Sova: An EVM-Compatible Bitcoin Execution Environment

Kevin Kennis
kevin@sovalabs.com

Robert Masiello
robert@sovalabs.com

Evan Powlowsky
evan@sovalabs.com

October 2025

Abstract

Sova is a new blockchain designed to add programmability to Bitcoin. While Bitcoin secures the most global economic value of any blockchain, applications and utility beyond value storage and exchange via are limited. This is because the Bitcoin network itself does not host a programmable virtual machine. Sova introduces a new EVM-compatible blockchain with a set of precompile-based VM extensions, combined with validator-level bindings to Bitcoin Core, that enables a smart contract execution environment to interact with Bitcoin and its native assets, such as Runes and Ordinals. Built as an Optimistic Layer-2 on the OP Stack, Sova settles to Ethereum for fraud proof security and data availability while enforcing Bitcoin finality at the protocol level through the Inspector and Sentinel validation systems.

1 Background

1.1 Bitcoin as a Settlement Layer

One of a blockchain's functions is as global state synchronization machine, allowing multiple entities to subscribe to and coordinate the creation of updates to the state ledger. Different blockchains represent state in different ways; while Ethereum represents value in an account-based manner, using trie storage for the state associated with each particular account, Bitcoin uses a model such that the global state on the Bitcoin chain can be described completely by the set of all Bitcoin UTXOs[1]. A Bitcoin UTXO contains an amount of value (denominated in satoshis, the base unit of $BTC), and rules for who can spend the value.

The spending rules for Bitcoin UTXOs are expressed through Bitcoin Script[1], a non-Turing complete stack-based scripting language. Each UTXO has what is called a "redeem script", which must return true for a transaction spending that UTXO to be successful. Script can express rules that require specific signers for a given transaction input (`OP_CHECKSIG`), or multiple signers (`OP_CHECKMULTISIG`). Through use of various machine instructions (called "opcodes") which do not affect transaction security, authors of Bitcoin transactions can also push arbitrary unstructured data to the Bitcoin blockchain, a practice known as "inscribing" data to the chain[20].

Before inscriptions, data was often posted to the Bitcoin blockchain in (`OP_RETURN`) outputs, however this method was considered nonstandard, space-constrained, and data posted in outputs was subject to pruning[18]. Inscriptions, which emerged after the 2021 Taproot upgrade for Bitcoin[22], have proven a more robust and more accepted way to post data to the Bitcoin blockchain. While inscriptions provide a format for which data-storage opcodes should look like, Taproot allows complex Script rules to be written with sublinear effects to on-chain transaction costs, reducing the fee overhead of using inscriptions. Used together, Taproot and Inscriptions formed the technical foundation for the emergence of native forms of value on Bitcoin besides $BTC itself.

There are a number of factors which make inscribed tokens more scarce than similar abstractions on other blockchains. First, each inscription must be tied to a specific UTXO, which must contain at least one satoshi of value; Bitcoin's intrinsic

supply cap also imposes a supply cap on the number of UTXOs that can be inscribed. Second, inscribing data has a transaction cost correlated to the amount of data that needs to be inscribed. These factors of cost, complexity, and asymptotic finite supply all combine to create an ecosystem for on-chain assets with different economic and scarcity fundamentals compared to EVM, SVM, and Move-based asset ecosystems.

These capabilities, and limitations, suggest the potential for a specific type of non-native asset ecosystem on top of Bitcoin. Higher transaction costs for inscriptions enforce an ecosystem of assets that favor quality over quantity; creating an asset on Bitcoin requires an additional level of thoughtfulness and financial commitment compared to more scalable chains.

Where assets of value emerge, so does finance; the self-emergent nature of economies around systems of value is an inherent property of human coordination. As the proliferation of assets of value on the Bitcoin blockchain increases, so does the demand for a native economy.

There is also a strong argument that data inscription is good for the long-term health of the Bitcoin network. Bitcoin has known long-term security budget concerns: as the block subsidy continues to decrease through successive halvings, the profitability of proof-of-work decreases as constant hashpower inputs earn smaller and smaller block rewards.

For Bitcoin's long-term survival, miners must make lost block reward value up through transaction fees; given that $BTC itself has been outcompeted as preferred medium of exchange by tokens on higher-throughput blockchains, it's unlikely that Bitcoin's store of value function alone will provide these needed fees. A vibrant ecosystem of native assets which are stored and transacted on the Bitcoin blockchain is one of the network's best bets at creating demand for blockspace to fund the long-term security budget.

1.2 Programmability on Bitcoin

Bitcoin programmability is limited by design; historically, the Bitcoin community has favored robustness over flexibility. Given Bitcoin's position as the oldest and largest blockchain, it's likely that its functionality will continue to ossify, rather than moving in the direction of supporting native general-purpose execution. While proposals like BitVM[13] suggest the possibility of more native programmability, their path to adoption by the Bitcoin community and implementation timelines remain indeterminate.

In terms of technologies currently supported on Bitcoin, Script has historically provided limited flexibility, mostly allowing abstractions such as time-locked or multisignature signing schemes. However, the set of Script-supported opcodes is limited by design, and there is a large gap between Script capability and general purpose execution. Working within these limitations, the Bitcoin community has developed a number of abstractions, both protocol-supported and protocol-extrinsic (based on convention), which have made it possible to define high-level assets and other data-defined abstractions on Bitcoin.

1.2.1 Partially-Signed Bitcoin Transactions

Partially-signed Bitcoin Transactions were introduced in BIP 174 and updated further in BIP370[4][5]. These transactions allow forms of multi-party transaction coordination. Two main capabilities of PSBTs are:

- Allowing multiple parties to asynchronously sign multisignature inputs.
- Allowing multiple parties to each sign individual inputs to a given transaction which spends multiple inputs from multiple addresses.
- Enabling the transportation of partially-signed transactions without revealing key material to either observers or subsequent signers.

These capabilities can provide the underlying substrate for multi-counterparty financial primitives, such as multisignature wallets, shared escrow accounts, and atomic swaps of currency (by matching inputs and outputs and have each counterparty co-sign).

1.2.2 Inscriptions

Inscriptions are a way of introducing data to the Bitcoin blockchain, by using opcodes with no other defined meaning to define a delineated space in

which the transaction writer can post arbitrary data. Inscriptions were introduced in January 2023 by Casey Rodarmor[19], as a new convention, since the defined opcodes do not affect the semantics of a transaction.

Since Bitcoin uses a UTXO-based accounting model, inscribed data must be attached to a transaction output, such that the redeem script for the given output includes the inscribed data. Given that the importance of such an output is not derived from the inscribed data, inscribed outputs often have a value of a single satoshi. Due to this, inscriptions are often called "sats".

There is nothing in the Bitcoin blockchain itself that parses, interprets, or relies on the any inscribed data; rather, inscriptions are "conventional", similar to the use of `OP\_RETURN` in earlier iterations of Bitcoin programmability such as the Omni protocol[7]. It is the responsibility of wallet designers and application developers to detect when a wallet's spendable outputs might be inscribed, and how to interpret the inscribed data. Akin to common data-transfer protocols such as HTTP, inscriptions often define a MIME type[20].

Given that inscriptions support arbitrary data, they also can be used to express NFT-like or token-like asset abstractions, in the same manner that ERC20 tokens are defined by a balance ledger in an account-based model. In this way, the inscribed data itself can have an agreed-upon value and be used for economic activity. Potential applications exist beyond tokens and other units of value; for instance, a social network protocol can use inscriptions to record a user's social graph.

1.2.3 Ordinals

While inscriptions are a method of posting data to the Bitcoin blockchain, if such data is to be used as an asset supporting economic activity, that asset must be transferable and often divisible. However, UTXOs on Bitcoin cannot be "re-used"; by design of the Bitcoin protocol, any Bitcoin transaction always fully spends all value in any UTXO defined as input, effectively destroying the unspent.

Ordinal Theory is a solution to this, and allows the same piece of inscribed data to be tracked across multiple UTXOs and transactions in perpetuity. Ordinal theory relies on the fact that across the entire Bitcoin blockchain, UTXOs are unique and can be numbered: by the block the given satoshi came into existence and its index in that block. As long as its unit of value is kept in a single satoshi and not combined into a larger output, it retains its unique ordinal number.

Like inscriptions themselves, Ordinals are conventional; wallet and application designers must agree to follow the same rules of Ordinal Theory in order to agree on a common ownership ledger over data inscribed to Bitcoin.

1.3 Looking Forward

While there is no demand for general-purpose execution capability from miners and developers, there is demand from asset owners. In Ethereum, a significant portion of total on-chain value is locked in decentralized finance applications. As the total market for Bitcoin-native assets grows larger, so will the demand for on-chain utility and thus programmability.

2 Sova: an EVM for Bitcoin

Sova proposes a novel way to build programmability for Bitcoin assets: an EVM[14]-compatible blockchain with precompiles that bind to the Bitcoin chain. Application developers can write protocols using existing smart contract languages and frameworks (Solidity), and use global BTC bindings to handle asset settlement when required. While assets and related asset state, such as ownership, remain native to the Bitcoin blockchain through inscriptions, all protocol-related state and execution logic is managed by Sova. This gives users and developers the best of both worlds, allowing minimal switching costs for EVM developers to build on Bitcoin, and for users to engage with or enter the Bitcoin ecosystem for novel applications.

Sova is built as an Optimistic Layer-2 rollup that settles on Ethereum using the OP Stack. The network consists of three core layers: an execution layer (sova-reth) built on the Reth SDK with Bitcoin-specific modules, a consensus layer (op-node) for block sequencing and Ethereum settlement, and a validation layer (Sentinel and Inspector) that enforces Bitcoin finality within the EVM. Each validator runs a full Bitcoin Core node alongside their Sova stack, enabling direct verification of

Bitcoin state without relying on external oracles or bridges.

Sova's fundamental innovation is a set of new *precompiles* for the Sova EVM that allow on-chain interaction with Bitcoin through EVM smart contracts. The Sova precompiles will allow smart contract authors to:

- Read Bitcoin balances by address and UTXO
- Read owners of specific inscription IDs
- Construct and sign partially-signed Bitcoin transactions (PSBTs)
- Co-sign multi-party interactions with a key tied to the smart contract ("Network Signing")

Sova's smart contract environment is meant so that logic can be written on top of a Bitcoin settlement layer; as such, Sova supports Bitcoin precompiles that exist as analogues to EVM settlement actions, such as `ERC20#transfer`, `ERC20#transferFrom`, and `ERC721#transferFrom`.

2.1 Precompile Functionality

Sova provides a read-write interface to Bitcoin, within its smart contract environment, through the use of *precompiles*. Precompiles are built-in, predeployed global EVM smart contracts, with which any other smart contract can interact[21].

Use of precompiles to interact with underlying consensus layers or separate blockchains has been previously deployed at scale in Ethermint-based chains such as Evmos[9]. Using precompiles in this manner requires updates to the virtual machine and validator code. For Sova, validators are required to connect to a Bitcoin full node (Bitcoin Core), and use that node when processing transactions that make use of precompiles. For instance, query-based precompiles require the execution environment to read a value from the Bitcoin blockchain and return it inside the EVM, and write-based precompiles produce event logs that block builders must ingest and parse in order to interact with the underlying Bitcoin blockchain. A full list of precompiles is available in Appendix A, and as the demand for new low-level Bitcoin abstraction grows, precompiles can be added to the network via hard fork.

Validators must embed the latest Bitcoin block header in each Sova block through a Bitcoin Anchor contract deployed at address `0x2100000000000000000000000000000000000015`. This ensures that all validators are using the same canonical version of the Bitcoin blockchain. In addition, validators coordinate through a signing service to collectively sign and broadcast Bitcoin transactions that result from write precompile interactions. It is validators' responsibility to interpret interactions with the precompiled contracts according to the design and rules of the network.

2.2 Inspector and Sentinel

A critical component of Sova's architecture is the enforcement of Bitcoin finality at the EVM storage layer. This is accomplished through two complementary systems: the Inspector module and the Sentinel service.

The Inspector is a runtime module integrated directly into the SovaEVM (sova-reth) execution layer. When a transaction involves Bitcoin-linked state—such as minting sovaBTC tokens after a Bitcoin deposit—the Inspector locks the relevant EVM storage slots until the corresponding Bitcoin transaction receives sufficient confirmations. Locked storage slots remain readable but cannot be modified, preventing premature state changes that could lead to inconsistencies between Sova and Bitcoin. This locking mechanism is deterministic: given the same sequence of Bitcoin blocks, every validator independently arrives at the same locking decisions, ensuring consensus.

The Sentinel is an off-chain service that each validator operates alongside their execution client. It connects to the validator's Bitcoin Core node via RPC and tracks the confirmation status of all Bitcoin transactions relevant to Sova operations. When the Inspector needs to make a locking decision during transaction execution, it queries the Sentinel for the current confirmation count. Once a Bitcoin transaction reaches the required threshold (6 confirmations by default), the Sentinel notifies the Inspector, which then unlocks the associated storage slots and allows the EVM state transition to finalize.

If a Bitcoin transaction fails to confirm within 21 blocks (approximately 3.5 hours), the Inspector automatically reverts the locked storage to its pre-transaction state. This reversion mechanism ensures that sovaBTC is never minted without cor-

responding confirmed BTC deposits, maintaining the 1:1 backing guarantee.

Together, the Inspector and Sentinel create a system where Bitcoin finality is enforced at the protocol level, not through external attestations or bridge signatures. Every Sova state transition involving Bitcoin is verifiable by anyone running a Bitcoin node and Sova validator.

2.3 Client Signing

Since any transaction which uses a write-based Bitcoin precompile will result in the construction of a Bitcoin transaction, the Sova network must ensure that any constructed transaction can be fully signed. End-users interacting with Sova smart contracts that may create asset transfers should pre-sign inputs approving transfers, akin to `ERC0#approve`.

Bitcoin-native assets involved in Bitcoin transactions invoked by Sova precompiles may be controlled via different ownership schemes: assets can be owned directly by the user, be owned directly by the network, or be owned by a multisignature redeem script with signing keys controlled by a mix of end-user counterparties and network keys. In the first and last cases, any Bitcoin-level transaction will necessarily require end-user signatures.

Pre-signing inputs and transmission of partially-signed transactions is possible with Bitcoin PSBTs. Smart contract authors should design flows that require end-user signatures such that when initiating such a flow, one of the smart contract's function's parameters is the pre-signed, bytes-encoded PSBT required for asset settlement. The PSBT should be fully pre-signed for any inputs which are either directly controlled by the user or require a user signature on a multisignature redeem script. Smart contracts can decode bytes-encoded PSBTs into `struct` objects that can then be validated against transaction semantics.

In order to generate pre-signed signatures for Sova smart contract function arguments, the Sova ecosystem will provide SDK functionality that can be built into EVM-based wallets. The Sova ecosystem may design an environment-specific wallet for such a purpose, with a synchronized public/private keypair across the Bitcoin and Sova EVM environments. Users will be able to pre-sign any inputs required for underlying Bitcoin transactions using their private key, after which the wallet software can build the EVM-format transaction for the required smart contract interaction. The wallet software can then prompt the end-user to again sign the fully EVM-format transaction using the same private key.

2.3.1 Coin Selection

In situations where clients must pre-generate and pre-sign Bitcoin transaction signatures, client software to facilitate this process must also efficiently generate the needed transactions. One longstanding challenge of Bitcoin transaction generation is the concept of *unspent selection*[3]. Unspent selection is the process of selecting *which* spendable inputs to use from a client wallet in a generated transactions.

Some unspents may be more desirable than others among multiple lines; in general, using fewer inputs to a transaction leads to lower transaction fees, with each additional input increasing the transaction size on average by 60 vBytes[15]. However, long-term fee optimization over multiple transactions also involves change output consolidation; creating many change outputs may lead to the need to use more inputs in future transactions.

Client transction-building software should be flexible and support multiple unspent selction algorithms, both naive (e.g. largest first) and complex. One efficient unspent selection algorithm that can generally balance long-term fee optimization concerns is branch-and-bound[17].

2.4 Network Signing

Like in Ethereum-based protocols, in many use cases the protocol itself must custody assets and participate in settlement. In an account-based computation model, the smart contract iself owns the asset natively, and the smart contract's code defines how and under what conditions assets can be transferred or approved for transfer.

To make this possible for assets natively held on Bitcoin, each deployed smart contract must itself control a Bitcoin private key, and should be able to leverage that key via Sova precompiles.

The feature of each smart contract controlling a network key it can sign with is called *network signing*.

With these extensions to the base EVM, Sova smart contracts can control asset ownership outright and govern transfer of those assets according to smart contract code, analogous to smart contract asset ownership capabilities on the EVM.

Keys used for network signing are defined by a base *network key*, held be a BIP32 validator wallet. When a smart contract is deployed to a given hexadecimal address on the Sova EVM, the corresponding Bitcoin address derived from the same corresponding public/private keypair will be a network key. This can be enabled through a modification to the EVM's smart contract deployment process to always deploy to a deterministic address.

When deploying an EVM smart contract, the smart contract's address is usually determined by a combination of the deployer's address and account nonce. In the Sova EVM, the smart contract's address is determined by a *network nonce*. This nonce is globally incremented on each smart contract deployment. In addition to defining the address that a smart contract is deployed to, the network nonce also defines the final component of an HD derivation path from the root network extended public key. The key controlled by the smart contract then corresponds to this key generated by its globally-unique derivation path. Since nonces are global and smart contracts are public, there is a clear bidirectional mapping between an EVM smart contract address and its corresponding Bitcoin public key. By using public derivation paths, a smart contract's address can be computed without access to the network private key itself, such that block builders can assign newly-deployed smart contracts to the correct addresses without accessing private key material. This does not require any reflection of the smart contract deployment on the Bitcoin blockchain.

2.5 Key Resilience

The existence of a global network key, shared by validators, creates obvious attack vectors for assets held by Sova smart contracts. If the root HD key were compromised, any asset held by a smart contract on the network could be stolen.

In order to mitigate this attack vector, the ownership arrangements specific to each Sova smart contract can be defined by Bitcoin-native smart contract signature schemes. These schemes should be defined such that the network key corresponding to a given smart contract holds one of m required signing keys for a multisignature redeem script, but one or more counterparties to the application (end users) must also provide signatures in order to fully sign any related settlement transaction. For instance, in an Ordinals lending protocol, a smart contract may define a 2-of-3 escrow address, with the borrower, the lender, and the network each serving as a particular signer. When the borrower wants to repay a loan to recover collateral, the network can cooperate by providing its signature given that the borrowers has fulfilled the terms of the loan (signed a PSBT for repayment of owed funds).

Another advantage of multisignature schemes for smart contracts which own assets is resiliency to network liveness failures. In the case that the Sova network failed and was no longer able to provide its signature to recover an asset, end-user signers can independently construct a transaction to remove the asset from the multisignature address. In the Ordinals Lending use case described above, in the absence of network participation a borrower and lender can constitute the necessary quorum to recover assets from the escrow wallet.

2.6 Signing Service and Transaction Broadcasting

While Sova block times are approximately 2 seconds (following OP Stack defaults), Bitcoin proof-of-work block times are unpredictable and average around ten minutes. This imposes constraints on how Sova coordinates Bitcoin transaction signing and broadcasting.

When a write precompile is invoked—such as when a user burns sovaBTC to initiate a withdrawal—validators must coordinate to collectively sign the corresponding Bitcoin transaction. This is accomplished through the signing service, which validators operate alongside their other node components. The signing service uses threshold signature schemes (implemented via MPC, TSS, or traditional multisignature depending on configuration) to allow any threshold of validators to collectively produce valid Bitcoin signatures without any single validator controlling complete private keys.

The signing service maintains a queue of pending Bitcoin transactions that require validator signatures. Validators are economically incentivized

to participate in signing through fee sharing and are subject to slashing if they consistently refuse to sign valid transactions. Once enough signature shares are collected to meet the threshold, the fully signed Bitcoin transaction is broadcast to the Bitcoin network by multiple validators to ensure propagation.

The Inspector's storage locking mechanism coordinates with this process: when a sovaBTC burn transaction executes on Sova, the Inspector locks the burn-related state until the corresponding Bitcoin withdrawal transaction confirms on Bitcoin. This ensures that the on-chain audit trail accurately reflects the complete deposit-withdrawal cycle, with every sovaBTC mint and burn verifiably linked to confirmed Bitcoin transactions.

2.6.1 Transaction Requirements

Bitcoin transactions constructed through precompile interactions may fail under a number of conditions:

- The computed inputs do not exist on the Bitcoin blockchain.
- The computed inputs are already tracked as pending by the Inspector.
- The computed unsigned inputs cannot be signed by validators through the signing service.

When a corresponding transaction cannot be properly constructed, the corresponding Sova EVM transaction should fail and no EVM state changes may occur.

In addition to honest construction of Bitcoin transactions to reflect the semantics of the Sova EVM transaction, validators must ensure for all Bitcoin transactions that:

- The transaction follows the BIP370 format for partially signed bitcoin transactions[5].
- All inputs not requiring validator signatures have been signed by users.
- Fees are not zero and have been set to a reasonable value.
- Change outputs have been determined and properly accounted for.

Failure to meet this set of constraints may result in validators being slashed. The Sova node itself supplies fee calculation and unspent selection algorithms, which can be tuned by node operators.

2.7 sovaBTC and Bitcoin Finality

sovaBTC is Sova's canonical Bitcoin representation, maintaining 1:1 backing with real BTC through a protocol-enforced finality mechanism rather than traditional bridge custody.

When a user deposits BTC, they send Bitcoin to a Sova-controlled address and initiate a mint transaction on Sova. Validators observe this Bitcoin transaction through their Bitcoin Core nodes, and the Sentinel service begins tracking its confirmation status. The Inspector immediately locks the EVM storage slots associated with the pending mint, preventing any state changes to the user's sovaBTC balance or the total supply.

After the Bitcoin transaction accumulates 6 confirmations (approximately 60 minutes), the Sentinel notifies the Inspector of finality. The Inspector then unlocks the storage slots, and sovaBTC is minted 1:1 to the user's address on Sova. If the Bitcoin transaction fails to confirm within 21 blocks (approximately 3.5 hours), or becomes invalid due to a reorganization, the Inspector automatically reverts the locked state and no sovaBTC is minted.

For withdrawals, users burn sovaBTC on Sova and specify a Bitcoin withdrawal address. The burn is processed immediately on Sova, but the Inspector locks the burn state until validators complete the withdrawal. Validators coordinate through the signing service to sign and broadcast a Bitcoin transaction sending BTC to the user's specified address. Once confirmed, the withdrawal transaction is linked on-chain to the burn event, completing the verifiable audit trail.

This design ensures that sovaBTC supply exactly matches confirmed BTC deposits at all times. The validity of every sovaBTC in circulation can be independently verified by anyone running a Bitcoin node and checking the on-chain mint/burn logs against Bitcoin transaction confirmations.

3 Network Security

3.1 Consensus

The Sova network is an Optimistic Layer-2 built on the OP Stack, settling to Ethereum mainnet for fraud proofs and data availability. This architecture combines Bitcoin verification for asset finality with Ethereum's fraud proof security for execution integrity.

Like other OP Stack rollups, Sova currently operates with a single sequencer responsible for ordering transactions and producing blocks. The sequencer aggregates transactions from the mempool, executes them against the current state, and periodically posts state roots and transaction data to Ethereum L1. These posted state roots are subject to a challenge period during which anyone can submit a fraud proof if the sequencer produced an invalid state transition. This fraud proof mechanism, inherited from the OP Stack, provides the base layer of execution security without requiring validator staking on Ethereum.

Sova's distinguishing feature is the additional Bitcoin verification layer. All nodes—including the sequencer and any verifying nodes—must run a complete stack consisting of four components: Bitcoin Core (for Bitcoin verification), sova-reth (the execution client), op-node (the consensus client), and Sentinel (for Bitcoin monitoring). This ensures that every block not only conforms to EVM execution rules but also maintains consistency with Bitcoin's canonical state.

The sequencer is subject to the following block production requirements:

- Transactions must be honestly populated from the mempool and executed according to EVM rules.
- Interactions with Bitcoin-query precompiles must return accurate results verified against the local Bitcoin Core node.
- Each interaction with Bitcoin-write precompiles must trigger appropriate Inspector locking and Sentinel tracking.
- The Bitcoin anchor must correctly reference the canonical Bitcoin chain as observed by the sequencer's Bitcoin Core node.

Each Sova block includes critical Bitcoin synchronization data: the block header of the current tip of the Bitcoin chain, embedded through the Bitcoin Anchor contract at address `0x2100000000000000000000000000000000000015`. Verifying nodes independently validate this anchor against their Bitcoin Core nodes and can challenge blocks with invalid or inconsistent Bitcoin references through the standard OP Stack fraud proof mechanism. This subordinates Sova's consensus to Bitcoin's proof-of-work chain—Sova cannot finalize state that conflicts with Bitcoin's canonical history.

As the network matures, governance may transition to a decentralized sequencer set, similar to the decentralization roadmaps of other major OP Stack rollups. However, Bitcoin finality enforcement through Inspector and Sentinel remains a protocol-level requirement regardless of sequencer centralization, as every validator independently verifies Bitcoin state through direct node connections.

3.2 Validator Incentives and $SOVA Staking

While Sova's execution security derives from Ethereum's fraud proof system (inherited via the OP Stack), honest operation of the network—particularly the enforcement of Bitcoin finality rules—requires additional incentive alignment among validators and node operators.

The $SOVA token serves as the native asset for validator participation and network governance. Validators who wish to participate in block verification and signing service operations must stake $SOVA tokens. This staking mechanism is internal to the Sova network and serves to align validator incentives with network health, but does not provide the base layer security (which comes from Ethereum fraud proofs) or Bitcoin finality (which comes from deterministic verification).

Validators earn $SOVA through transaction fees and block rewards, incentivizing them to:

- Maintain accurate Bitcoin Core synchronization for proper Inspector and Sentinel operation
- Participate honestly in the signing service for Bitcoin transaction coordination

- Validate that blocks conform to both EVM execution rules and Bitcoin finality requirements
- Contribute to network liveness and availability

As the network evolves toward decentralized sequencing, $SOVA staking will play an increasing role in validator selection and reputation. However, even with a single sequencer, Bitcoin finality enforcement remains deterministic: any node operator can independently verify that sovaBTC supply matches confirmed Bitcoin deposits by replaying state transitions against their local Bitcoin Core node.

3.3 Slashing Conditions

Beyond the fraud proof security provided by the OP Stack's challenge mechanism on Ethereum L1, Sova implements internal slashing conditions for staked validators. These conditions specifically address Bitcoin-related misbehavior and are enforced within the Sova validator set through the $SOVA staking mechanism, complementing but distinct from Ethereum's fraud proof system.

The following Bitcoin-specific slashing conditions apply to Sova validators:

- ***Failure to keep Bitcoin node synchronized.*** Blocks published on Sova must contain the Bitcoin block header at the time of publication. Publishing a different block header from the canonical chain may result in slashing. Honest publications of the canonical chain which are later re-orged away will not be slashed.
- ***Failure to return accurate Bitcoin-query results.*** Blocks published on Sova must contain a "query log"; this is a mapping of EVM transactions which interact with Bitcoin-query precompiles, the corresponding queries made to the Bitcoin node by the validator, and the response data returned to the EVM. Inaccurate response data is subject to slashing. This slashing mode should incur a high penalty, as it may lead to inaccurate state transitions on the Sova EVM chain.
- ***Failure to include Bitcoin-write transactions.*** Blocks published on Sova must contain a "execution log"; this is a mapping of EVM transactions which interact with Bitcoin-write precompiles. If a published EVM transaction included a call to a Bitcoin-write precompile, but a corresponding Bitcoin transaction was not published in the execution log, the validator is subject to slashing. Validators can also be subject to slashing for publishing an entry in the execution log with an inaccurate execution queue ID.
- ***Mishandled Bitcoin-write transactions.*** Bitcoin-write precompiles should clearly define the semantics of what should happen on Bitcoin when the corresponding EVM transaction is processed. If the execution log publishes a transaction that does not accurately match the semantics of the Bitcoin-write precompile, the validator is subject to slashing. This slashing mode should incur a high penalty, as it may lead to unapproved transfers of assets on the Bitcoin blockchain.
- ***Failure to validate Bitcoin-write transactions.*** Transactions that are built, included in the execution log, and sent to the Execution queue must be fully signed for all inputs not owned by the network. If a transaction is not fully signed, it should not be included on the execution queue, and is a similar class of failure as the failure to include the transaction at all. Validators must validate signatures at block-building time and cause EVM transactions to fail if external inputs are not signed.
- ***Bitcoin input double-spending.*** As described in Section 2.5, inputs that are placed on the Sova execution queue are flagged until confirmation. In some cases, an EVM transaction may attempt to operate on flagged inputs. These transactions should be caused to fail by the validator; if a validator includes and honestly processes such a transaction, and adds duplicate inputs to the Sova Execution Queue, they are subject to slashing.

3.3.1 Slashing Proofs

Within the Sova validator set, validators can challenge other validators for violating Bitcoin-specific slashing conditions. Each validator has access to

the full data needed to verify Bitcoin-related behavior: the Bitcoin blockchain (via their Bitcoin Core node), the Inspector state, and the Sentinel tracking logs.

To challenge suspected misbehavior, a validator should replay the questioned block's execution, comparing their locally-generated query logs and execution logs against those published by the original block producer. If discrepancies exist—such as incorrect Bitcoin query results, missing Bitcoin-write transactions, or improper Inspector locking—the challenger can submit proof of the violation.

This internal challenge system is distinct from the OP Stack's fraud proof mechanism on Ethereum L1, which addresses EVM execution correctness. Bitcoin-specific slashing addresses validator duties unique to Sova: maintaining Bitcoin node synchronization, honest precompile execution, and proper coordination with Inspector and Sentinel systems. Slashing penalties are applied to the validator's staked $SOVA and are determined by the severity and frequency of violations.

3.3.2 Finality

Sova's finality model operates on two independent timescales, each addressing a different aspect of state validity:

Bitcoin Finality (6+ confirmations): sovaBTC mints and burns are tied to Bitcoin transaction confirmations. The Inspector enforces a 6-confirmation threshold (approximately 60 minutes) before unlocking EVM storage slots related to Bitcoin deposits. If a Bitcoin transaction fails to confirm within 21 blocks, the Inspector automatically reverts the pending state. This finality constraint is deterministic and enforced at the protocol level—no validator can override it.

Execution Finality (OP Stack fraud proofs): Like other Optimistic Rollups, Sova blocks achieve finality on Ethereum after the challenge period (typically 7 days) during which anyone can submit fraud proofs to dispute invalid state transitions. This fraud proof mechanism secures the integrity of EVM execution and ensures that the Ethereum L1 chain correctly reflects Sova's state.

Bitcoin finality is probabilistic, as Bitcoin itself provides only probabilistic finality through proof-of-work[16]. Sova inherits this constraint: if a Bitcoin reorganization invalidates a previously-confirmed deposit, the Inspector's deterministic replay will detect the inconsistency. The signing service coordinates with the Inspector to handle such cases, potentially requiring Sova state adjustments to maintain synchronization with Bitcoin's canonical chain.

The sequencer must ensure that Bitcoin-write transactions are not broadcast to Bitcoin until the corresponding Sova block has sufficient settlement assurance. This prevents scenarios where Bitcoin transactions confirm but their originating Sova transactions are challenged and reverted. The timing coordination between Inspector locking, block finalization, and Bitcoin broadcasting is handled by the signing service as part of normal network operation.

3.4 MEV Mitigation

Given the finality constaints described above, it's likely that economic activity on Sova creates the opportunity for Maximum Extractable Value, or MEV[12].

Given that Sova is an EVM chain, Sova will present similar MEV opportunities to those on other EVM chains with economic activity, such as Ethereum mainnet. However, state synchronization with Bitcoin through the Execution Queue, finality constraints, and the smart contract deployment flow described in Section 2.4 create new classes of possible MEV.

3.4.1 Contract Deployment Frontrunning

Given that deployed contract addresses can be precomputed based on the global nonce, parties may be motivated to try to deploy their contract to a specific nonce, or manipulate the nonce other contracts are deployed under. It is possible that given a pending deployment to a certain address, that deployment could be frontrun such that the originally intended nonce gets pushed one back, and the smart contract associated with the target nonce is replaced by one containing malicious code.

3.4.2 Bitcoin State Frontrunning

Frontrunning opportunities exist when Bitcoin transactions are pending confirmation. While the Inspector's locking mechanism prevents double-spending of inputs within Sova, it's possible that malicious users could attempt to double-spend inputs through methods extrinsic to Sova (such as broadcasting competing transactions directly to Bitcoin miners). The 6-confirmation threshold for finality mitigates this risk, but applications dealing with high-value operations may wish to require even deeper confirmation depths.

Outside of network-level halting attackers precipitated by users, the economic protocols enabled by Sova can themselves create frontrunning opportunities: both garden-variety within-block MEV, and across multiple blocks with assistance from other Bitcoin-native protocols outside of Sova. In the former case, it's important that Sova protocols include common mitigation mechanisms such as offer deadlines, slippage protection, and signature expiry dates.

In the latter case, mitigation is more nuanced given that other Bitcoin-native protocols can exist outside Sova. For instance, an observer of the execution queue might notice buying activity for one particular Bitcoin-native asset in a set of transactions on the Bitcoin-native execution queue. This creates an arbitrage opportunity for the observer, who can use external marketplaces such as Magic Eden to buy up assets in question before the Sova-native transactions broadcast and affect asset prices across all markets.

3.4.3 Mitigations

Like on Ethereum itself, there are promising lines of research to minimize externalities and user loss caused by MEV. These include private mempools (or priate Execution Queues) and internalization of MEV by validators. Both of these techniques as well as other lines of research can and should be implemented on the Sova network.

4 Economics

Like other OP Stack Layer-2 rollups, Sova uses ETH as its gas token for transaction fees. Users pay transaction fees in ETH when submitting transactions to the Sova blockchain. These fees are collected by the sequencer when a given transaction is included in a block.

The $SOVA token currently serves as the network's governance token. In the future, as the network transitions toward decentralized sequencing, $SOVA will be required for validator participation. Under this planned validator model, validators will be required to stake a minimum amount of $SOVA to be eligible to participate in block production and signing services. Staked validators will earn rewards from both transaction fee revenue and potential $SOVA emissions, while facing slashing penalties for provable misbehavior such as signing invalid Bitcoin transactions, equivocation, or extended downtime. This future staking mechanism will incentivize validators to maintain proper Bitcoin Core synchronization, participate honestly in the signing service, and contribute to network liveness.

4.1 Token Properties and Design

$SOVA is a fixed-supply, non-inflationary token designed to align long-term incentives across the network. With a total supply of 21 billion SOVA (fixed cap), the token has no ongoing inflation—minting is disabled after genesis, making the supply permanently immutable. Network rewards and validator compensation come from transaction fee revenue and usage activity, not from token issuance.

- **Network Security:** Secure the network through validator staking and slashing mechanisms (planned future implementation).
- **Governance Coordination:** Coordinate governance across protocol upgrades, validator set changes, and parameter adjustments.
- **Treasury Funding:** Fund ongoing network development through an automated Treasury mechanism, subject to governance approval.
- **Ecosystem Alignment:** Maintain long-term alignment between on-chain activity, validator performance, and Bitcoin adoption metrics.

The $SOVA token serves as the network's governance token and is used for validator participation. Validators earn fees in ETH from transaction

processing and may also earn $SOVA rewards for validator duties when staking is active. Staking rewards incentivize validators to maintain proper Bitcoin Core synchronization, participate honestly in the signing service, and contribute to network liveness. As the network transitions toward decentralized sequencing, $SOVA staking will play an increasing role in validator selection and block production rights.

Beyond end-user and validator economics, validators must coordinate to hold enough $BTC to be able to pay fees to the Bitcoin network when broadcasting transactions through the signing service. End-users can specify higher fees to be taken from pre-signed inputs in order to incentivize faster inclusion both on the Bitcoin network and for Sova validators.

In order to make the signing service economically viable, the $BTC spent during the signing process must be offset by per-transaction fees paid by users. As such, per-transaction fees will be split between validators and the signing service operations. Failure to maintain sufficient $BTC balances would equate to a liveness failure of the network.

In summary, the requirements and economics of the principals of the network can be described as below:

- ***Users:*** pay per-transaction fees in ETH for gas in return for the utility the network provides. Opt-in to providing $BTC fees directly for transaction prioritization.
- ***Validators:*** earn fees in ETH from transaction processing. In the future staking model, validators will be required to stake $SOVA and will earn $SOVA staking rewards while facing slashing penalties for misbehavior. In return, validators provide the computational services required by validation and maintain Bitcoin finality enforcement through Inspector and Sentinel operations.
- ***Signing Service:*** earns a portion of transaction fees in ETH in return for coordinating Bitcoin transaction signatures and the subsidy of $BTC fees for transactions signed and broadcasted.

Given these requirements, it is clear to see the required components of the economic model of the network:

- ***Users*** must receive more utility from the network than their per-transaction costs.
- ***Validators*** must earn more in staking rewards and transaction fees than overhead costs (and opportunity cost) to validate the network.
- The ***Signing Service*** must earn more in transaction fees and other subsidies than overhead costs and $BTC cost to participate in signature coordination and transaction broadcasting services.

More economic research is needed to determine the optimal inflation rates, transaction fee models, and methods of minimizing operation costs for validators and the signing service. In a long-term economically viable network, each party responsible either supplying blockspace or providing demand from it must stand to make a profit for participating.

5 Conclusion

Sova represents a new approach to Bitcoin programmability, introducing an EVM-compatible execution environment that operates directly on native Bitcoin assets. By building on the OP Stack and integrating Bitcoin finality at the protocol level through the Inspector and Sentinel systems, Sova enables developers to create sophisticated financial applications using Runes, Ordinals, and BTC itself without relying on bridges or wrapped assets.

The network's architecture addresses fundamental challenges in Bitcoin programmability: the Inspector enforces deterministic state transitions tied to Bitcoin confirmation depth, the Sentinel tracks Bitcoin finality across the validator set, and Bitcoin-aware precompiles expose Bitcoin network functionality directly to smart contracts. This design enables use cases ranging from decentralized exchanges and lending protocols to more complex financial primitives, all while maintaining direct ownership of Bitcoin-native assets.

As Sova evolves toward decentralized sequencing and validator-based consensus, the $SOVA token will play an increasingly important role in network governance and validator participation. The network's economic model is designed to balance

the needs of users, validators, and the signing service, ensuring long-term sustainability while keeping Bitcoin assets secure and accessible.

Sova's contribution to the Bitcoin ecosystem extends beyond technical innovation. By providing a programmable layer that respects Bitcoin's security model and finality guarantees, Sova creates infrastructure for a native Bitcoin economy—one where Bitcoin and its associated assets can be used as collateral, traded in decentralized markets, and integrated into complex financial protocols without leaving the Bitcoin security domain.

References

[1] A.M. Antonopoulos. *Mastering Bitcoin*. O'Reilly, 2014. URL: https://books.google.com/books?id=h_zToAEACAAJ.

[2] aslikaya. Proof-of-stake rewards and penalties. URL: https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/rewards-and-penalties/#penalties.

[3] Bitcoinops. Coin selction. URL: https://bitcoinops.org/en/topics/coin-selection/.

[4] Ava Chow. Bip-0174. URL: https://github.com/bitcoin/bips/blob/master/bip-0174.mediawiki.

[5] Ava Chow. Bip-0370. URL: https://github.com/bitcoin/bips/blob/master/bip-0370.mediawiki.

[6] EigenLayer. Mainnet launch announcement: Eigenlayer ∞ eigenda. URL: https://www.blog.eigenlayer.xyz/mainnet-launch-eigenlayer-eigenda/.

[7] JR Wallet et al. Omni protocol specification. URL: https://github.com/OmniLayer/spec.

[8] Ethereum roadmap: Proposer-builder separation. URL: https://ethereum.org/en/roadmap/pbs/.

[9] Evmos. Evmos: Evm extensions. URL: https://docs.evmos.org/develop/smart-contracts/evm-extensions.

[10] Flashbots. Mev-boost in a nutshell. URL: https://boost.flashbots.net/.

[11] Interchain. Understanding the basics of a proof-of-stake security model. URL: https://blog.cosmos.network/understanding-the-basics-of-a-proof-of-stake-security-model-de3b3e160710.

[12] Marius Kjærstad. Maximal extractable value (mev). URL: https://ethereum.org/en/developers/docs/mev/.

[13] Robin Linus. Bitvm. URL: https://bitvm.org/.

[14] minimalsm. Ethereum virtual machine. URL: https://ethereum.org/en/developers/docs/evm/.

[15] Murch. What are the sizes of single sig and 2-of-3 multisig taproot inputs? URL: https://bitcoin.stackexchange.com/questions/96017/what-are-the-sizes-of-single-sig-and-2-of-3-multisig-taproot-inputs.

[16] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. May 2009. URL: http://www.bitcoin.org/bitcoin.pdf.

[17] Gholamreza Ramezan, Manvir Schneider, and Mel McCann. A survey on coin selection algorithms in utxo-based blockchains, 2023. URL: https://arxiv.org/abs/2311.01113, arXiv:2311.01113.

[18] BitMEX Research. The $\mathrm{op}_r eturnwarsof2014–dappsvsbitcointransactions.URL$: .

Casey Rodarmor. Inscribing mainnet. URL: https://rodarmor.com/blog/inscribing-mainnet/.

Casey Rodarmor. Ordinal theory handbook. URL: https://docs.ordinals.com/.

smlXL. Precompiled contracts. URL: https://www.evm.codes/precompiled.

N. Wuille, P. Jonas and A. Towns. Taproot. URL: https://bitcoinops.org/en/topics/taproot/.

6 Appendices

6.1 Appendix A: List of Precompiles

6.1.1 Query Precompiles

1. `getRawTransaction(bytes32 txid): Transaction`

   Returns full transaction information about the specified txid, including inputs and outputs.

2. `getTxOut(bytes32 txid, uint256 n): Unspent`

   Returns information about the nth output of the specified txid.

3. `analyzePsbt(string calldata psbt): Transaction`

   Decodes base64-encoded PSBT into a `Transaction` struct. Can be used to verify that pre-signed PSBTs match settlement instructions defined by the calling smart contract.

4. `isInputQueued(bytes32 txid, uint256 n): boolean`

   Returns whether a given unspent is already part of a pending Bitcoin transaction tracked by the Inspector. Unspents that return `true` should not be included in newly-constructed transactions.

5. `joinPsbtSigs(Transaction[] memory psbts): Transaction`

   Joins multiple PSBTs with pre-existing signatures that sign the same transaction into a single Bitcoin transaction with all signatures applied.

6. `combinePsbts(Transaction[] memory psbts): Transaction`

   Joins multiple PSBTs which are independent of each other into a single Bitcoin transaction.

7. `inscriptionOwnerOf(uint256 inscriptionId): address`

   Checks the owner of a specified inscription, specified by ordinal ID. While on Bitcoin, ownership is represented by a Bitcoin address, this function returns the equivalent Ethereum address.

8. `runeBalanceOf(uint256 runeId, address owner): uint256`

   Checks the balance of a specified rune for a specified wallet. While the function parameter is an Ethereum address, the balance is checked for the equivalent Bitcoin address.

9. `createMultisig(address[] memory signers, uint256 m): (address, bytes)`

   Creates a new m-of-n multisig address with the specified addresses as signers. While the function parameter is a list of Ethereum addresses, the equivalent Bitcoin addresses are designated as signers. Returns the newly-created address and redeem script.

6.1.2 Write Precompiles

1. `sendBTC(uint256 sats, address to): void`

Send the specified amount of satoshis to the specific address. The network will perform coin selection and ordinal sats will not be used as inputs. The transaction will fail if the addresses signable by the smart contract do not have sufficient balance.

2. `sendOrdinal(uint256 inscriptionId, address to): void`

   Send the specified ordinal (a single-satoshi UTXO) to the specified address. The transaction will fail if the owner of the specified ordinal is not an address associated with the smart contract.

3. `sendRune(uint256 runeId, uint256 amount, address to): void`

   Send the specified amount of a given rune to the specified address. The transaction will fail if the addresses signable by the smart contract do not have sufficient balance of the specified rune.

4. `sendTransaction(bytes tx): void`

   Broadcast a transaction to the Bitcoin network. This precompile does not trigger any network signing; the transaction must be fully pre-signed. The transaction will fail if any inputs are unsigned.

5. `signAndSendPsbt(bytes psbt): void`

   Trigger network signing of a partially-signed PSBT and broadcast it to the Bitcoin network. This should be used for most smart contract flows which require signing from multiple counterparties (see Appendix B and C below). The transaction will fail if any unsigned inputs require signatures from keys not controlled by the network.

6.2 Appendix B: A Bitcoin-Native AMM (Use Case 1)

A Bitcoin-native AMM requires coordination between an unbounded number of counterparties and a single liquidity pool, where tokens for the specified market are held. The example below describes a pool where the one asset is BTC and the other asset is a rune. Rune-to-rune AMMs are similarly possible with slight modifications to transaction flow.

6.2.1 Deployment

- Segregate a single network key to hold assets on both sides of the LP.
- Etch a new Rune to represent the LP token of the pool.
- Deploy a Uniswap-style smart contract to Sova, with modified settlement functions as described below, with contract references to the rune ID of each asset (or `0xBBB` for Bitcoin), and a contract reference to the LP token rune ID.

6.2.2 Adding Liquidity

1. User constructs a PSBT with 3 outputs. The first 2 outputs should be sends of correlated amounts of each pool asset to the LP address, with the last output being the sending of LP tokens to the user.

   The user's wallet software should source inputs for each output and prompt the user to sign the inputs sourcing the LP pool assets, leaving inputs for the LP token unsigned (to be signed by the network). Amounts to be signed can be computed by reading state from the smart contract.

   Open question: how to prevent the transaction from failing due to slippage?

2. User calls `addLiquidity(bytes psbt)`. In the smart contract's execution logic, the contract must verify that all user inputs are signed, and that the input amounts and LP token output amount in the PSBT match the reserve ratios defined by the contract.

3. Smart contract updates contract state for LP token balances, reserve ratios, and the total liquidity of the pool. The Inspector locks these storage slots pending Bitcoin confirmation.

4. Smart contract calls `signAndSendTransaction` to send the user's PSBT to the signing service. Validators coordinate to sign and broadcast the transaction.

   Once the transaction receives sufficient Bitcoin confirmations, the Inspector unlocks the storage and the state change finalizes. The LP pool will receive the correlated token amounts and the user will receive the LP token, which acts as a redemption receipt for their funds (see "Removing Liquidity").

6.2.3 Trading

1. User constructs a PSBT with 2 outputs. The first output should be the token the user wishes to receive in the trade, with the user as the output address. The second output should be the token the user will provide, with the LP pool as the receiving address.

   This must correlate to the two assets of the LP pool. The amounts must match the current price of the pool as defined by the reserve ratio of the pool, which can be queried from the smart contract.

2. User calls `swap(bytes psbt, uint256 minAmountOut)`. In the smart contract's execution logic, the contract must verify that all inputs are signed, and that the input amounts and LP token output amount in the PSBT match the reserve ratios defined by the contract.

If pricing has changed since the user constructed the original PSBT, the smart contract can replace the PSBT's output for the output token with an updated amount. If the pricing has changed and the updated amount is less than `minAmountOut`, the transaction should fail.

3. The smart contract should update internal contract state for reserve ratios and pricing. The Inspector locks these storage slots pending Bitcoin confirmation.

4. Smart contract calls `signAndSendTransaction` to send the user's PSBT to the signing service. Once the transaction confirms with sufficient Bitcoin confirmations and the Inspector unlocks state, the LP pool will receive the token swapped from and the user will receive the token swapped for.

6.2.4 Removing Liquidity

1. User constructs a PSBT with 3 outputs. The first 2 outputs should be sends of correlated amounts of each pool asset to the user's address, with the last output being the sending of LP tokens to the LP address.

   The user's wallet software should source inputs for each output and prompt the user to sign the inputs sourcing the LP token, leaving inputs for the individual assets unsigned (to be signed by the network). Amounts to be signed can be computed by reading state from the smart contract.

2. User calls `removeLiquidity(bytes psbt)`. In the smart contract's execution logic, the contract must verify that all user inputs are signed, and that the input amounts and LP token output amount in the PSBT match the reserve ratios defined by the contract.

3. Smart contract updates contract state for LP token balances, reserve ratios, and the total liquidity of the pool. The Inspector locks these storage slots pending Bitcoin confirmation.

4. Smart contract calls `signAndSendTransaction` to send the user's PSBT to the signing service. Once the transaction confirms with sufficient Bitcoin confirmations and the Inspector unlocks state, the LP pool will receive the LP token and the user will receive the correlated token amounts.

6.3 Appendix C: Bitcoin-Native Peer-to-Peer Lending (Use Case 2)

A peer-to-peer lending protocol requires coordination between two counterparties (the borrower and the lender), as well as a collateral escrow mechanism controlled by the protocol itself. The example below describes a lending protocol for ordinals, but the same architecture can be used for any peer-to-peer lending arrangement regardless of collateral asset and funding asset. The terms implied are fixed-term loans with a pre-defined stable interest rate.

6.3.1 Deployment

- Segregate a derivation path under the smart contract's derivation path to be used for escrow addresses generated by the smart contract.
- Deploy a P2P lending protocol smart contract to Sova, such as that built by Arcade.xyz. Like in the AMM use case, settlement functions use Bitcoin precompiles instead of native EVM token transfer functionality.

  Note: Modified settlement protocols can also support using BTC-native as collateral with EVM-native funding currencies or vice versa.

6.3.2 Origination

The origination steps initializes a loan based on terms agreed upon by both counterparties. Collateral is set to an escrow address, with release of collateral conditioned on smart contract logic (see "Repayment" and "Default" steps).

1. Loan origination requires coordination between both counterparties. At origination time, one counterparty will be the **signer** and one the **originator**. The **originator** will initiate the origination transaction, and must provide a signature from the **signer**. Either of the borrowing or lending counterparty may play either of the signer or originator roles.

   The signer will sign an EIP712 signature to the loan terms as well as construct a PSBT with 2 outputs. The first output will send the funding currency from the lender to the borrower, and the second output will send the collateral from the borrower to escrow.

   To avoid race conditions and possible collisions, the escrow address must be pre-generated and marked as used by the smart contract. The signer should sign whatever inputs correspond to their responsibilities in settlement. Amounts to be specified and signed in the PSBT can be derived from the loan terms.

2. The originator should co-sign the origination PSBT, such that the PSBT is fully signed for submission to the network. Then, the originator calls `initializeLoan(struct LoanTerms, bytes signature, bytes psbt)`.

   The smart contract must verify that the loan terms match the terms signed in the EIP712 signature. It must also verify that the outputs in the PSBT match the loan terms, and that the signer of the signature has also signed the relevant inputs in the PSBT.

   Finally, the smart contract must verify that the escrow address specified in the PSBT's outputs is 2-of-3 multisignature address, with the counterparties controlling 2 keys and the last key being the pre-generated escrow address controlled by the smart contract, as specified in Step 1.

3. The smart contract stores the loan state and associated terms in smart contract storage. The Inspector locks the loan state pending Bitcoin confirmation. Smart contract calls `sendTransaction` to send the counterparties' transaction to the signing service for broadcasting.

Once the transaction confirms with sufficient Bitcoin confirmations and the Inspector unlocks state, the borrower will receive the loan funds and the escrow address will receive the loan collateral.

6.3.3 Repayment

In the repayment steps, the borrower pays owed funds to the lender, and receives their collateral back from the escrow address.

1. The borrower should construct a PSBT with 2 outputs: the first output should send the loan principal and accrued interest to the lender, and the second input should send collateral from the escrow address to the borrower.

   The borrower should pre-sign both the inputs for repayment as well as provide a partial signature for the withdrawal from the escrow multisig.

2. The borrower calls `repayLoan(bytes psbt)`. The smart contract must verify that the loan is active and that the escrow wallet still holds the collateral (the borrower has not defaulted).

   It must also verify that the outputs in the PSBT match the owed principal and interest as calculated by the smart contract. If the repayment amount is not sufficient, the transaction should fail.

3. Given a valid repayment, the smart contract should update stored loan state. The Inspector locks the loan state pending Bitcoin confirmation. The smart contract calls `signAndSendTransaction` to send the borrower's PSBT to the signing service, with validators providing the co-signature on the withdrawal from the multisig.

   Once the transaction confirms with sufficient Bitcoin confirmations and the Inspector unlocks state, the borrower will receive their collateral and the lender will receive their owed principal and interest.

6.3.4 Default

Default can occur if the borrower has not repaid funds before the loan's due date. In the default flow, the lender coordinates with the smart contract to recover the collateral from escrow.

1. The lender should construct a PSBT with a single output: the sending of the collateral from the escrow wallet to the lender.

   The lender should partially-sign the multisignature input from the escrow wallet for the withdrawal of collateral.

2. The lender calls `claimLoan(bytes psbt)`. The smart contract must verify that the loan is active and that the escrow wallet still holds the collateral (the borrower has not repaid).

   It must also verify that the due date of the loan has passed. If the escrow no longer holds the collateral or the due date has not passed, the transaction should fail.

3. The smart contract should update stored loan state. The Inspector locks the loan state pending Bitcoin confirmation. The smart contract calls `signAndSendTransaction` to send the lender's PSBT to the signing service, with validators providing the co-signature on the withdrawal from the multisig.

   Once the transaction confirms with sufficient Bitcoin confirmations and the Inspector unlocks state, the lender will receive the collateral, and the borrower will have no further loan obligation.