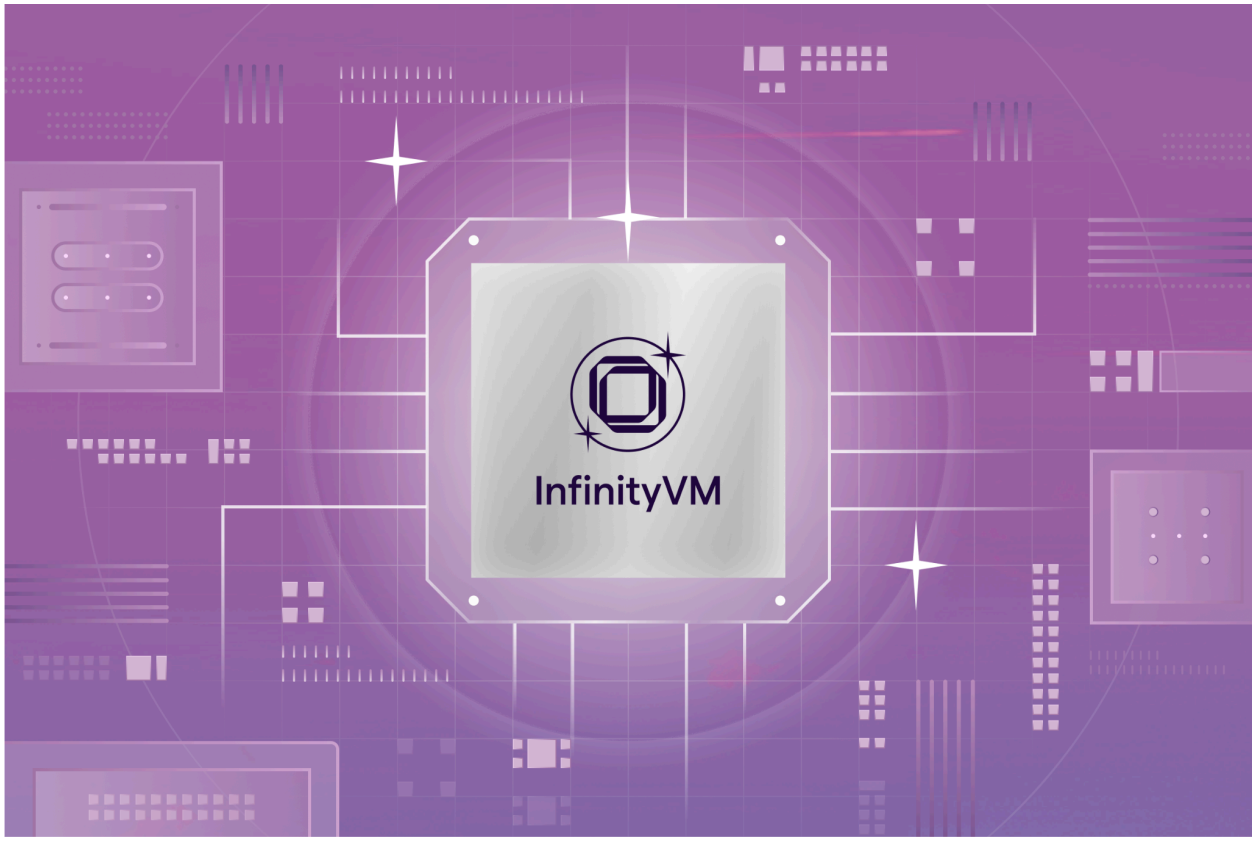
TLDR

InfinityVM is an extension to the execution model of the EVM. The InfinityVM runs as an *enshrined* execution sidecar alongside the EVM, enabling users to make requests for offchain compute to be completed in-parallel. Offchain servers can also fully orchestrate onchain application flows via coprocessing jobs, enabling composability with optimized compute.



The results of these offchain computations are then incorporated directly into the chain via an enshrined mechanism such that the fork-choice of the chain is defined by the correct result of the InfinityVM task. If an InfinityVM task is challenged, the chain will re-org to remove any blocks built with the incorrect result.

This leads to a fully consistent chain where all valid blocks must include the correct results of InfinityVM requests. This design provides a very similar UX as writing an EVM application: developers do not need to concern themselves with rollbacks because all transactions happen atomically with the honest state.

With InfinityVM applications can:

- Incorporate optimized server code into their onchain application
- Utilize offchain data
- Access historical state on any chain
- Scale up their application dynamically by ratcheting up InfinityVM task performance, not bounded by the chain they are deployed on
- Utilize specialized hardware to optimize their application
- Write code in their favorite language

Various applications are unlocked via these new capabilities: we hope this will extend the frontier of applications that could fit into EVM transactions by enabling any VM that is ZK-provable.

Motivation

The EVM has been the dominant model for turing complete compute onchain since its release in 2015. This dominance has largely been a product of the EVM's first mover advantage. While alternatives have been attempted, we believe the path towards unlocking new functionality is through supplementing the EVM rather than replacing it.

In order to accomplish this, we are taking advantage of the rapidly improving capabilities of ZK. ZK promises to enable blockchains to trustlessly verify arbitrary compute, enabling greater decentralization and efficiency.

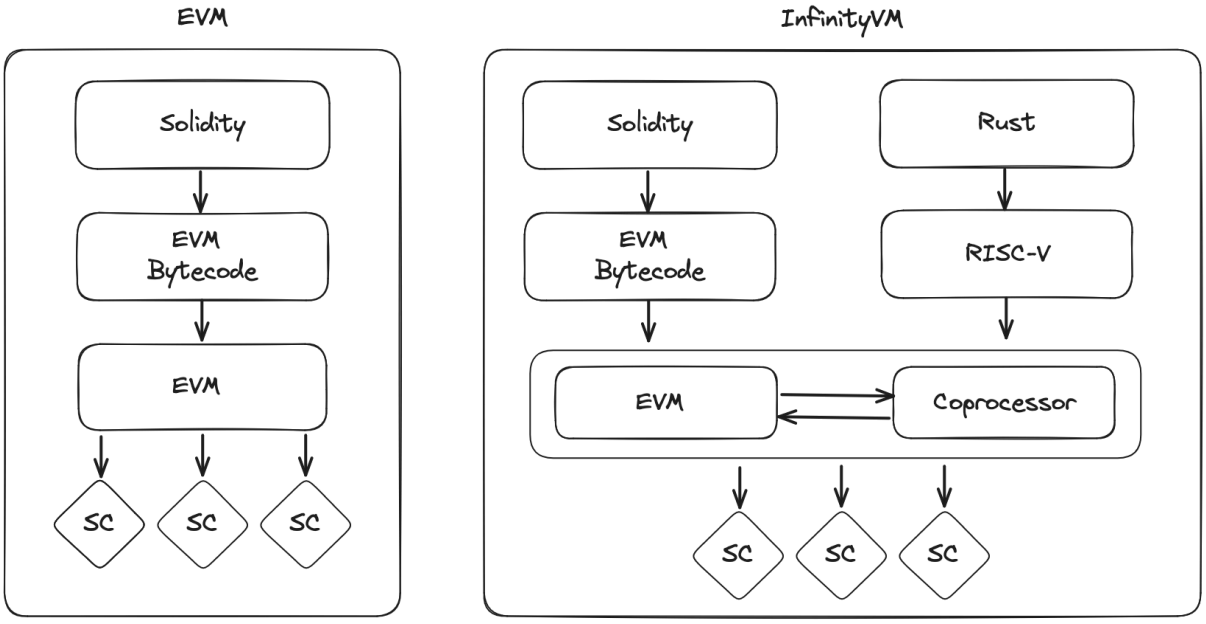
The inevitable trajectory of blockchains is towards further centralized compute with decentralized verification through ZK. We believe that for the blockchain ecosystem to further expand it must move beyond the strict limitations of both the EVM and the existing block structure.

We introduce the InfinityVM framework that enables us to frontrun the advances in ZK by utilizing the benefits of further expressivity and verification *without being burdened by expensive proofs with high latency*.

InfinityVM enables us to leverage the best of blockchains (censorship resistance, decentralization, and composability) with trustlessly validated, flexible and optimized compute, all secured through ZK.

InfinityVM

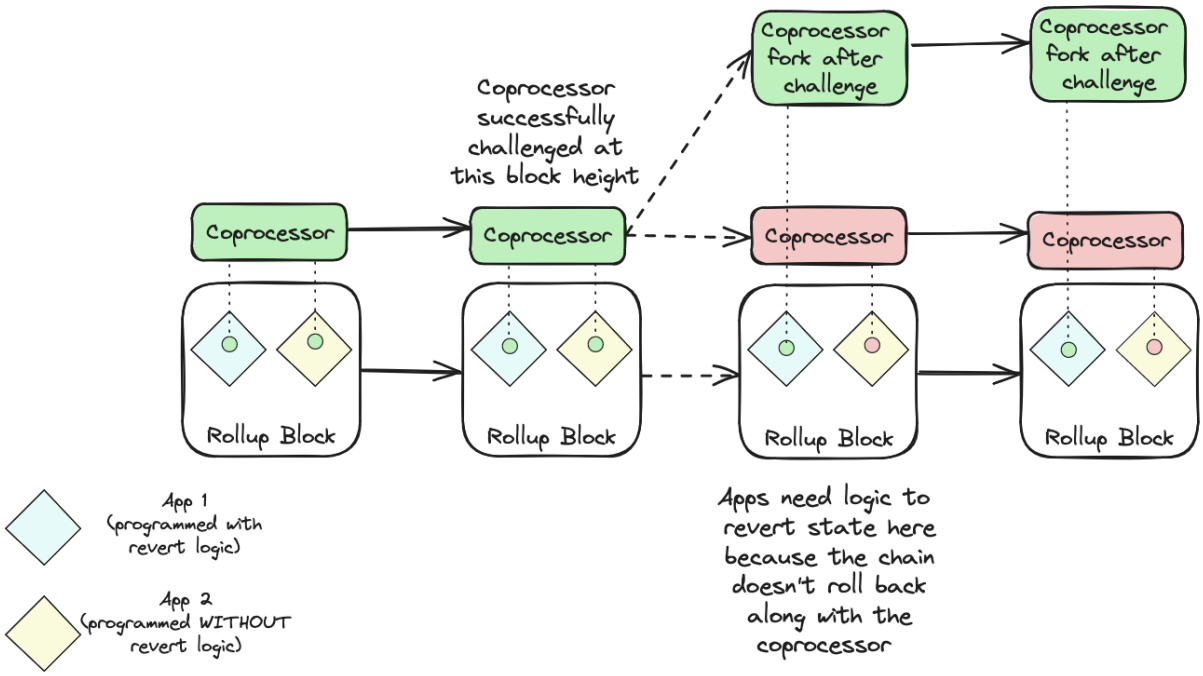
InfinityVM exists to introduce a new paradigm by utilizing the best of the EVM in combination with more expressive compute to unlock new onchain capabilities. The InfinityVM specifically is an execution sidecar process that a chain's validating nodes will run alongside node software.



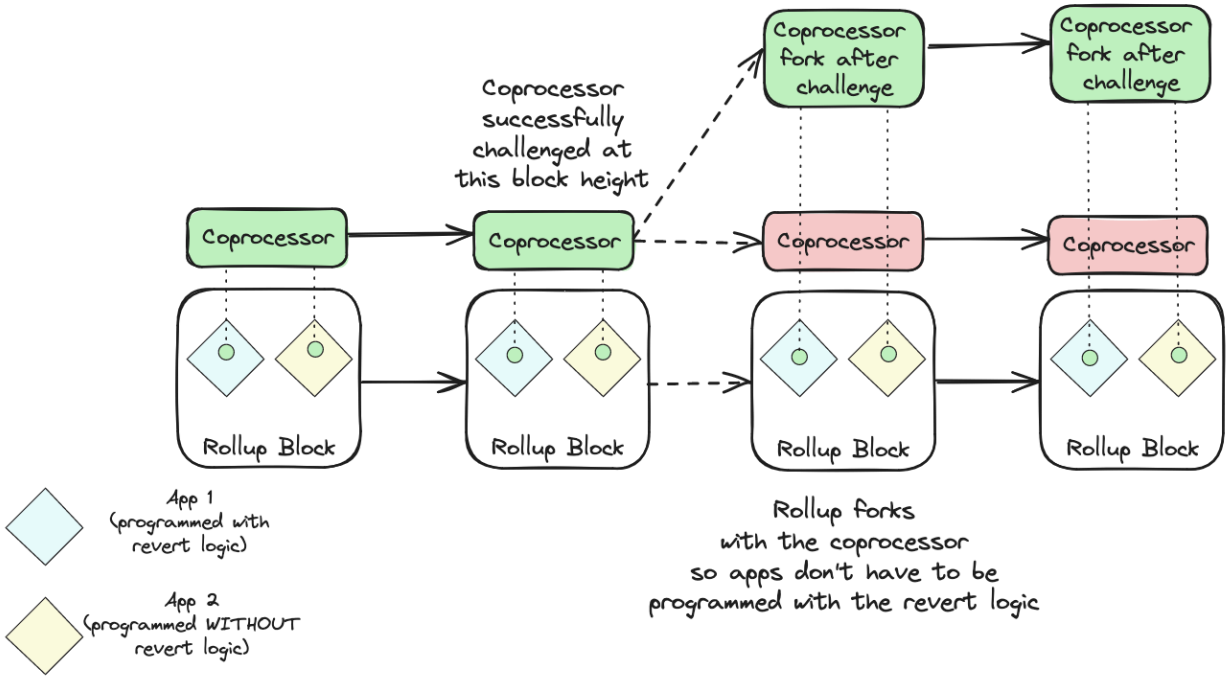
This process will expose a mechanism for users to request jobs that can run on given sets of inputs (including both onchain and offchain data) in order to either supplement their onchain application or to entirely run their application offchain. These jobs fundamentally consist of server-level code that is either run offchain in parallel (async) or synchronously with a requesting EVM transaction onchain. This mechanism can broadly be categorized as "**coprocessing**": dedicating a specialized and parallel offchain process to supplement the execution onchain via verifiable computation.

Coprocessing comes in many flavors; InfinityVM is specifically building an enshrined coprocessor. This works by bringing the coprocessor results *into the fork choice rule of the chain* itself. Existing implementations of coprocessing are primarily focused on bringing offchain compute into existing blockspace, inheriting the limitations of a system built for other purposes (e.g. Eth L1). This leaves coprocessing as an expensive but powerful supplement to applications, rather than a first class primitive to truly redefine applications.

Coprocessors without InfinityVM



Coprocessors with InfinityVM



Applications built on InfinityVM can come in two primary flavors:

1. Applications maintaining state onchain and using optimized offchain server compute to process complex logic
2. Applications maintaining state offchain in optimized servers but posting verified state commitments and transitions onchain

Verification

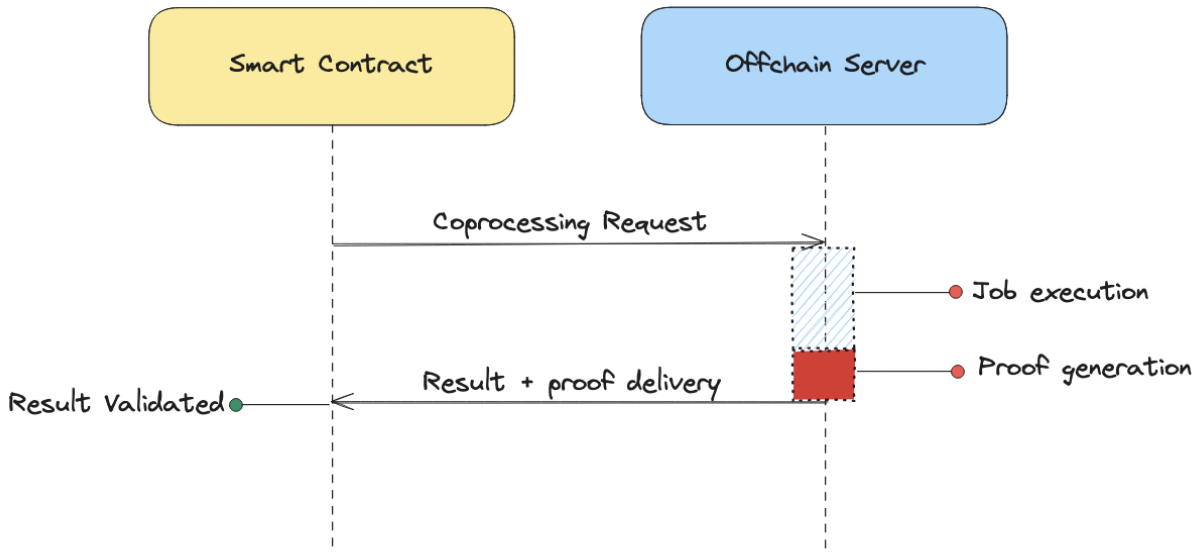
Bringing offchain compute onchain requires a robust verification system. There are various approaches to verification of compute, see the Appendix below for an enumeration of available options.

Verification	Onchain Latency	Offchain Latency	Cost	UX
Optimistic	High	Low	Low	Poor – devs must handle application level reorgs
Zk	Low	High	High	Great – No reversions, no trust assumptions
Enshrined	Low	Low	Low	Good – Reversion logic is abstracted

Because rollups and coprocessors are logically very similar, it should be unsurprising that the primary verification modes for coprocessors are optimistic and Zk, with their associated tradeoffs.

Optimistic verification relies on long fraud proof windows (often 7 days) for challengers to submit proofs, leading to delayed finality for interoperability between tasks. However, optimistic verification is the cheapest option as one user can attest to a result that can be implicitly (read: freely) verified after the fraud proof window. Optimistic verification also introduces complexity because of the potential for reversion if a result is proven fraudulent later.

Zk verification instead leverages validity proofs of correct execution, providing instant interoperability between proven results, but at the cost of proof generation. This cost is largely measured in time (proofs take time to generate, scaling logarithmically with the complexity of the task itself) but also incurs compute costs to generate the proof and to onchain costs to verify the proof.

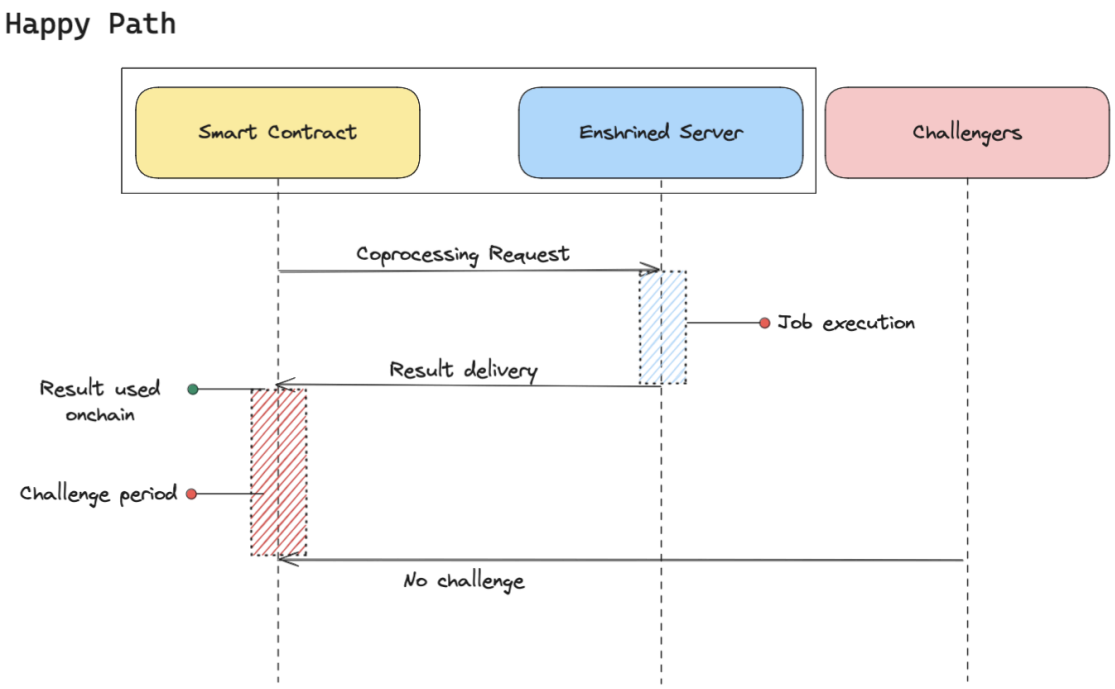


ZK Verification: Low latency onchain, High overall costs (proof generation + complexity + verification)

Enshrined Verification

InfinityVM is optimized to be efficient, low latency, and low cost by combining the best of each design in a unique way and absorbing the tradeoffs into the chain itself. We accomplish this by utilizing Zk fraud proofs and tying all fraud proofs together into the fork-choice of the chain itself, enshrining the coprocessor's verification mechanism.

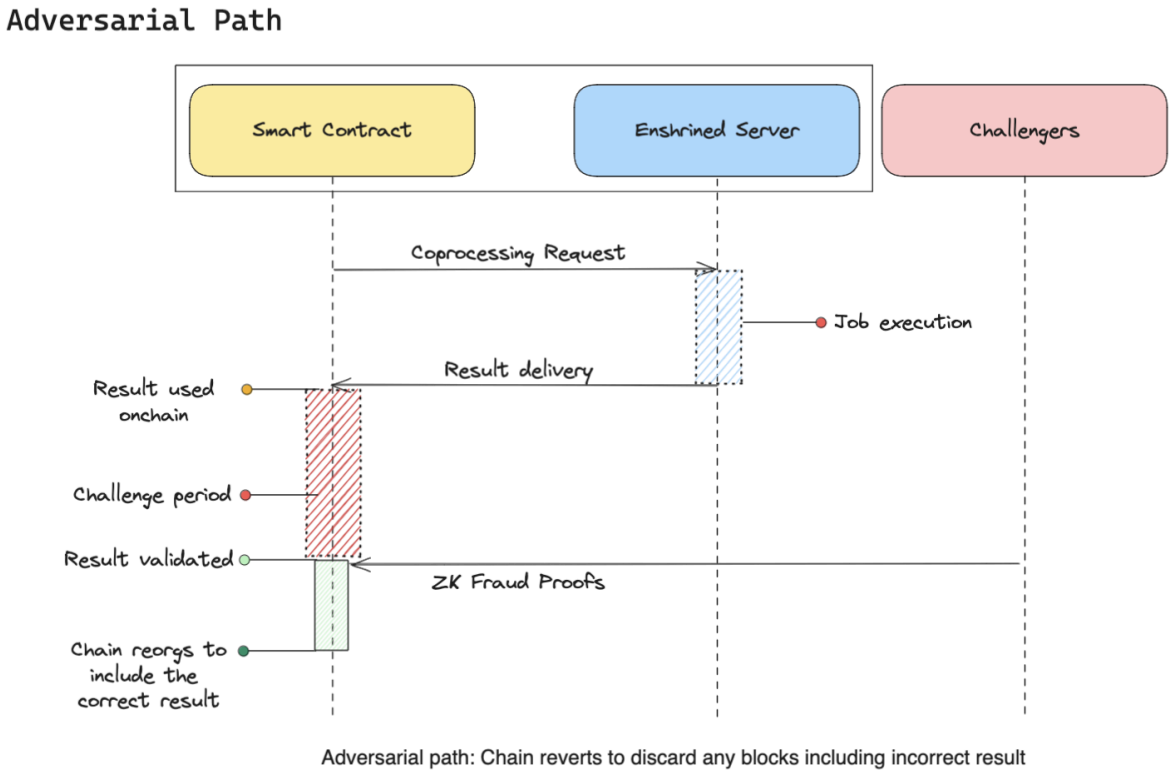
What this means is that similar to the canonical view of any other chain being defined by the honest execution of all VM transactions, the canonical view of the InfinityVM will include honest execution of any coprocessing tasks.



Happy Path: Users instantly use the results onchain

Zk fraud proofs enable InfinityVM to enable all computation that is Zk-provable: if you can define it in a ZkVM you can run it offchain securely and trustlessly on InfinityVM. Importantly, InfinityVM runs these tasks without the overhead of the prover, so these tasks can be optimized and run at near-native speeds. The prover is only engaged in the event of a fraud challenge.

Fraud proofs run with a designated **challenge period**: any user can audit the output of a coprocessing job and submit a challenge onchain to dispute the result. Once challenged, the challenger is given time to run the job and generate a Zk validity proof of the correct execution. Upon a successful challenge, the chain itself will reorg by invalidating any blocks built off of the incorrect coprocessor result. Importantly, all honest transactions that were included in these blocks can be immediately replayed in order to preserve UX.



Adversarial path: Chain reverts to discard any blocks including incorrect result

This mechanism ensures that all applications built on the InfinityVM will be derived from the same shared state and so they will all either succeed together or reorg together, enabling composability. This frees developers and users from needing to process the complex overhead of potential reversions at the application level, freeing them to consider all offchain actions the same way they consider EVM operations.

Replication Modes (who runs the coprocessor?)

Enshrining a coprocessor in this way also maintains ultimate optionality: validators of the chain can choose to run the chain in many different modes of verification. Simplest is for users to simply trust the sequencer to be honest; this includes strong trust assumptions but is lowest cost and often lowest latency. On the other end of the spectrum is for users or interop partners to run full validating nodes that repeat execution of all transactions and coprocessing jobs. This has virtually no trust assumptions (outside of DA) but is also least efficient. In the middle are two additional options.

First, a node can choose to only replay coprocessing for state they care about. We call this **local state verification**, as opposed to **global state verification**. With InfinityVM, users can choose to only verify state that is local to them (i.e. results of the coprocessing tasks directly affecting the state they are accessing). This gives users similar levels of trustless guarantees without incurring all of the costs of running the coprocessing network for transactions that are not relevant.

Second, nodes can utilize Zk the same way that a traditional Zk coprocessor might, posting eventual Zk proofs of correct execution on a lagging basis in order to provide trustless guarantees on state at low cost to the user. It is an important quality of the InfinityVM that at worst we can revert back to a Zk regime, but we are importantly not beholden to this mechanism for all actions onchain.

Execution

Utilizing the coprocessor begins with users submitting `Program`s to the **Coprocessor Network** (a network of nodes running the InfinityVM sidecar). These Programs look like any other program developed today to run on a server. They include optimized code to accomplish a particular purpose without needing to adhere to a strict framework of blockchain semantics (block/tx limitations). A Program will define the code it runs as well as the interface of the inputs and outputs it expects when run.

Once the Program is submitted, a user can request a Job to run this program over a set of inputs. This data is posted to the coprocessor network and also replicated on an external DA service to provide strong availability and verifiability guarantees.

Validators will then run this Job via the InfinityVM sidecar which operates as a parallel process to regular node consensus. This parallelization applies at a per-job basis, enabling these nodes to fully optimize execution via beefy execution machines on-demand per job. Jobs can be requested in two primary modalities:

First is asynchronous requests: these requests can be made either onchain or offchain and are a commitment from the user of their intention for a coprocessing job to run a specific Program with a given set of inputs and to post the result to a given smart contract's callback.

Alternatively, users may make synchronous requests to the coprocessor. This is a unique capability inherent to InfinityVM enshrining the coprocessor into the chain. A user in this case can send a transaction requesting a coprocessing result to be committed within the same transaction as the request. This works for execution that has fixed cycle limits in order to not affect average block times. The proposer then commits to the result in the same transaction which may then use that result in-process to continue further actions.

Regardless of the approach taken, a result is then posted to the requested smart contract's callback function (see the interface below) with an associated signature from the proposer of the result and the commitment to the job.

```javascript
function submitResult(
     // Includes job ID + inputs hash + max cycles + program ID + result value
     bytes calldata resultWithMetadata,
     bytes calldata signature
```

```
)
```

Roadmap

InfinityVM v1:

- Coprocessing network to generate fraud-provable results for submitted jobs
- Asynchronous coprocessing results enshrined in the chain
- Centralized coprocessor result proposers
- Centralized proposers give trusted guarantees on coprocessor result inclusion

InfinityVM v2:

- Synchronous coprocessing results enshrined
- PoA Coprocessor network
- Decentralized Coprocessor network with associated stake

InfinityVM v3:

- Decentralized enforcement of coprocessor result inclusion (enshrined censorship resistance of coprocessing jobs)
- Storage proofs integrated for historical state access in the coprocessor
- Light client implementation with lagging Zk proofs of execution

InfinityVM v4:

- Proof aggregation to enable fraud proofs across many chains and associated storage proofs
- Proof of Stake Coprocessor Network
- Support for additional coprocessing modes: TEEs and honest-majority

Discussion

Can you give me an example of a Coprocessing task?

Simple examples can include various existing coprocessing use cases, including AMM hook designs to governance proposal computations. However, the core of InfinityVM is unlocking new applications by embracing offchain compute, so we will explore an example of running a Central Limit Order Book (CLOB) via offchain compute.

This application flips the existing smart contract paradigm on its head: we run the bulk of the application offchain and only post commitments and verify state transitions onchain. This works by dedicating a server to run the CLOB offchain. This server can receive orders and cancellations and handles them like any centralized CLOB, enabling free order submissions + cancellations. Once an order is received or canceled, the CLOB server will return to that user a signed commitment to that particular order and the state of the order book at that time. These commitments are stored onchain, and the orders themselves will be made available via a cheap DA service (like EigenDA or Celestia).

Once an order is matched (computed offchain) the matching algorithm is actually what runs in the coprocessor itself. The InfinityVM will post the result of this matching (two orders buy/sell) onchain and start the challenge period. This matching will take the effect of any other CLOB, exchanging assets between two parties and taking a fee.

Any user can now verify the matching themselves, by constructing the order book from the DA layer and referencing the commitment. This can also be hosted by frontends for this application that host all orders and verify the commitments. Importantly, any user that submitted an order can now determine if their order was correctly matched or not. If a user is able to detect that the server violated price-time priority (either by ignoring their order or including a new order in its place) they can now challenge the matching result. This challenge would work by posting the signed commitment from the server showing that it received the user's order and that the state of the order book at that time included that order which should have been matched.

If this challenge is successful, the chain itself will then reorg to remove that matching tx and any txs built off of it, guaranteeing full security for traders.

We can then add some simple measures to enable censorship resistance (server rotation, forced inclusion of orders on the chain itself) to allow for a CR, trustless, and performant CLOB.

Importantly, this kind of application has existed in the past (see dYdX v3/v4) but relied on trusted parties (either dYdX in the case of V3, or their validators in V4) to maintain the order book offchain and to actually run the matching against their view of the orderbook.

How does this work from a validating node's perspective?

Nodes that are validating a chain with the InfinityVM will run this sidecar process in parallel to the existing node software that manages consensus for the chain. The InfinityVM process will maintain a list of all requested jobs and begin threads to execute them. As these jobs finish, the validators will come to consensus on the result and post the result to the callback function on the requester's smart contract. This successfully separates what we expect to be a very compute optimized + parallel process (the coprocessor) from the largely single threaded work of coming to consensus.

Full nodes may also operate in a fully validating mode, but are also able to verify the coprocessor state transitions selectively, we expand upon this more in the Replication Modes section above.

Who can submit Coprocessing results?

A malicious proposer of coprocessing results could cause arbitrary chain reorgs which would be very painful for users and could be used to extract MEV. Thus, we must be very careful with who we allow to propose results – we treat this similarly to how an optimistic rollup treats their state root proposer. To that end, the core validating nodes of the chain will be the only valid proposers of coprocessing results.

We will start by posting results ourselves from the proposer (similar to Arbitrum's sequencer), and then graduate to a trusted PoA set with reputational and economic bonds behind their commitments. Then we will finally graduate to a PoS network with significant bonds to incentivize honest behavior. See the roadmap for this plan.

Importantly, any validating node can also be rotated in the event that they are censoring or otherwise harming users. This can be done via a social/governance action and would likely require offchain coordination, but is crucial to ensure the system cannot be corrupted by malicious actors.

How does this compare to Alt-VMs?

The bulk of the innovation around execution models has been centered around alternative VMs like Move, SVM and the FuelVM. These all excel for their particular purposes, but are plagued by playing second fiddle to the EVM's first mover advantage. Each is also still subject to the inherent limitations of an onchain VM, namely that it must work within the bounds of a particular block with maximum gas limits. They must also maintain a simple VM that can be reproduced across many distributed machines with a minimum hardware profile that must be carefully managed.

Another approach shepherded by Arbitrum's Stylus is to incorporate an Alt-VM (in their case a VM over WASM) directly alongside the EVM, allowing Stylus to incorporate the EVM and its first mover advantage while also adding more expressivity. This is one of the modes that InfinityVM can run under, however, this approach is still subject to the bounds of the block.

What are all the ways coprocessing tasks can be verified?

1. **Traditional Verification**
   Blockchain VMs replicate computation across many nodes, ensuring state transitions are consistent and agreed upon. This method is secure but inefficient, as each node must either execute every transaction or trust a third party, and it is limited by gas and block size constraints.

2. **Optimistic Verification**
   A subset of nodes process computation and posts results onchain with a fraud period (e.g. 7 days) during which results can be challenged. Fraud proofs involve an interactive process to re-run parts of the contentious transaction, which, while effective, is complex and slow. Fraud proofs also often require emulation of operations onchain which can be complex and places the crux of the security into the efficacy of that emulation.
3. **Modified Optimistic Approach**
   To reduce latency, cryptoeconomics can be used to circumvent fraud proofs for demanding applications, imposing significant penalties on fraudulent proposers. Systems like stakesure can insure users against potential harm from fraud. However, this only works for applications that either have low stakes or can very accurately model potential user harm from fraud.
4. **Zk Fraud Proofs**
   Instead of interactive fraud proofs, we can use Zk proofs. Approaches here may vary, a simple design is to have challengers pinpoint a particular state transition and submit a Zk proof of the correct execution of that state transition, effectively bundling up the fraud proof into a single shot. This can shorten the fraud proof window significantly and encapsulates most of the complexity within the ZkVM implementation.
5. **Validity Proofs**
   Use Zk to prove the validity of computation at the time of posting results. This provides the strongest security guarantees and immediate finality. However, this approach introduces latency and cost incurred by generating these proofs for every task, especially with complex computations.

Each verification system exists on a spectrum of costs/benefits. We introduce these properties between verification tradeoffs:

1. Efficiency: how much work does each actor need to perform to verify results. This is highest for traditional VMs, constant for Zk and effectively 0 for optimistic approaches.
2. Proof latency: how quickly a result can be utilized safely onchain. This is highest for optimistic fraud proofs (7d) and shorter for validity proofs that are bounded by the time to run the prover for each task.
3. Proof cost: how expensive is the proof verification onchain. This is highest for Zk validity proofs and effectively 0 in the case of optimistic fraud proofs.
4. Reversion risk: Zk avoids all risk of reversion, while optimistic approaches must be able to revert fraudulent results, incurring additional complexity.

Why does this need to run as a sidecar process? Why not just extend the node software itself?

By running the InfinityVM in a sidecar process we are able to separate logical roles: the node process handles all of consensus while the InfinityVM process is able to handle execution extensions. This also means that these processes can run on separate (but likely colocated or physically connected) machines, enabling both to optimize their hardware for their particular roles.

From the perspective of the node client, InfinityVM job results look like oracle updates. This means the consensus model can remain entirely independent of any coprocessing, fully separating concerns and enabling more flexibility on both sides. This concretely means we can explore updating the InfinityVM or the chain's consensus independently, unburdening each from the complexities of the other. This will enable faster innovative iteration.

What are the tradeoffs?

No blockchain system exists without tradeoffs, there is no free lunch. The primary tradeoff we've made is in favor of further expressivity and optimized execution, at the cost of absorbing the potential re-org risk within the chain. This can lead to complications for interoperability for any transactions going offchain. However, we will argue in the sections below that this is not so different from existing rollups because anyone can run a full validating node.

A major factor in our decision to make this tradeoff is the rarity of reorgs and fraud today. This is of course not something we can count on going forward, particularly as we decentralize, but our point of view is that we should **optimize for the happy path**, while still securing against the worst case.

We're also requiring validating nodes to perform more work, but that is implicit in any scheme that is scaling up execution, and we think our approach enables the best form of scaling up hardware requirements. This is because hardware / performance requirements can be scaled *per-job* rather than on an entire chain basis. This means that validating nodes don't need to run a massive machine all the time, particularly when they aren't running high intensity coprocessing jobs.

As a motivating example, consider an application that required a massive MapReduce job once every 100 blocks that processed the last 100 blocks in memory and performed some analysis. Say this task required say ~100GB of memory to be performed optimally. In a naive approach, we would scale up all hardware requirements for all nodes to the worst-case scenario, which in this case means we need all validating nodes to always have access to 100GB of memory.

With the InfinityVM, though, jobs can be hyper specialized and include detailed specifications. Because these jobs run in parallel, any validating node can simply spin up a node with the necessary hardware (e.g. 100GB memory in this case) for this particular job and only utilize it to process this job. That way, we aren't wasting excess resources we don't need, but we are still able to scale to the needs of specialized applications.

What is the reversion overhead problem with an Optimistic coprocessor?

If we were to use an Optimistic coprocessor with a fraud period where results can be challenged and reverted, we are left with the question of what to do on reversion. Developers building applications that utilize offchain compute may not have a meaningful way within their application to handle reversions of state. Take as an example an application that will mint an NFT for any

user that has sent a transaction within the past year – they utilize a coprocessor to detect this as it requires knowledge of historical chain state. If a fraudulent proposer posts that user `U` did qualify for this, `U` will be able to mint that NFT. If a challenger then later is able to prove that `U` did not actually qualify, the developer of this NFT project must now encode the logic to either:

- Revoke the NFT from this user – this can greatly complicate the NFT contract itself because it is not otherwise possible to reclaim an asset that an EOA wallet controls
- Block that NFT from being usable/transferable – this now limits the expressivity of your NFT because now the cap of the NFT collection (say there are only 10000) now has to account for "revoked" NFTs and allow new NFTs to be minted, or to be stuck permanently at less than 10k total.

While either solution may be deemed appropriate for this particular situation, this nonetheless leaves the onus on the developer to devise an application-level solution to fraud reversions, increasing the burden on developers to develop and importantly to explain to users how their application works.

How does this compare to Rollups today?

This mechanism works similarly to a fraud proof for existing optimistic rollups today: if Optimism's sequencer were to post a malicious state root to the L1 it would eventually be fraud proven and result in removing the malicious state root. Any actors following the state given by the malicious rollup sequencer would need to adjust their view of the state after a successful fraud challenge to be the verified state and they would need to find a new trusted party to follow.

Importantly, any actor can also run a validating node to follow the rollup trustlessly by running each transaction themselves and verifying the state transition. This is also true with InfinityVM: any validating node can run both the EVM transactions and associated Coprocessing tasks to get a current trustless view of the state of the rollup. Crucially, we utilize an external DA service (like EigenDA) to ensure that anyone can access the necessary inputs to derive the honest coprocessing results. This is similar to how LPs work around OP rollups' fraud periods today.

Another important similarity is to how Ethereum rollups today are built off of the DA of Ethereum despite Eth L1 having eventual finality (after 2 slots, ~12 minutes). This is considered safe for rollups because the rollup state is *derived* from the state of Ethereum: if a rollup builds off of the state of the Eth DA at block `X` up to block `X + Y` and then Ethereum block `X` is reorged to block `X'`, the state of the rollup will also be reorged with it. The way this is handled varies by rollup, but the simplest solution is to simply replay honest transactions (batches of L2 transactions posted to the L1) onto the L1 again with the new block `X'`. The InfinityVM is built with a similar mechanism where the state of the next block is derived from the posted coprocessing results in the previous block. If this result is eventually fraud proven it will reorg the block including that state which will reorg all downstream transactions as well.