmos-SDK-based blockchain, we need to see Cosmovisor and some Cosmos-SDK modules.

This article contains the following contents:

- **'Upgrade Overview'.** We are going to see how the chain can be upgraded in an abstract view.
- '**Terms**': We are going to describe the words that show up frequently in the post.
- '**Upgrade From 'Cosmovisor' Point of View**': Lots of components from different layers work together for upgrading the blockchain successfully, in this section we are going to see the upgrade from the Cosmovisor point of view.
- '**Upgrade From 'Application' Point of View**': we are going to see the upgrade from the Application point of view.
- **'Component Details'.** We are going to deep-dive into each component related to the chain upgrade and how they are related to each other. Such as `x/upgrade`, `x/gov`, `ModuleManager`, `Configurator`.
- **'Things to do for the upgrade as a developer'.** Before going to the details of each component, we're going to list what we need to write code from the developer's perspective.

- '**Demo**'. We are going to see a simple demo of the Cosmos-SDK-based chain upgrade example with its sample script.

Terms

- **Cosmovisor**: Cosmovisor is a daemon that runs the application as its subprocess. And also a file watcher that polls and reads the file of `$DAEMON_HOME/data/upgrade-info.json`. `upgrade-info.json` file has the information of the upgrade name, and download URLs of the binary.

- **UpgradeHandler**: UpgradeHandler defines what modules need to be updated for the version. If the modules that need to be updated are not managed on the application such as `x/bank` on the Cosmos-SDK, that modules upgrade logic could be written in the UpgradeHandler. UpgradeHandler is written on the App level, not on the module level. UpgradeHandler must be registered to `x/upgrade` module.

- **MigrationHandler**: MigrationHandler defines how the module should be upgraded. MigrationHandler can contain the logic of how the parameters are updated, and how the state of the old scheme should be migrated into the new version. MigrationHandler must be registered to `Configurator` .

Upgrade From 'Cosmovisor' Point of View

- Run application (blockchain) by Cosmovisor.

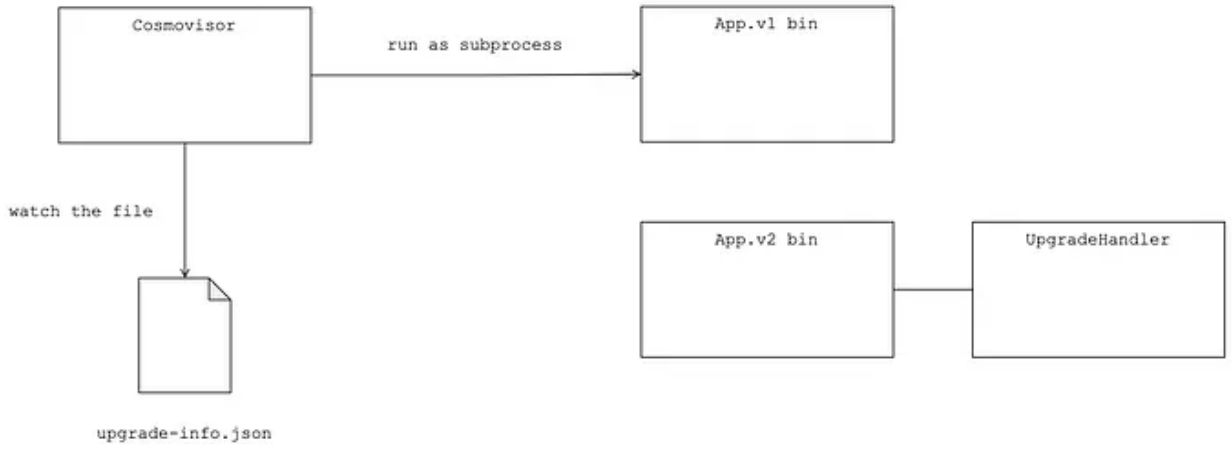
- If we need to upgrade the blockchain, build a new application binary that contains a MigrationHandler and UpgradeHandler. Then place that binary into the directory where Cosmosvisor can lookup.

- Submit SoftwareUpgrade proposal transaction to the `x/gov` module with the deposit. And make it pass with the vote. SoftwareUpgrade proposal contains the name of the upgrade, and the block height we want to upgrade. Each node that receives a proposal set `Plan` data structure inside `x/upgrade` module store

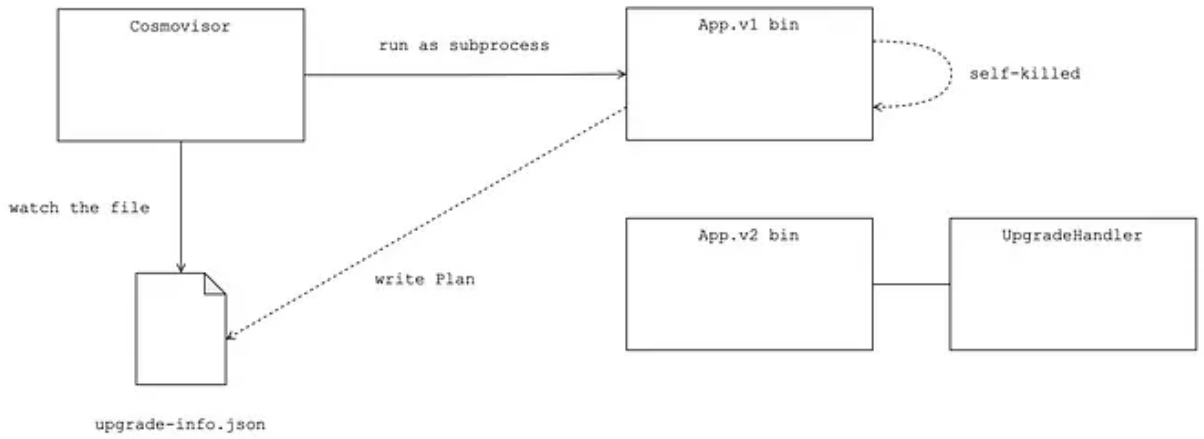
- Every block is created and received by the node, `x/upgrade` compares the block height with the height of `Plan` we stored before. If blockchain reaches that height it writes `Plan` data as a file into `$NODE_HOME/data/upgrade-info.json` and killed itself by `panic()` .

- Cosmovisor finds the application process is killed and based on the `upgrade-info.json` data it updates the symbolic link to lookup the proper directory which contains the updated binary. After that, Cosmovisor re-runs the application binary on the symbolic link. It automatically runs the new version application.
- As a new version of the application has the migration handler, it runs the migration handlers first. And if it was successful, the application runs normally as before.

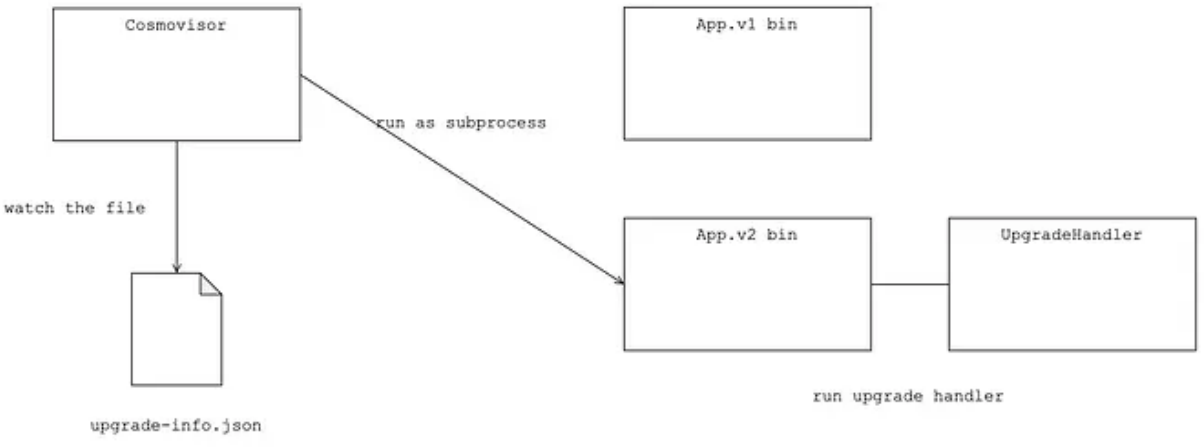
We can describe the abstract workflow of the upgrading procedure in the diagrams below:



Cosmovisor run App.v1 binary and watching upgrade-info.json



If blockchain reaches that height it writes Plan data as a file into `upgrade-info.json` and killed itself

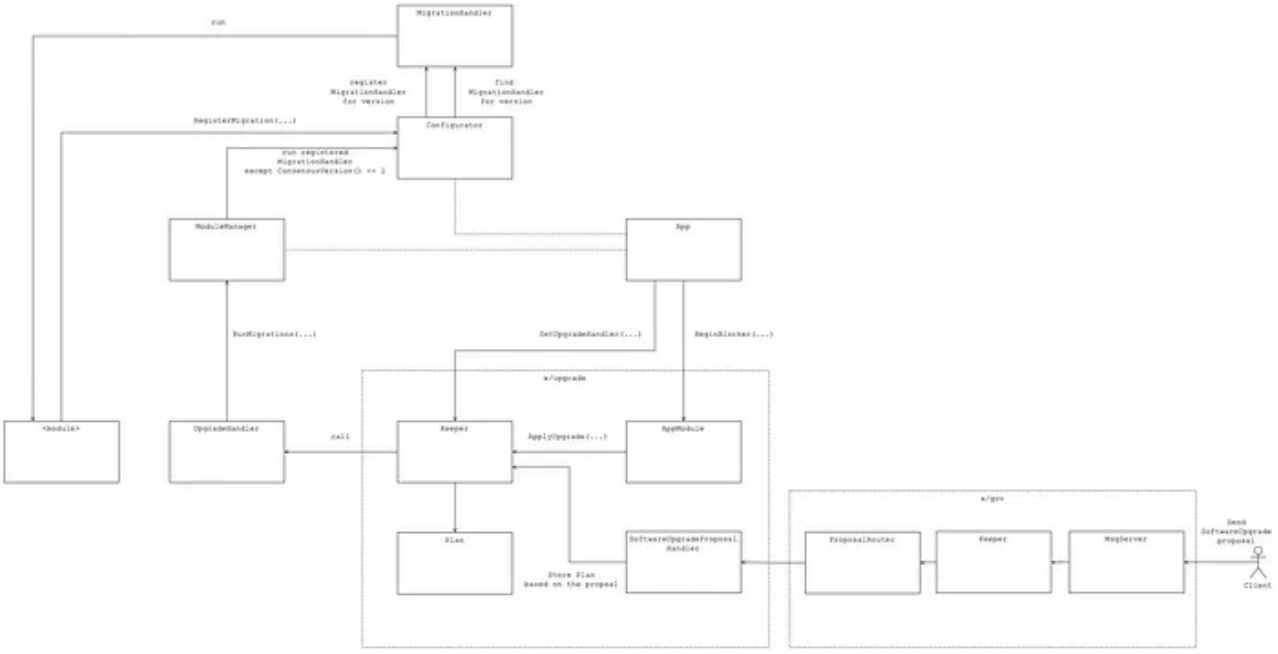


Cosmovisor re-run the application binary on the symbolic link

Upgrade From 'Application' Point of View

Let's take a look from the Application point of view. There're lots of components that work together to make In-Store Migration correctly.

- The client can send the SoftwareUpgrade proposal to the `x/gov` module MsgServer. After the proposal is passed, the content of the proposal is routed to the `x/upgrade` module SoftwareUpgradeHandler and is saved as `Plan` data in the `x/upgrade` module state.
- Every time the block is created, `x/upgrade` 's BeginBlocker checks whether it's time to upgrade by block height. If it's time, the old version application is panic and the new version of the application is run by Cosmovisor.
- After the new version of the application is run, `x/upgrade` module runs the registered UpgradeHandler and UpgradeHandler runs the ModuleManager `RunMigrations(…)` method which calls each module's MigrationHandler if needed. So for the module to be updated successfully, MigrationHandler must be registered on the new version of the application.



Component relation inside Application

Component Details

Now, let's take a look at each component in detail.

Cosmovisor

For Cosmovisor to work correctly, we or the node operator need to setup the folder structure. The whole folder structure would be as follows:

`cosmovisor` is the directory that Cosmovisor actually uses for upgrading the application. And the others like `config`, `data` is for the node as we know.

Let's look at the `cosmovisor` directory. `current` is a symbolic link Cosmovisor generates so we don't need to create it manually. `genesis` the directory contains the first version of the application binary. Cosmovisor looks up `$DAEMON_HOME/data/upgrade-info.json` when it starts and if the file does not exist, create `current` with the reference of `genesis`. In other words, when Cosmovisor runs up the first time, it runs the binary and points to the `genesis` directory.

`upgrades` directory contains various versions of binaries. You can find the `v2.0.0` directory under the `upgrades`. **`v2.0.0` must match the upgrade name when we submit the SoftwareUpgrade proposal to the `x/gov` module.**

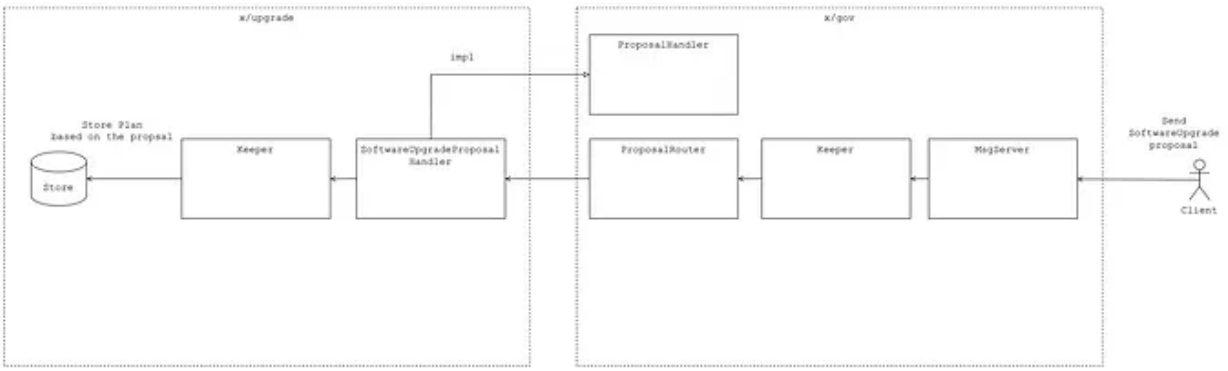
As we've seen in the 'Upgrade Overview' section, before the old version of the application killed itself, it writes `upgrade-info.json` with the information of the new upgrade name. And that name is determined by the contents of the SoftwareUpgrade proposal we submitted before.

And Cosmovisor reads the `upgrade-info.json` and finds which version of the application should be run. Cosmovisor will update the `current` symbolic link with the directory name of the upgrade name (e.g. `v2.0.0`). As always, Cosmovisor finds the binary to run in the `current` symbolic link, but at this time, it points to the new version of the application.

You can find the official docs here.

x/gov

What `x/gov` mainly done for the upgrade is to receive the SoftwareUpgrade proposal and after the proposal is passed, send proposal content to the `x/upgrade` module to make it do subsequent jobs for the upgrade.



Submit SoftwareUpgrade proposal

The client can send the software upgrade proposal to the `x/gov` module message server. Proposal structures look as follows:

https://github.com/cosmos/cosmos-sdk/blob/main/proto/cosmos/upgrade/v1beta1/upgrade.proto#L12-L56

The proposal contains the title and description of the upgrade and the '`Plan`' structure. The `Plan` contains important data related to the upgrade. `name` must match with the directory name that Cosmovisor watches under the

`$DAEMON_HOME/cosmovisor/upgrades/`. For example, if the name of the upgrade is `v2.0.0`, the application binary `v2.0.0` must be in the `$DAEMON_HOME/cosmovisor/upgrades/v2.0.0/bin/`.

As `x/gov` module receives the proposal through a message server, it waits for the proposal to be passed through voting.

https://github.com/cosmos/cosmos-sdk/blob/main/x/gov/abci.go#L15-L130

If it passes, the proposal router routes the content of the SoftwareUpgrade proposal to the `x/upgrade` module. The proposal router knows where this proposal should be routed. We usually setup a proposal router in `app/app.go` before injecting `x/gov` module into the application.

Now the proposal is passed to the `x/upgrade` module. `x/upgrade` module gets the `Plan` data and saves it in its state.

x/upgrade

`x/gov` module helps `x/upgrade` to know the `Plan` of software upgrade: when (block height) and what (the name of version). **But still don't know how to upgrade for each module**. An application consists of modules. Each module functionality makes up the whole functionalities of the application. Upgrading an application (i.e. chain) usually means upgrading some modules. There might be some bugs, improvements, or adjustments to parameters.

So we need to define UpgradeHandler for the version we planned to upgrade. By registering the UpgradeHandler, `x/upgrade` modules know how to upgrade for that version. UpgradeHandler signature looks as follows:

https://github.com/cosmos/cosmos-sdk/blob/main/x/upgrade/types/handler.go#L26

https://github.com/cosmos/cosmos-sdk/blob/55054282d2df794d9a5fe2599ea25473379ebc3d/types/module/module.go#L369

We've seen `Plan` structure already. VersionMap is a map with the key of moduleName and value with the ConsensusVersion. **ConsensusVersion** is the module scoped version. So each module has a different ConsensusVersion. If the

application needs to update `x/foo` module only, we need to increase the ConsensusVersion of `x/foo` module.

`x/upgrade` manage the state of VersionMap in addition to `Plan`. Every time the upgrade happens, `x/upgrade` module injects the current VersionMap into the UpgradeHandler.

Then how UpgradeHandler for a specific version is invoked?

For the old version of the application, it just panic and self-killed its process. Every time the block is created `x/upgrade` module checks the `Plan` whether it's time to run the upgrade procedure. It is normal that the old version application to panic and be killed when reaches the target block height because it has not UpgradeHandler. The old version application just writes the Plan data into the `$DAEMON_HOME/data/upgrade-info.json` file and exits the process when reaches the block height. So don't panic about the panic message of the application.

As we talked about before, Comovisor watches the `$DAEMON_HOME/data/upgrade-info.json` file and updates the symbolic link to reference the new version of binary. And run the new version of the application. **So UpgradeHandler should be defined in the new version of the application code, not in the old version.**

Before the new version application is completely bootstrapped. it runs its UpgradeHandler. `x/upgrade` module finds the `upgradeHandlers` by the upgrade name (as I said before upgrade name is important) and then runs it. After that update the VersionMap with the return value of UpgradeHandler.

https://github.com/cosmos/cosmos-sdk/blob/55054282d2/x/upgrade/abci.go#L23-L95

UpgradeHandler

Now, let's look at how UpgradeHandler can be implemented. There are various ways to structure the UpgradeHandler but I think Evmos way is neat.

Before the code, we should think about the upgrade scenarios: when we need the upgrade.

1. **We need to upgrade the module we depend on**, the module we depend on means the module we depend on such as `x/bank` on Cosmos-SDK or `x/evm` on

Ethermint.

2. **We need to upgrade the custom module**, custom module means the module we developed for the application.
3. **We need to add a new module.**

The ways to achieve each goal are slightly different.

Let's look at the package structure first:

Under the `app` directory there's `upgrades/` directory and under the `upgrades/` there are upgrading codes for the specific version. It is ok not to match the name of the directory for the specific version with the upgrade name we proposed before.

`app/upgrades.go` file contains the aggregate of `upgrades/` package codes and `app/app.go` uses the `app/upgrades.go` code.

The abstract format of UpgradeHandler and setup codes are as follows:

`app/upgrades/v2/constants.go` contains the value of such: upgrade name, Testnet/Mainnet upgrade block height. For the upgrade name, it must match the name we submitted SoftwareUpgrade proposal before.

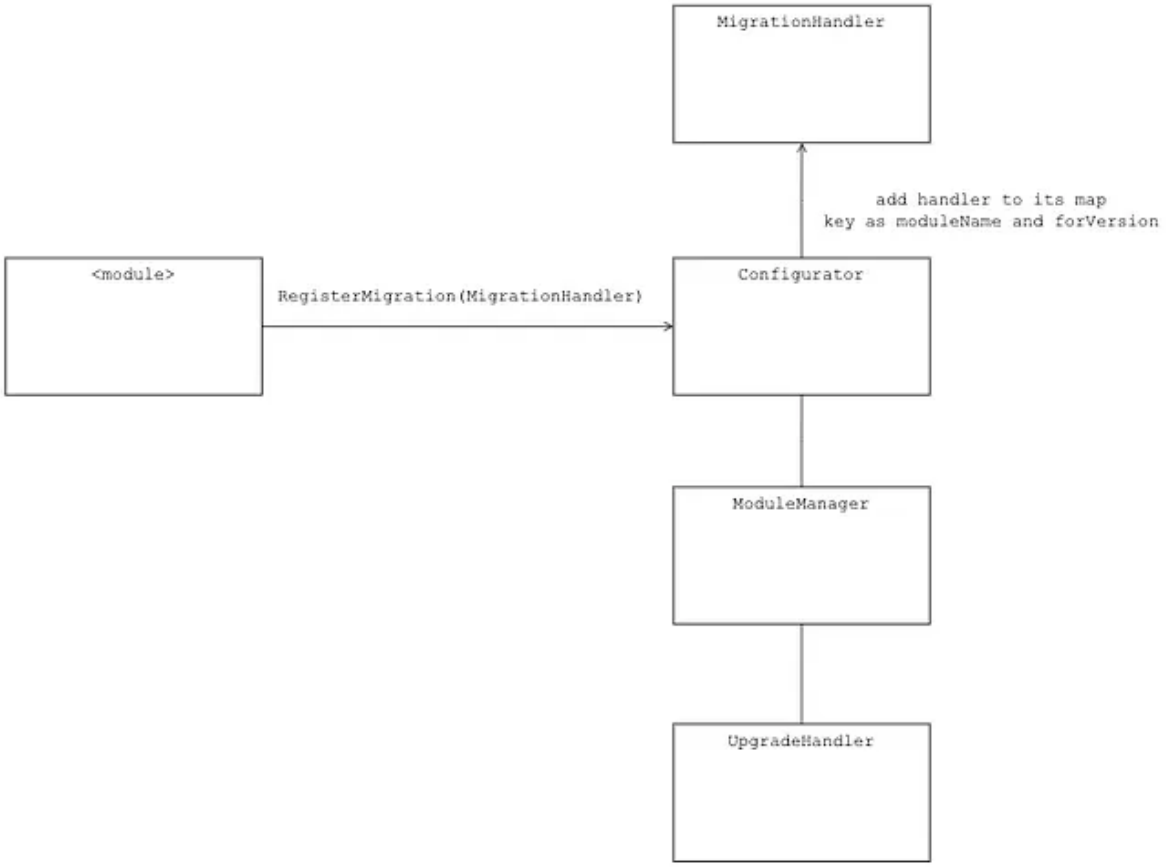
`app/upgrades/v2/upgrades.go` defines the UpgradeHandler which runs when Cosmovisor executes the new version of the application binary. **And UpgradeHandler must end with** `return mm.RunMigrations(ctx, configurator, vm)` . Inside ModuleManager, it triggers each module's MigrationHandler. We'll see the process later.

On the `app/upgrades.go` , we need to register UpgradeHandler to `x/upgrade` module. `x/upgrade` module tries to find the UpgradeHandler for the 'upgrade name' we saw it several times and if it exists, run it in the BeginBlock steps.

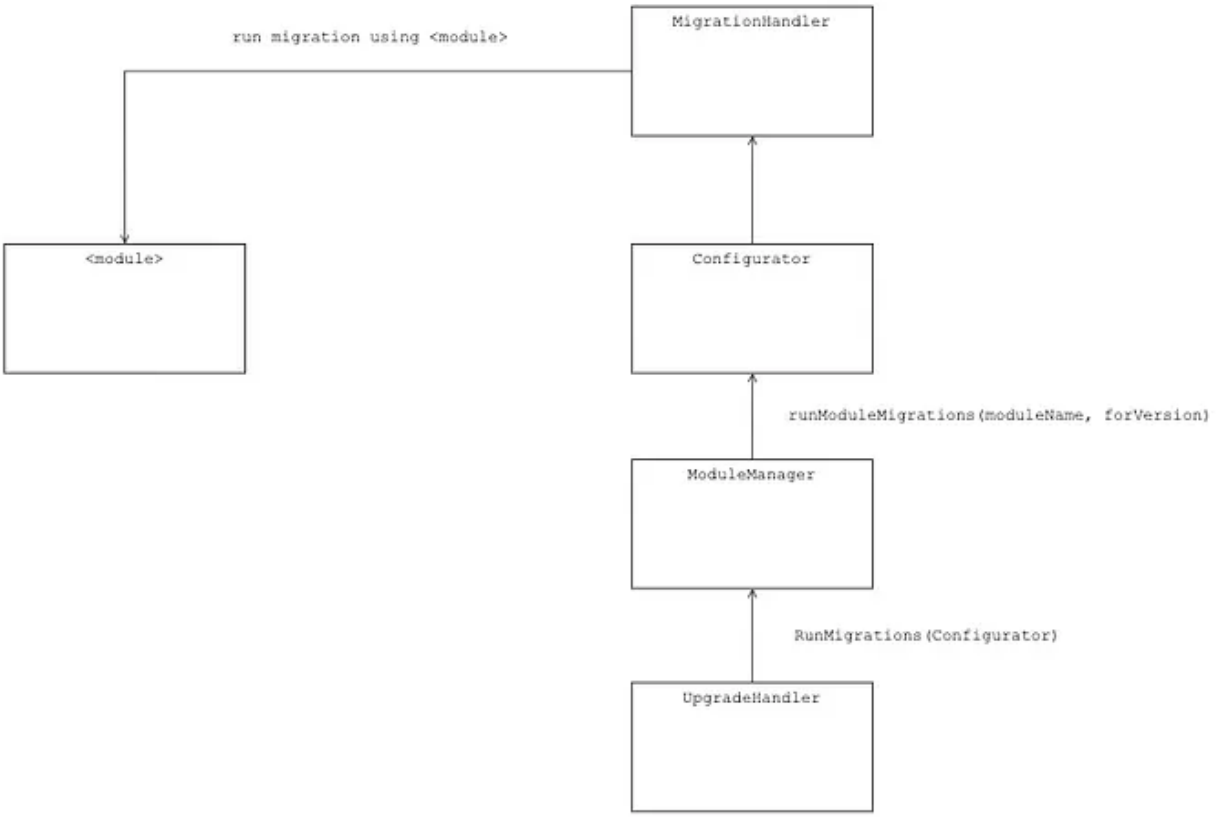
And if we want to add new modules, instead of updating existing modules, we need to call `app.UpgradeStoreLoader` . This also will be explained below.

reference code: https://github.com/evmos/evmos/blob/main/app/app.go#L774-L780

On the `app/app.go` file, we need to call the method `setupUpgradeHandlers` we defined previously in the constructor of the application. This method must be called before `app.LoadLatestVersion()` , because that call sealed the options and parameters of the app and we cannot modify those anymore.



Register MigrationHandler of the module through Configurator



When UpgradeHandler runs, it calls ModuleManager to find its MigrationHandler. And MigrationHandler migrates the state using the module.

Scenario#1: Upgrading Existing Module

Let's think about the first scenario: upgrading existing modules.

You may have noticed that we cannot update the codes of existing modules. For example, `x/bank` in Comos-SDK we cannot update its codes directly. So the extent we can update is parameters and states (not schema). For example, we can update the `sendEnabled` parameters for specific denom or add the denom metadata using `x/bank` `SetDenomMetaData()` interface. we cannot update the internal state schema of denom metadata.

For updating the existing module, we need to inject related keeper into the `CreateUpgradeHandler()` . Let's say we want to add `x/bank` denom metadata. We need `x/bank` keeper to update its state. So the code would be as follows:

reference code:
https://github.com/evmos/evmos/blob/main/app/upgrades/v5/upgrades.go#L31-L99

Scenario#2: Upgrade Custom Module

Second scenario: upgrading custom modules.

In this case, we can update the code easily. In other words, in addition to updating parameters and data, we can update the state schema. But for upgrading the schema, we need to do additional work for the modules that migrate the schema.

- Backup old version state schema into the other directory. Most projects save the old version schema into `x/<module>/migrations/<version>/` directory. Although the application is a new version, we still need the old version of the state schema. Because we need to migrate the old version states into the new version schema.
- Write migration handler for the specific version.

Below is a workflow for upgrading the custom modules.

Before generating code with the new version of Protobuf copy the current Protobuf generated files into the `x/<module>/migrations/<version>/` directory. For example, if `x/foo` module we developed needs to update its state schema, first place current state schema code files like `params.pb.go` into the `x/foo/migrations/v1/types/params.pb.go`. Then generate codes with the new Protobuf files which will be placed in `x/foo/types/` directory.

If we need to migrate the old state into a new state with a different schema. We need to write the migration handler and register it to the configurator. These migration handler codes usually put into `x/<module>/keepers/migrations.go` file and concrete logic of migration is placed under the `x/<module>/migrations/<new-version>/migrations.go` .

For example, if we are going to migrate `x/foo` module from version 1 to 2, place Migrator into `x/foo/keeper/migrations.go` and write concrete MigrationHandler codes into the `x/foo/migrations/v2/migrations.go` as below:

In this example code, `x/foo` module updates its parameter and state schema. Migrator uses Keeper to get the state of x/foo. Concrete update logic is defined in `x/foo/migrations/v2/migrations.go`

`x/foo/migrations/v2` package uses both `x/foo/types` and `x/foo/migrations/v1` packages for migrating state into a new version. For updating the parameter, it simply gets the reference of Subspace of `x/foo` module and set the value of the parameter key.

In the case of `MigrateStore()`, for example, we can implement migration logic as below: first retrieve the key, value of state from the store and save it with the new schema and delete the old state. This is one of the ways to implement it.

Finally, register the migration handler on the Configurator using `RegisterMigration(moduleName string, forVersion uint64, handler MigrationHandler) error` method. Once the module registers the migration handler, ModuleManager runs the MigrationHandler using Configurator inside of UpgradeHandler.

You can find that MigrationHandler is registered via `cfg.RegisterMigration()` with the `forVersion` of 1. And `ConsensusVersion()` method of `x/foo` module returns 2.

VersionMap version of this module works as `fromVersion` and return value of `ConsensusVersion()` works as `toVersion` when running loop of upgrading module inside `ModuleManager`. Range for `fromVersion` as inclusive, `toVersion` as exclusive. And inside each loop, `ModuleManger` tries to find the MigrationHandler for the `forVersion`. For example, if the current version of foo in the VersionMap is 1 and the return value of `ConsensusVersion()` of foo is 3. `ModuleManager` tries to find MigrationHandlers for versions 1 and 2.

In the code above, because the current version in the VersionMap is 1 and the return value of `ConsensusVersion()` is 2. We only need the MigrationHandler for the `forVersion` of 1.

https://github.com/cosmos/cosmos-sdk/blob/55054282d2/types/module/module.go#L371-L462

Scenario#3: Adding New Module

Third scenario: adding a new module to our application.

In this case, we don't need to write the migration handler as Scenario#2. Because we still don't have states for the new module. Instead, we need to update the store for our new module. This goal is achieved by calling `UpgradeStoreLoader` with the `StoreUpgrades` .

reference code: https://github.com/evmos/evmos/blob/main/app/app.go#L1029-L1104

In the code above, we've specified `newmoduletypes.StoreKey` the `Added` field inside `StoreUpgrades` struct. And that values are passed into `UpgradeStoreLoader` . By doing this, we can allocate a store for the `x/newmodule` in the multi-store of application. And make `x/newmodule` is correctly added to the app. 'Correctly added' means the basic things we need to do to add a module to the application such as adding Subspace of the module, adding a module to the ModuleManager, etc.

Things to do for the upgrade as a developer

Now from the application developer perspective, what codes do we need to write for upgrading the chain successfully? Let's list TODOs from the high level.

TODO#1: Write UpgradeHandler

First, we need to write UpgradeHandler and register it to `x/upgrade` module. If we want to update modules we do not manage such as `x/bank` that Cosmos-SDK provides, we should inject `x/bank` module into the UpgradeHandler. And we need to call ModuleManager's RunMigration method.

TODO#2: Write MigrationHandler

If we want to upgrade the existing module that we manage, we need to place the old version of types into the other directory, update proto files for the new version then write the migration logic from the old states into the new scheme of state. After writing MigrationHandler, we need to register it via Configurator

TODO#3: Update `ConsensusVersion()` of the module that needs to be updated

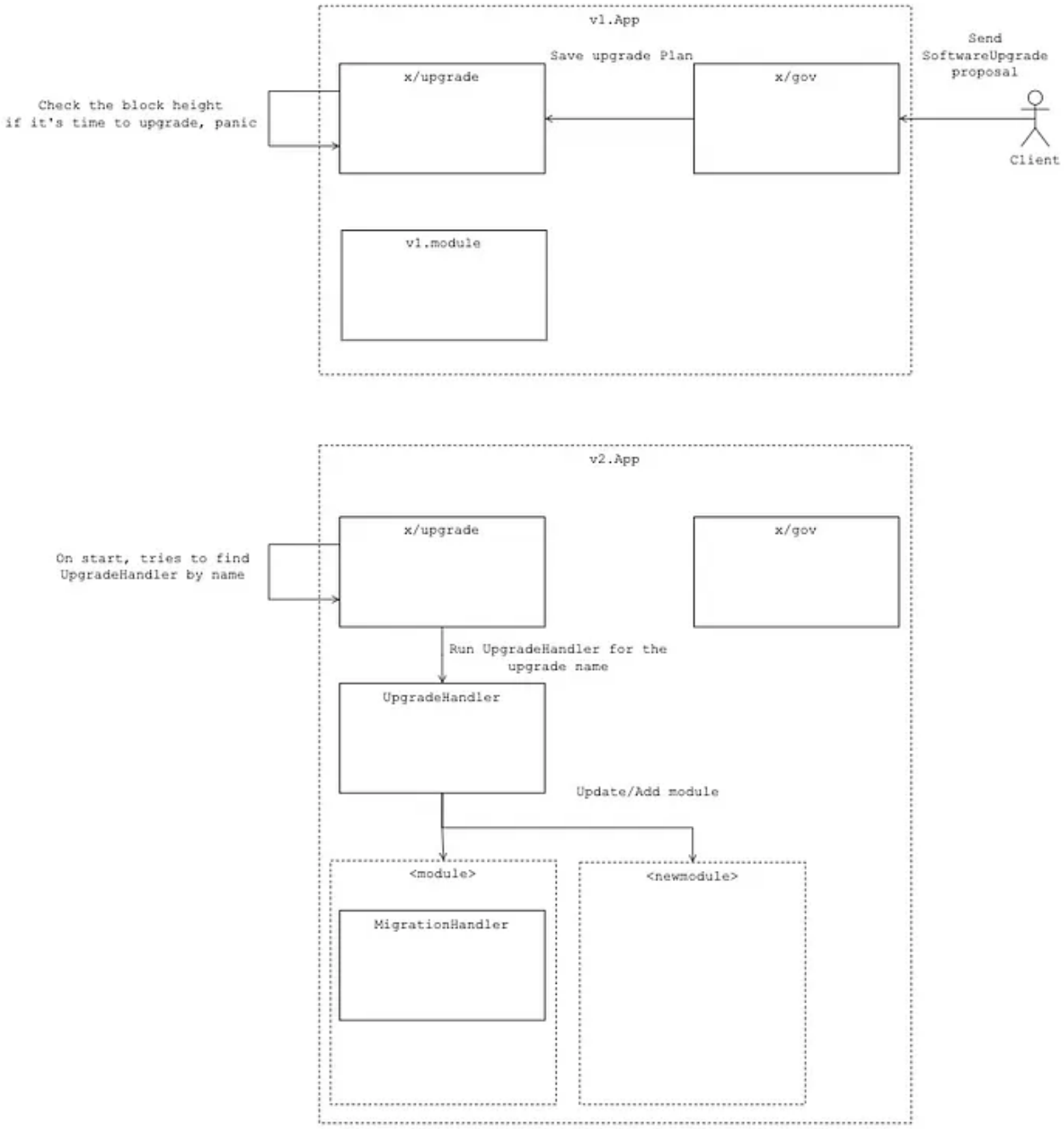
ConsensusVersion simply puts the module scoped version. If we update the ConsensusVersion for the module, ModuleManager detects it and runs the

MigrationHandler for the version. And that will be the handler we wrote in TODO#2.

Those TODOs might be slightly different by the upgrade scenarios:

1. **We need to upgrade the module we depend on**, the module we depend on means the module we depend on such as `x/bank` on Cosmos-SDK or `x/evm` on Ethermint.
2. **We need to upgrade the custom module**, custom module means the module we developed for the application.
3. **We need to add a new module.**

Keep in mind that UpgradeHandler and MigrationHandler should be written in the new version of the application code, not in the old version.



Cosmos Blockchain Upgrade Demo

This is a simple blockchain upgrade demo. The script used in the demo is explained below. You can find the module state schema is changed after the upgrade. The code of migration is similar to the code in the 'Component Details' section.

Sample Scripts for Testing Upgrade on the Local

Before running the test, we need to download ignite, Cosmovisor binary.

setup-node.sh

From the terminal, if we run the `setup-node.sh` , we build the application binary of versions v1 and v2. And copy those into Cosmovisor temporary directory. After initializing the data for the node with *ignite*, copy Cosmovisor temporary directory into the `$DAEMON_HOME` . Finally, run the application via `cosmovisor` command.

upgrade-proposal.sh

After running `setup-node.sh` script successfully, open another terminal window, then run the `upgrade-proposal.sh` to send the SoftwareUpgrade proposal to upgrade to `v2.0.0` on the 40th block.

After blockchain has reached to 40th block, the v1 application panics and Cosmovisor re-run the application with the version of v2.

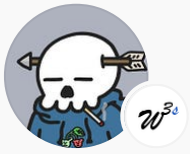
What's Left/Next?

We've seen several scenarios to update/add modules on the application. But there are still left some edge cases. For example, we should handle the case when the UpgradeHandler not working correctly, so as MigrationHandler, how should we roll back temporarily, etc.

In this article, we've mainly seen the chain upgrade process from the technical part. But we need to address more about governance such as how the network going to propose the version upgrade to the community and vote.

For those who are curious about chain upgrades in the technical view, hope this post helps you understand more in detail.

Blockchain | Cosmos | Blockchain Development | Software Development | Cosmos Network

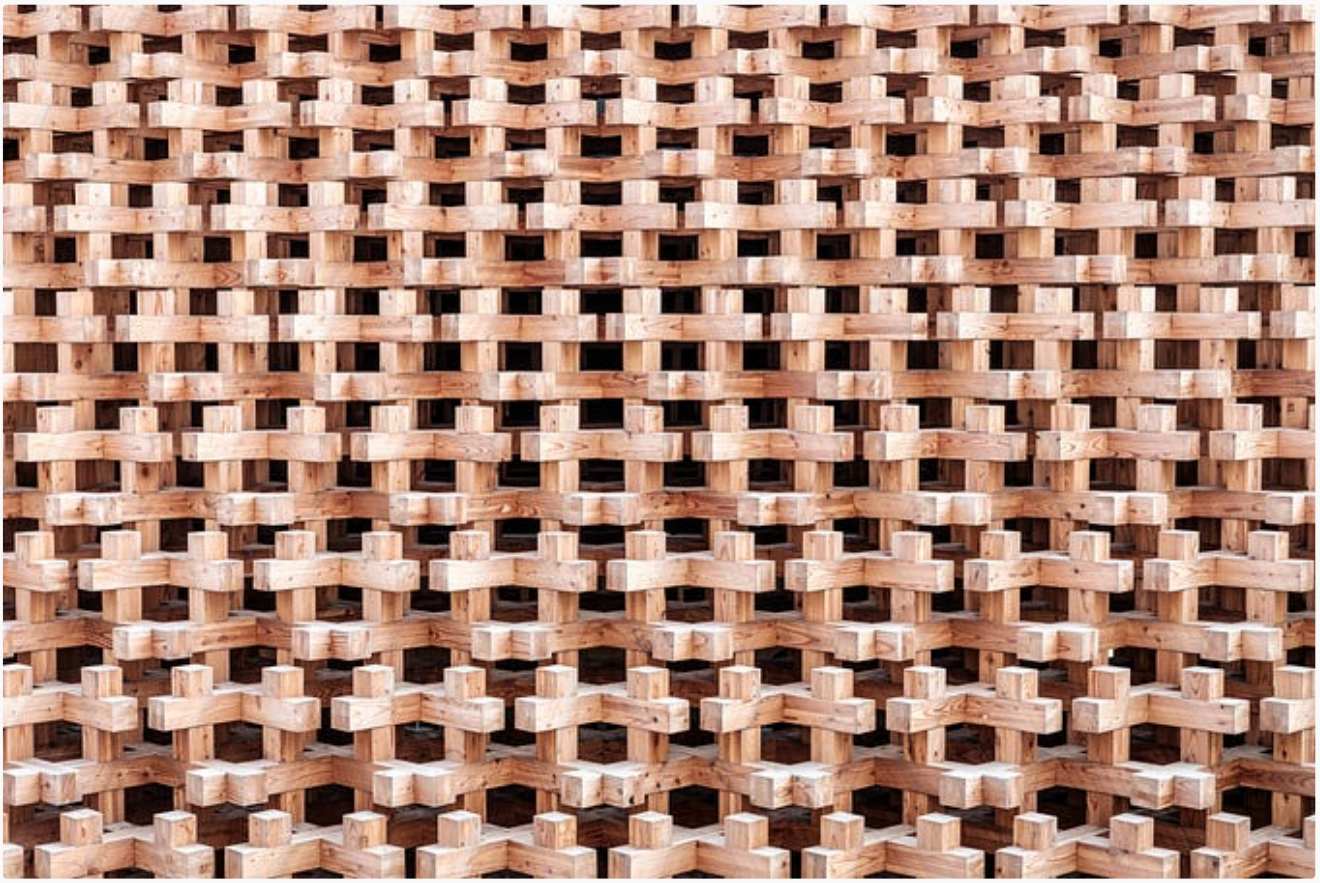


Written by zeroFruit

178 Followers · Editor for Web3 Surfers

Software/Blockchain Engineer, https://github.com/zeroFruit

More from zeroFruit and Web3 Surfers



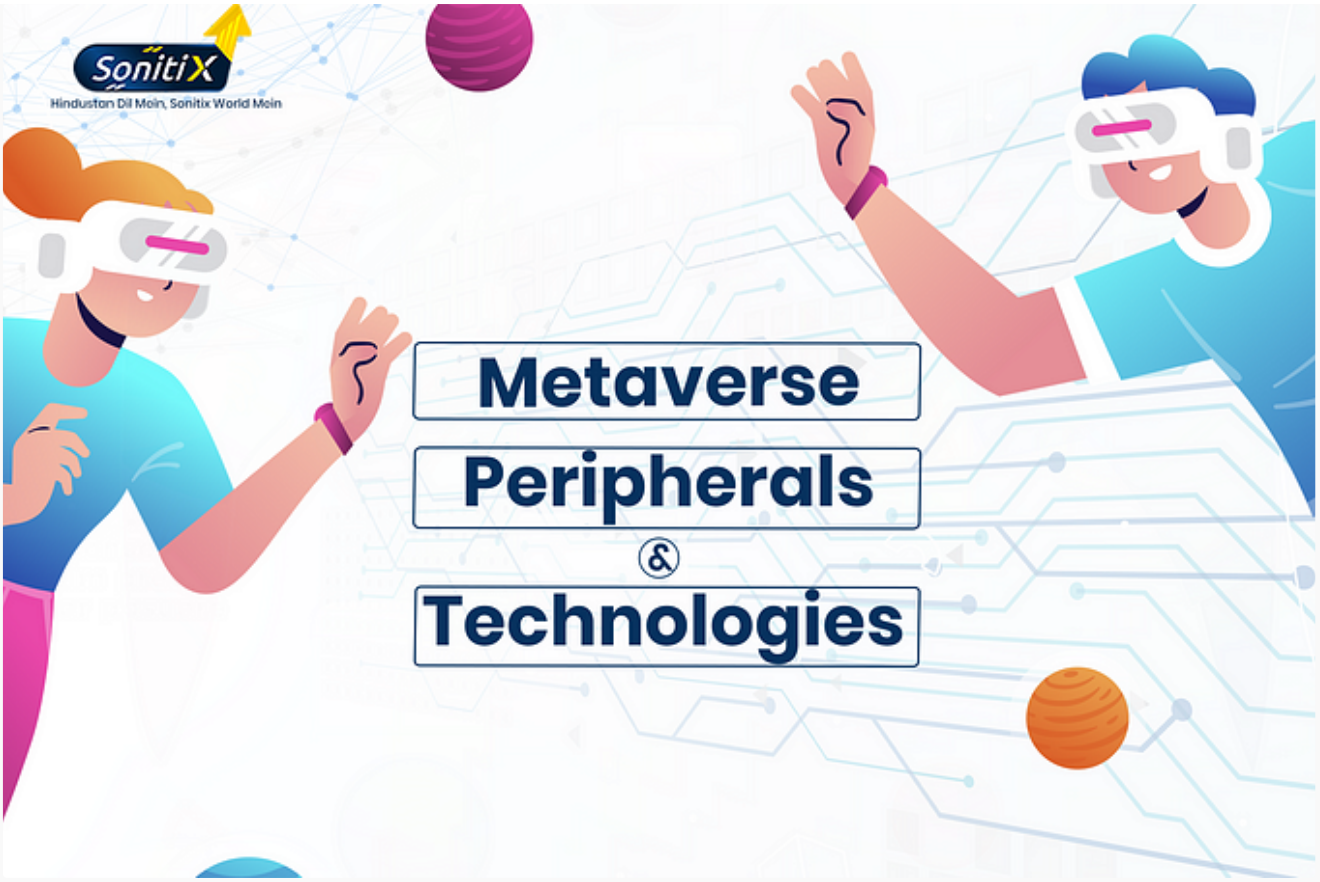
zeroFruit

What is AES?—Step by Step

In this post, we are going to find out what is AES, how its algorithm works. And in the last section using python AES modules we are going...

7 min read · Feb 13, 2019

88 2



Sonitix Exchange in Web3 Surfers

Metaverse Peripherals & Technologies

Chris Ahn in Web3 Surfers

Dune for dummies

Learn basic data analytics for on-chain data in 5 minutes !

4 min read · Jul 10, 2022

97 1

zeroFruit in Coinmonks

DelegateCall: Calling Another Contract Function in Solidity

In this post, we're going to see how we can call another contract function. And we talk about

8 min read · Sep 1, 2019

890 13

See all from zeroFruit

See all from Web3 Surfers

Recommended from Medium

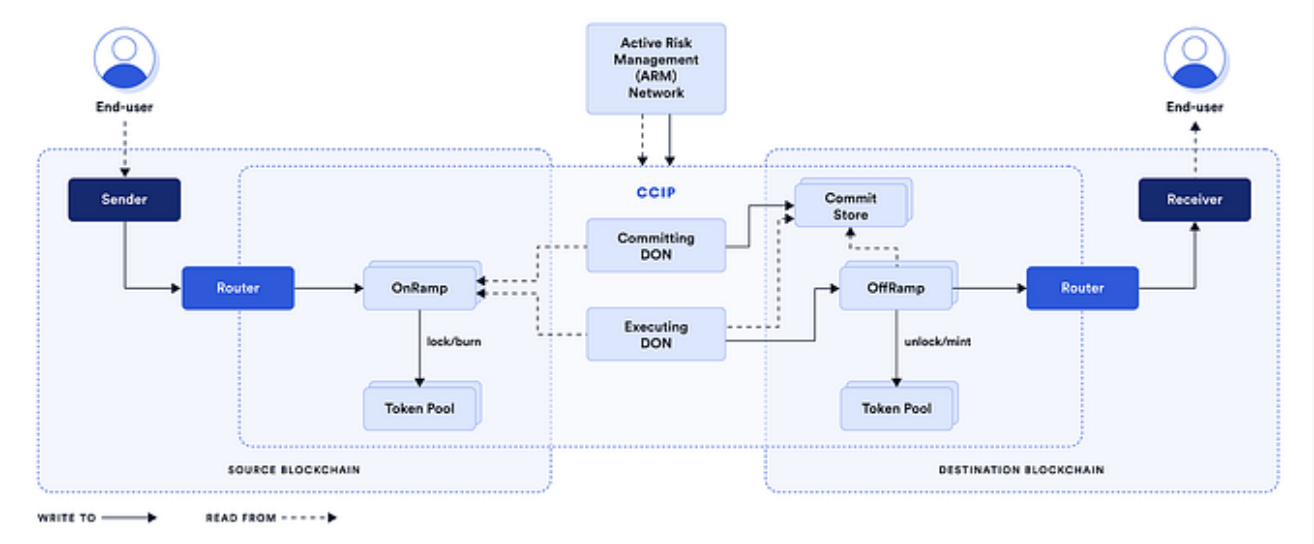
Adam Boudjemaa in Solichain Web3 Blog

Solidity 1.0.0 - Next-Gen Smart Contracts

Solidity Unleashed: The Exciting Evolution to Version 1.0.0

6 min read · Nov 24

261 4



Av. Elif Hilal Umucu in Chainlink Community

CCIP (Cross-Chain Interoperability Protocol) Guide🏗️🛠️

24 min read · Jul 29

Lists

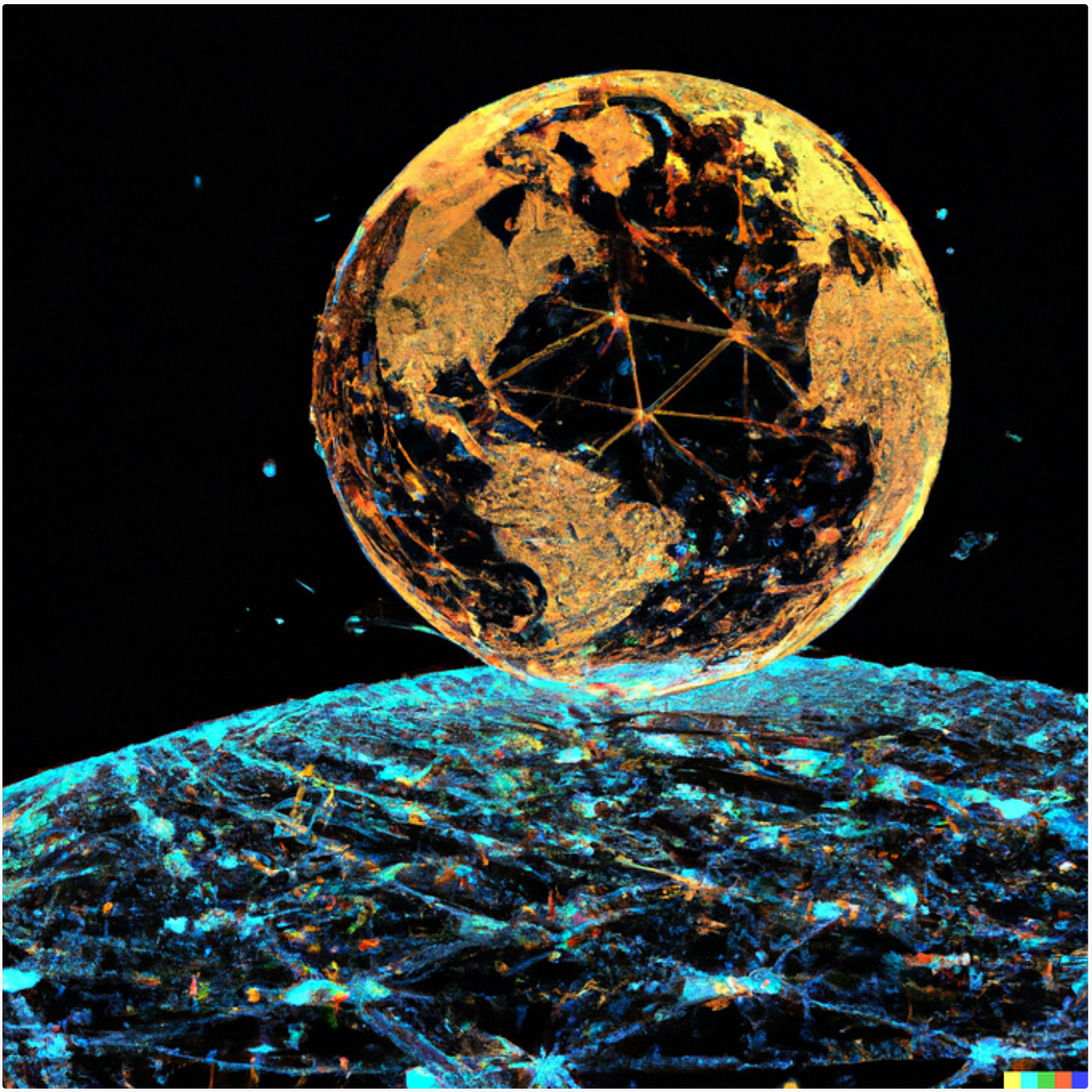
MrNewton in Hexmount

Solidity & Beyond: Zero-Knowledge Proofs

zkSync, roll-ups and all the good stuff explained!

5 min read · Jul 29

9



Abdus Salaam Muwwakkil in Rather Labs

The Top 10 Blockchain Trends in 2024 that Everyone Must Be Ready For

Exploring BaaS, DeFi Expansion, Scalability Solutions, and Regulatory Trends in Blockchain's Enterprise Adoption through 2024

Enterprise Adoption through 2024.

12 min read · Jul 18

133 8

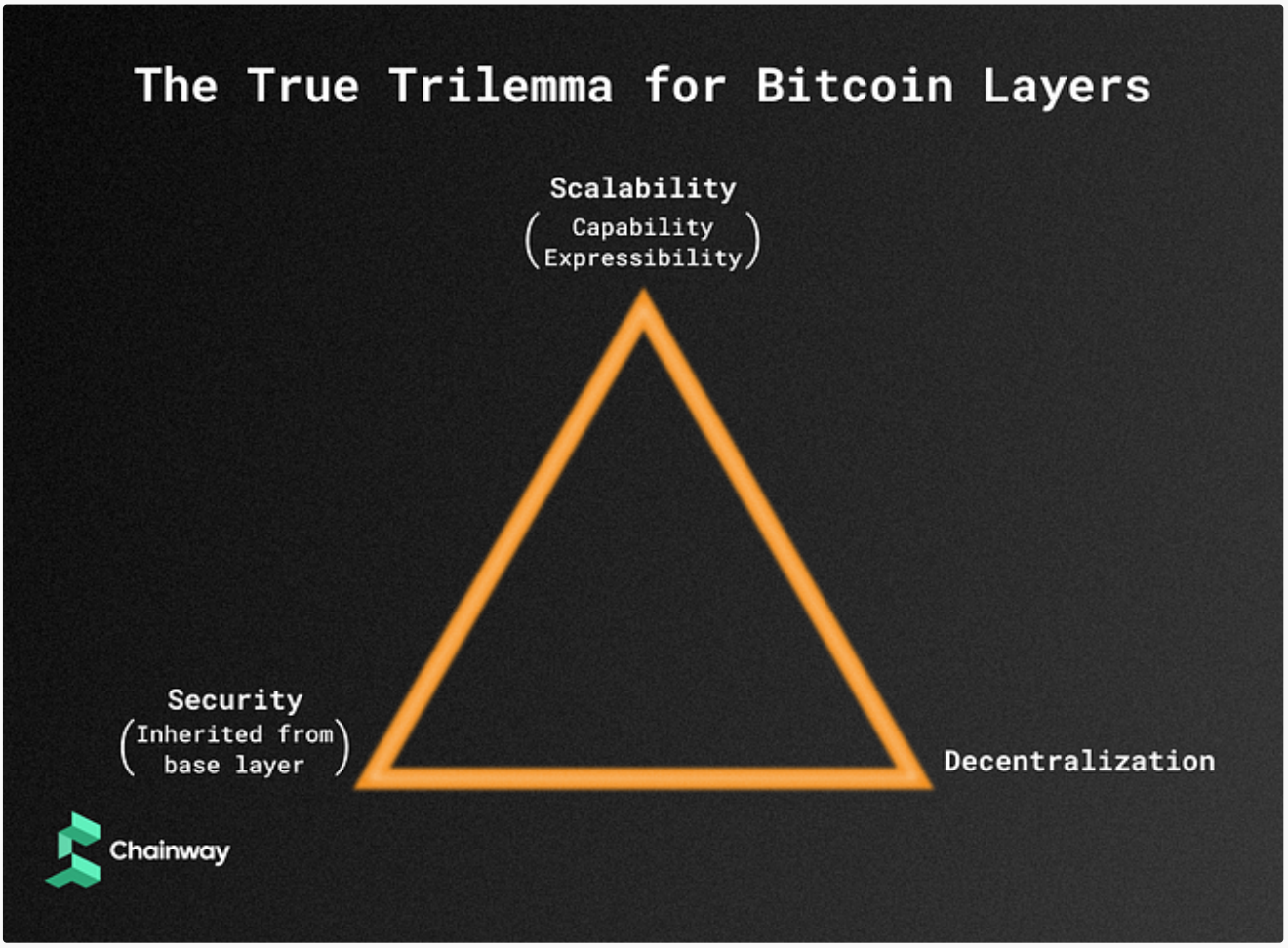
Sebastian Faron in Coinmonks

Mesh Security |Cosmos|

Cosmos Hub, ICS, and the current stage of work towards Mesh implementation in the

· 3 min read · Aug 8

339 1



Chainway

The True Trilemma for Bitcoin Layers

There has been a struggle in identifying and addressing the scalability trilemma for Bitcoin layers. Previous attempts to address Bitcoin...

13 min read · Nov 6

23

See more recommendations