VARA.ETH

Vara.eth Technical Documentation

Authors: Eugene Way

Version 0.2 | October 2025

Table of Contents

1. Context

Vara.eth is a high-performance application platform built directly into Ethereum's ecosystem, powered by the Gear Protocol execution engine. Validator-operated executors watch Router/Mirror contracts, execute uploaded WASM programs (Gear runtime), and, by validator consensus, finalize state transitions back to Ethereum. There are no Vara.eth blocks, no shared ledger — only per-program state roots mirrored on L1.

Each program operates as an independent actor with its own isolated state and message queue, enabling deterministic parallel execution. Programs are executed on-demand across a decentralized network of Executor nodes, with no shared memory or global storage between them—forming a "swarm of rollups" architecture that enables massive horizontal scalability.

The core UX goal is to enable real-time dApps with gas-abstracted execution: users pay only ETH for L1 message submission, while program execution costs are covered from the program's Executable Balance in wVARA. This architecture is built around the Reverse-Gas Model, shifting compute payment away from the sender and into the program's own runtime budget.

Additionally, Vara.eth provides a pre-confirmation mechanism that delivers instant feedback to users. Executors sign computation results immediately after execution using FROST threshold signatures, allowing applications to display outcomes in real-time via RPC — before transactions are finalized on Ethereum L1. This enables Web2-like responsiveness while preserving blockchain security guarantees.

The key differences vs rollups are: no single sequencer, no L2 blocks, no global L2 state. Ordering and delivery arise from validator consensus on per-program transitions. Horizontal scaling comes from independent program queues and parallel execution.

Links

- Vara.eth GitHub - GitHub - gear-tech/gear: Web3 Ultimate Execution Engine
- Vara.eth Whitepaper - Vara.eth Whitepaper
- Official Website - Gear Technologies

2. Actors

The Vara.eth architecture consists of several key actors, each with distinct responsibilities that ensure the network operates securely, efficiently, and decentrally.

Router (Ethereum L1)

Central on-chain contract coordinating code registration, program creation, and validator-verified state transitions.

Responsibilities:

- Stores the registry of approved WASM codes.
- Deploys Mirror contracts.
- Verifies signed state transitions.
- Maintains validator key lists and runtime version control. Emits system-level events to trigger off-chain processing.
- Handles rewards and slashing commitments.

Mirror (Ethereum L1)

On-chain representation of a single Gear program on Ethereum side.

Responsibilities:

- Holds the program's current `stateHash`.
- Accepts incoming messages.
- Allows top-ups of Executable Balance in wVARA.
- Emits user-facing events and value transfer notifications.
- Binds program logic to Router governance and validator consensus.

Executors (P2P)

Validator-operated nodes executing program logic and forming commitments for Router.

Responsibilities:

- Observe L1 events from Router/Mirror.
- Execute WASM programs deterministically in the Gear runtime.
- Maintain local program queues and state DB.

- Aggregate validator signatures for new state roots.
- Submit commitments to Router for on-chain verification.

Validators / Operators (via Symbiotic)

Stakers operating executors and participating in validator consensus.

Responsibilities:

- Provide and rotate validator signing keys.
- Maintain uptime and correct execution.
- Participate in validator set elections per era.
- Accept slashing penalties for misbehavior.
- Earn rewards for validated execution work.

Clients / SDK (Sails)

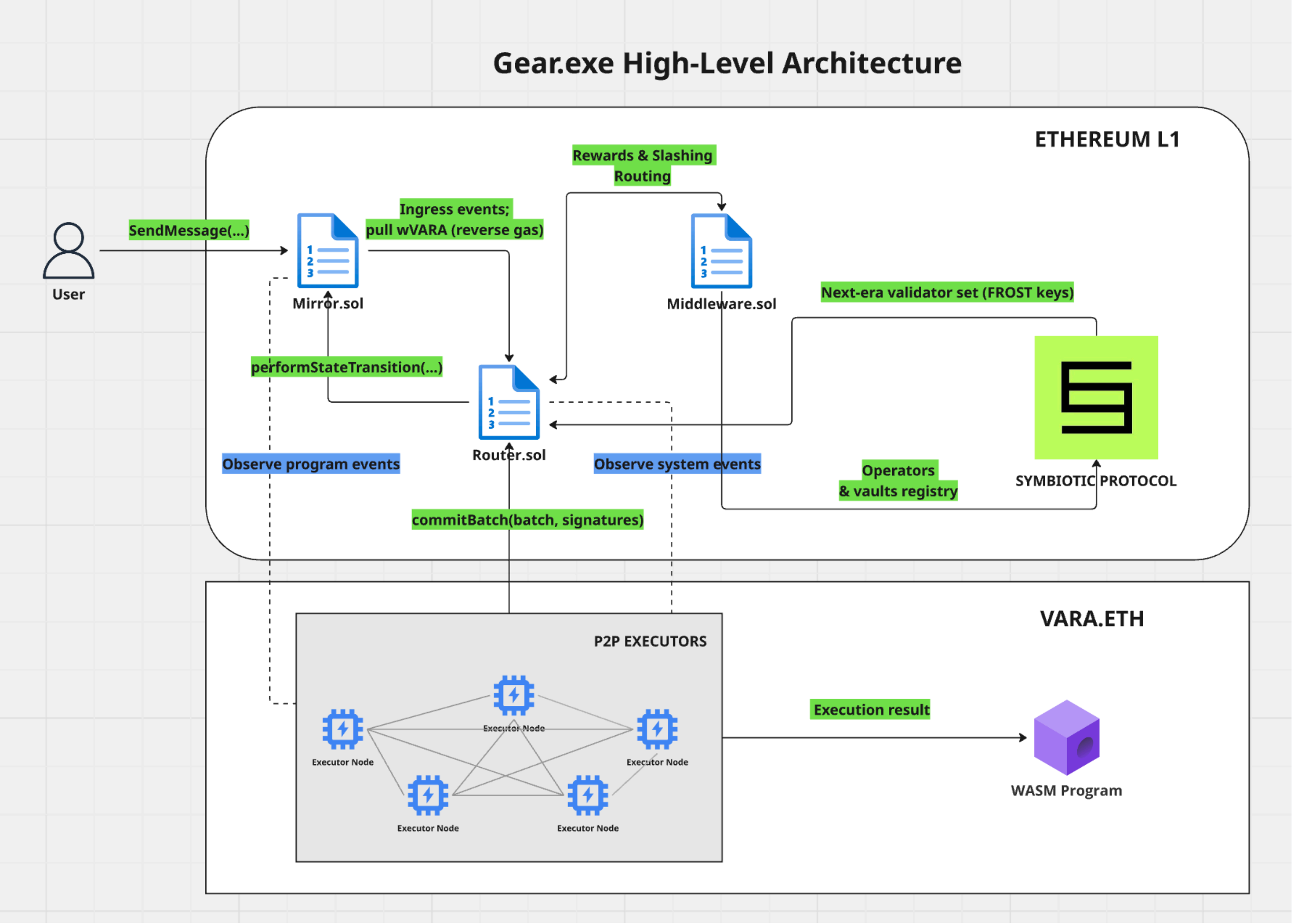
Developer-facing API layer for interacting with Gear programs without on-chain decoding.

Responsibilities:

- Encode and decode messages/events based on program IDL.
- Generate strongly typed client SDKs (sails-js, Rust).
- Integrate with `@gear-js/api` for connection and subscriptions.
- Provide a stable contract interface for dApp developers.

3. Components

Vara.eth is built on several interconnected components that work together to provide a seamless, scalable, and efficient computational layer for Ethereum.



Component overview showing Router, Mirror, Executors, Middleware, and WASM programs

3.0 Gear Program (Core, Vara.eth)

The **Gear Program** is the on-chain representation of a WASM smart contract running inside the Gear runtime. It encapsulates program logic, state, and ABI definitions, and is the execution target for messages routed from Ethereum via the Mirror contract.

The program is deployed and registered through the Router, either as a minimal instance (via `createProgram(...)`) or with an attached ABI interface (via `createProgramWithAbiInterface(...)`) for typed interaction. Once created, it is represented on Ethereum L1 by a Mirror contract that serves as its on-chain proxy.

Key Characteristics

- **Execution Target:** Receives messages from L1 through the Mirror and processes them inside Gear's WASM runtime.
- **State Management:** Maintains its own isolated storage state (`stateHash` represents this state on L1).
- **Interface Contract:** Optionally exposes a strongly typed ABI (Sails interface) for client encoding/decoding.
- **Deterministic Addressing:** Program `ActorId` and L1 Mirror address are derived from `(codeId, salt)` combination.

3.1 Router (Core, Ethereum L1)

The **Router** is the on-chain coordination hub of Vara.eth. It anchors all off-chain execution back to Ethereum by finalizing batches of work with validator signatures. Whenever executors replay program queues and produce transitions, they eventually come back to the Router to have those results recognized on L1. In that sense, the Router is the single entry point where Vara.eth state becomes Ethereum state.

Source: `Router.sol`, `iRouter.sol`

Core Functions

At a high level, the Router takes care of four things:

- **Validates code:** Developers submit a WASM blob (EIP-4844) and call `requestCodeValidation(codeId)`. Executors verify it off-chain; a later batch carries the result.
- **Creates programs:** Once a `codeId` is validated, anyone can deploy a deterministic Mirror clone via `createProgram(...)` or `createProgramWithAbiInterface(...)`.
- **Finalizes execution:** `commitBatch(...)` atomically applies everything for a given head — program transitions, code results, rewards, and the next validator set — if and only if validator signatures reach the threshold and the batch correctly chains.
- **Rotates validators:** Within explicit windows, batches can commit the next era's validator addresses plus FROST key material. The Router stores that and activates it at the next era start.

commitBatch(...) Contents

The `commitBatch(...)` call is the single settlement entry point. Each batch is a structured package containing program state transitions, code validation results, reward distribution instructions, and (when applicable) a new validator set. The Router verifies signature thresholds, hash chaining, and timing constraints. If any check fails, the entire batch reverts atomically.

After all four sections are assembled, the Router computes a single `batchHash` that covers their sub-hashes along with the previous batch hash and the current head/timestamp. Validator signature verification is the final step: the Router checks that enough signatures over this `batchHash` meet the configured threshold against the active FROST key. If the threshold is not met, or if any section is malformed or out of window, the entire batch reverts atomically.

Ordering & Safety

The Router enforces strict rules when finalizing a batch. Each `batchHash` must chain to the previously committed one, ensuring there are no gaps or forks. Timestamps must progress monotonically and never fall behind the last committed batch. Validator signatures must meet the configured threshold and verify against the aggregated FROST key of the current era. When included, the validator set update must strictly target `currentEra + 1` and appear within the election window. Finally, every `StateTransition` must belong to its designated Mirror contract, preventing cross-program execution. Any violation of these conditions causes the entire batch to revert atomically.

From a developer's perspective, the Router exposes only a few public calls — `requestCodeValidation(...)`, `createProgram(...)`, and `commitBatch(...)`. Everything else is internal to its settlement flow.

3.2 Mirror (Core, Ethereum L1)

The **Mirror** is the L1 twin of a Gear program. It queues user requests, applies state transitions authorized by the Router, and emits user-level events for applications and indexers. There are two deployments: the main Mirror implementation, and `MirrorProxy` — a minimal proxy that always points to the latest Mirror version. MirrorProxy ensures consistency and provides an ABI-friendly interface for developer tools and explorers.

Source: `Mirror.sol`, `IMirror.sol`, `MirrorProxy.sol`

Core Functions

At a high level, the Mirror takes care of four things:

- **Queues user inputs:** Users or contracts can send messages, replies, value claims, or balance top-ups. The Mirror records the request, transfers any attached wVARA to the Router, and signals executors to process it.
- **Top-up execution balance:** Through the `executableBalanceTopUp(...)` call, users can add wVARA to a program's balance. This serves as the entry point for the reverse-gas model, ensuring that execution costs are covered from the program's funds.
- **Applies transitions:** Only the Router can call `performStateTransition(...)`. The Mirror checks ownership, executes outgoing messages or replies, transfers value claims, and updates the program's state hash.
- **Manages lifecycle:** On deployment, the Router initializes the Mirror. Until the first message is sent, only the initializer may interact. Once a program exits, queueing is disabled and remaining balance can be passed to an inheritor.
- **Emits events:** The Mirror re-emits program-level outputs and forwards custom logs, so applications can subscribe directly on Ethereum.

performStateTransition Internals

When the Router finalizes a batch, it invokes each affected program's Mirror with a `StateTransition`. The Mirror first verifies that the transition truly belongs to this program. It then delivers any outgoing messages or replies to external accounts or contracts, transfers wVARA for pending claims, and updates the stored state hash to reflect the new program state. If the program has exited, the Mirror also records the designated inheritor and transfers any remaining locked balance. Finally, the Mirror returns a transition hash back to the Router, which incorporates it into the overall batch hash. The entire process is atomic: if any step fails, the Router reverts the whole batch, ensuring that state transitions across all programs remain consistent.

How Vara.eth Avoids a Decoder Contract

From a developer's perspective, interaction with a Gear program is intentionally designed to feel native to Ethereum users. Thanks to the MirrorProxy design, a Gear program appears on Ethereum as if it were a standard smart contract. When an ABI is attached, the proxy exposes not only the utility methods like `sendMessage`, `sendReply`, `claimValue`, or `executableBalanceTopUp`, but also the program's custom functions defined in its interface. These show up directly in explorers such as Etherscan under "Write As Proxy," so developers and users can call them just like any other Ethereum contract.

If no ABI is attached, programs can still be accessed through the raw entrypoint `sendMessage(bytes)`, where SCALE-encoded payloads are sent directly. In either case, event decoding is handled externally by the Sails framework based on the program's IDL, so no on-chain decoder contract is required. This avoids additional gas overhead for ABI translation, while still allowing applications and wallets to display program events in a familiar, human-readable form.

3.3 Executors (Off-Chain)

Executors are validator-operated nodes that monitor Router and Mirror events, run the Gear WASM runtime off-chain, process program queues deterministically, and co-sign per-block batches that the Router finalizes on L1. There is no centralized sequencer; ordering emerges from validator consensus on each batch.

High-Level Operation

Each program has its own message queue, its own state, and its own lifecycle. There is no global queue and no sequencer deciding a single order for everyone. Instead, a set of validators (running executors) watch Ethereum, replay inputs off-chain, and then collectively attest to "what happened to this block" in a batch that the Router finalizes on L1.

3.3.1 Observation Phase

Executors order all inputs deterministically: logs are sorted by `(blockNumber, txIndex, logIndex)` per program before execution.

Two classes of queues are maintained:

- **System queue (Router):** code validation requests, compute/timeline updates, rewards, validator elections, program balances.
- **Program queues (Mirror):** ingress messages, replies, claims, and top-ups for a specific Mirror address.

3.3.2 Execution Phase

Gear achieves parallel execution naturally. Each program has its own private queue; executors may run multiple programs concurrently, while strictly preserving intra-program order. This yields horizontal scalability with the number of active programs.

Compute policy applies a free threshold first, then meters execution time against the program's Executable Balance (wVARA). If the balance is insufficient, a message remains queued until the program is topped up.

A `StateTransition` includes:

- `newStateHash`
- `outgoing messages[]`
- `valueClaims[]`
- optional `exited/inheritor`
- `valueToReceive` (pre-transfer by Router)

Executors also capture event frames for Sails passthrough, which the Mirror re-emits as native Ethereum logs after safety checks.

3.3.3 Aggregation & Signing Phase

Executors package results into a batch containing:

- program state transitions (chain commitment)
- code validation outcomes
- reward distributions
- validator set commitments (when within the election window)

The batch is hashed (`batchHash`) together with the previous head. Validators co-sign this hash using the FROST threshold scheme; acceptance requires signatures from at least the configured percentage of the active validator set.

3.3.4 Settlement Phase

When submitted, the Router first pre-transfers any `valueToReceive` to the program, then calls `performStateTransition(...)`.

Election and timing guards apply:

- validator commitments must target `currentEra + 1` and be included inside the election window
- timestamps must be strictly monotonic relative to the latest committed batch

Executors follow best-head discipline for reorg handling: on reorg, per-program queues are rebuilt for the new head and transitions are recomputed. Any non-chaining batch is rejected on-chain.

3.4 Middleware (Symbiotic Layer)

Validators are the authority that co-sign Vara.eth batch commitments, while operators run the off-chain executors that observe Router and Mirror events and derive program state transitions. The validator set is sourced via the Symbiotic restaking stack and is activated on Ethereum L1 by the Router contract on an era schedule. Rewards for operators and stakers are routed on-chain through a dedicated Middleware contract.

Source: `Middleware.sol`, `IMiddleware.sol`

Roles

- **Operator:** An entity that runs executors. Operators register and opt in via Symbiotic and are tracked by the Middleware contract.
- **Validator:** An on-chain authority address that co-signs batch commitments. Each era's validator set (addresses plus FROST key material) comes from Symbiotic election output and is activated by the Router.
- **Vault (VA):** A Symbiotic staking vault that holds wVARA collateral, delegates stake to operators, routes staker rewards, and participates in slashing. Vaults must be registered in Middleware.

Operator Lifecycle

To be recognized by the protocol, an operator must:

- Register via Symbiotic and opt in to the subnetwork.
- Be explicitly enabled in Middleware.
- Be subject to disable or unregister calls with grace periods.

Once registered, an operator's stake is tracked through delegated vaults. Middleware uses stake snapshots to calculate operator weights and applies a deterministic process to select the top operators by stake, forming the validator set for the next era.

Vault Lifecycle

Vaults represent restaking collateral from nominators. A vault holds wVARA and delegates stake to one or more operators. In order to be accepted, it must:

- be registered in Middleware with parameters such as token type, epoch duration (at least twice the era length), rewards routing, veto/slasher setup, and resolvers.
- comply with enable, disable, and unregister rules that include delays to avoid destabilizing elections mid-era.

Registered vaults contribute their stake weights to operator elections and also receive routed rewards or slashing penalties.

Rewards Routing

During batch application, the Router approves wVARA to the Middleware, which then distributes rewards:

- Operator rewards are sent to operator reward contracts, proportional to performance and election outcome.
- Staker rewards are distributed across registered vaults and further split among their nominators.

This separation ensures that both node operators and capital providers are rewarded transparently.

Slashing

To preserve safety, Middleware enforces slashing through a three-step process:

- **Request:** A slash is proposed against one or more operators or vaults, with amounts and reasons.
- **Veto/Delay:** Each vault's VetoSlasher contract defines windows in which slashes can be vetoed or must wait before execution.
- **Execute:** Once the delay passes and no veto remains, the slash is finalized.

This ensures misbehaving operators or misconfigured vaults can be penalized without destabilizing validator elections.

Timing and Safety

Middleware validates its parameters against the Router's era schedule to avoid conflicts. For example, vault epochs must be at least twice the era length, and veto or slash delays must align with era boundaries. This guarantees that elections, reward distribution, and slashing do not interfere with the Router's validator rotation cadence.

3.5 Sails / SDK

Sails is the developer toolkit for Gear programs: it lets you author programs in Rust, define their interfaces (IDL), and generate client integrations for Ethereum. Sails complements the on-chain contracts (Router/Mirror) by providing type-safe encoding/decoding, ABI generation, and ready-to-use client bindings.

Sources: Sails GitHub, Docs wiki

What Sails Provides

- Rust macros, types, and helpers to define messages, replies, and events with a strongly typed interface (IDL).
- Solidity ABI interface contract generated from the Rust IDL (optional), attachable at program creation so explorers and wallets show human-readable methods.
- Client bindings for applications: TypeScript (sails-js / ethers.js), plus bindings for Rust and .NET, enabling type-safe calls and subscriptions without manual SCALE handling.

Encoding & Type Safety

Typed calls are compiled against the Sails IDL. At runtime, Sails encodes requests into SCALE and signs/dispatches them as messages to the program's Mirror. Replies are decoded back to the expected Rust/TypeScript types on the client side. Because decoding is handled off-chain by the generated bindings and the IDL, no on-chain decoder contract is required; this avoids extra gas for ABI translation while preserving a familiar UX.

Event Framing to Ethereum

Sails maps Rust events to Ethereum logs via a lightweight event frame. Programs emit frames (topics count, topics, ABI payload) to a reserved address `ETH_EVENT_ADDR`. The Mirror parses the frame and re-emits it as a native EVM log (`log1..log4`). Selector-collision guards ensure protocol safety. The result is a 1:1 mapping of program events to Ethereum logs that indexers and explorers can consume without custom decoders.

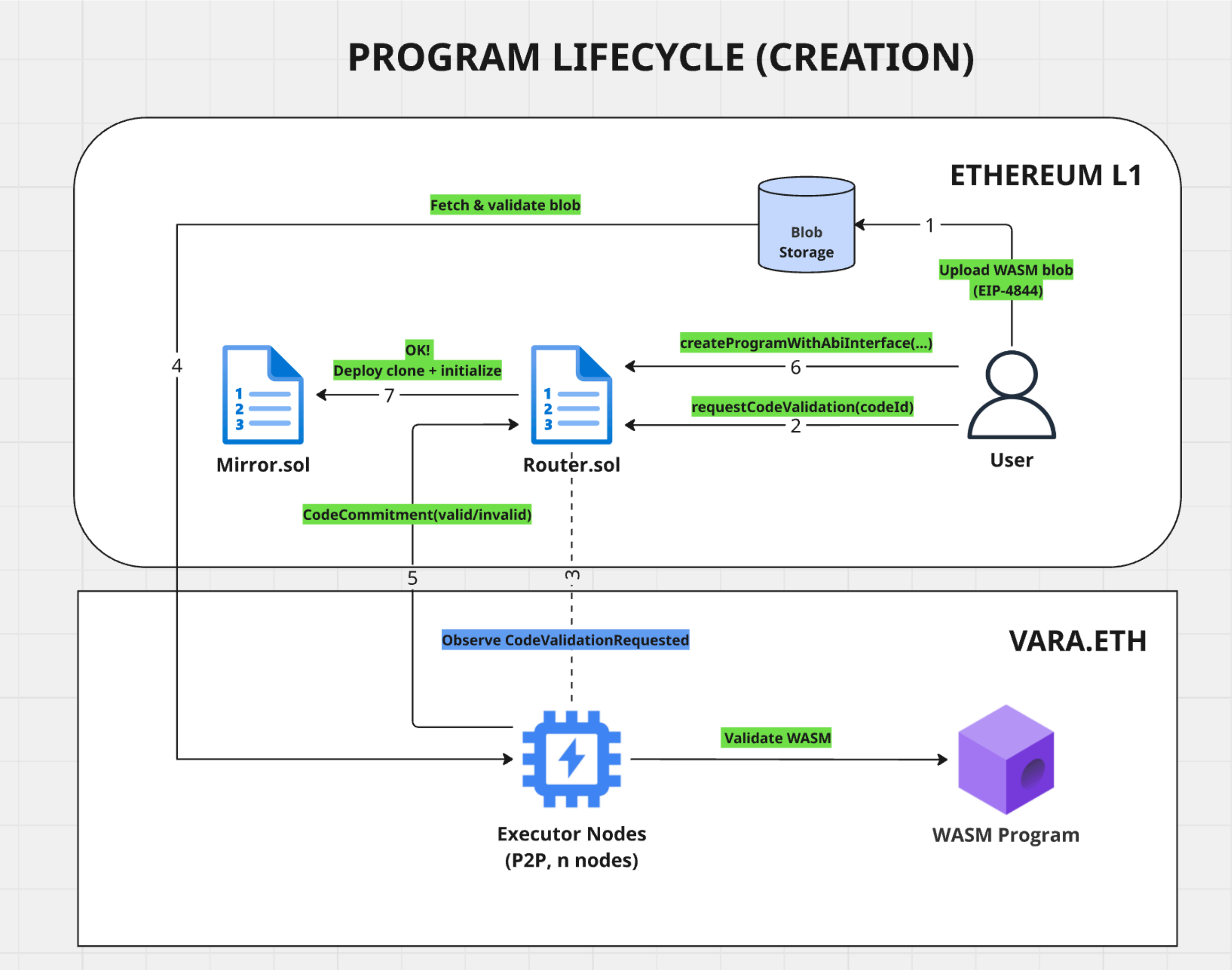
4. System Flows

4.1 Program Lifecycle Flow

A Gear program follows a clear sequence of steps: it starts as a WASM blob on Ethereum, becomes an active instance that processes messages, and finally produces state updates anchored on L1. Here's how it works end-to-end.

Involved Components

- **Router.sol** — entrypoint for validation, program creation, and batch settlement. [source]
- **Mirror.sol** — per-program proxy that manages queues, value pulls, and event emission. [source]
- **Executors** — off-chain workers that run WASM and prepare StateTransitions.
- **Middleware.sol** — reward and slashing router. [source]


PROGRAM LIFECYCLE (CREATION)
ETHEREUM L1
Fetch & validate blob
Blob Storage
1
Upload WASM blob (EIP-4844)
4
OK!
Deploy clone + initialize
7
createProgramWithAbiInterface(...)
6
requestCodeValidation(codeId)
2
Mirror.sol
Router.sol
User
CodeCommitment(valid/invalid)
5
3
VARA.ETH
Observe CodeValidationRequested
Validate WASM
Executor Nodes (P2P, n nodes)
WASM Program

Step-by-step flow: Upload blob → Validate code → Create program → Initialize → Deploy Mirror

Step 1: Uploading the Code

A developer starts by submitting a WASM blob inside an EIP-4844 transaction and calling `requestCodeValidation(codeId)` on the Router.

The Router records that this `codeId` is now pending validation and emits a `CodeValidationRequested` event. Executors notice this, fetch the blob, run it through validators, and prepare a code-commitment for inclusion in the next batch.

Sources: `Router.sol::requestCodeValidation`, `Router.sol::CodeValidationRequested`

Step 2: Code Validation Result

At batch time, executors include a `CodeCommitment` for that `codeId`.

The Router applies it via `commitCodes(...)`: if the code is valid, it gets marked `Validated` (and can now be used to create programs). If not, the request is dropped. Either way, an event `CodeGotValidated(codeId, valid)` goes on-chain.

Step 3: Creating a Program

Once a `codeId` is validated, anyone can spin up a new program by deploying its Mirror instance.

There are two call types:

- `createProgram(...)` — minimal clone, no ABI surface.
- `createProgramWithAbiInterface(...)` — same, but with a Solidity ABI interface attached for typed interaction.

The Router deploys the program instance deterministically (`salt = keccak256(codeId, salt)`), links the program's `actorId` to the `codeId`, emits `ProgramCreated`, and calls `Mirror.initialize(...)`.

With an ABI attached, explorers like Etherscan will show your custom methods directly in "Write as Proxy." Without it, interaction happens via the raw `sendMessage(bytes, value, callReply)` entrypoint.

Step 4: Initialization

Every program must be initialized. The initializer address sends the first message to the Mirror. The Mirror enqueues it and emits `MessageQueueingRequested`.

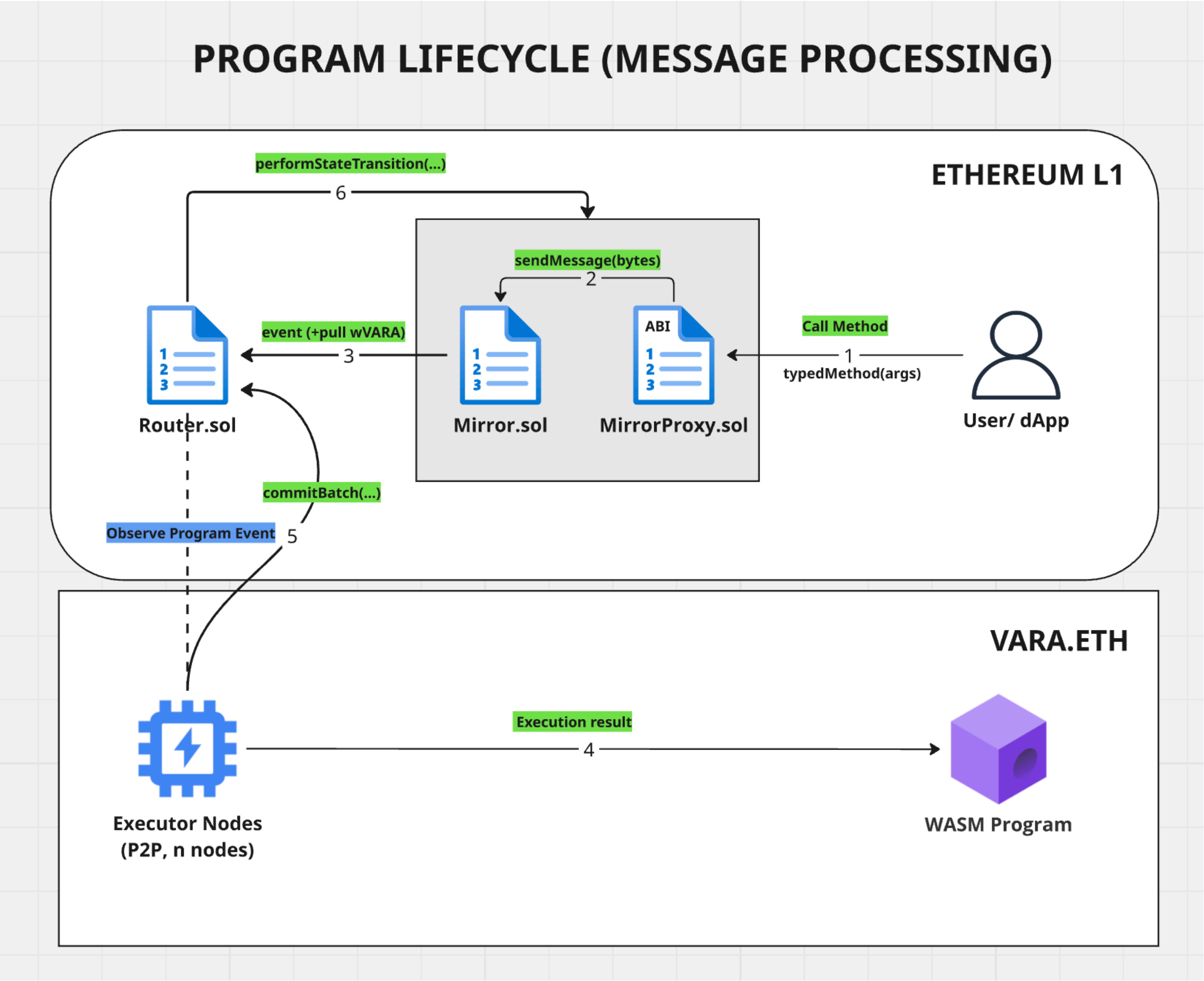
If the message carries value, the Mirror immediately transfers the attached wVARA from the sender to the Router. These funds are later credited back to the program during execution, ensuring that value transfers are accounted for deterministically.

Step 5: User Interaction

From here, anyone can interact with the program by sending messages:

- `sendMessage(payload, value, callReply)`
- `sendReply(repliedTo, payload, value)`
- `executableBalanceTopUp(value)`
- `claimValue(claimedId)`

Each call is just a simple transaction to the Mirror. Executors watch these events, classify them into the right per-program queues, and prepare them for execution in strict `(blockNumber, txIndex, logIndex)` order.



User sends message → Mirror queues → Executors execute → Router commits batch → Mirror applies state transition

Step 6: Off-Chain Execution

Executors then feed each program's queue into the Gear WASM runtime. Computation is charged deterministically: every program enjoys a small free threshold first, after which execution is paid from its executable balance. The runtime produces a `StateTransition`, which describes how the program has changed. It includes the new state hash, any outgoing messages or replies, and pending value claims. If the program is terminating, the transition also records the exit status and inheritor information.

In addition, it may specify `valueToReceive` — funds that must be transferred into the program before the transition can apply. When the program has emitted Sails events, these are framed so the Mirror can later re-emit them as native Ethereum logs.

Step 7: Settlement on Ethereum L1

Finally, the aggregated batch is submitted with `commitBatch(...)`, signed by at least the required validator threshold. The Router verifies chain continuity, timestamp monotonicity, and validator signatures against the current era's FROST key. Once verified, the entire batch is applied atomically.

In a successful batch, the Router processes all sections in order:

- Transfers any `valueToReceive` into the program before execution.
- Calls `Mirror.performStateTransition(...)` for each transition, delivering messages and replies, updating state, and emitting program events.
- Applies code validation results.
- Routes operator and staker rewards via the Middleware.
- If included, locks in the next-era validator set.

Afterward, the Router emits `BatchCommitted`, `HeadCommitted`, and `NextEraValidatorsCommitted`. At this point, the program's state is final on Ethereum.

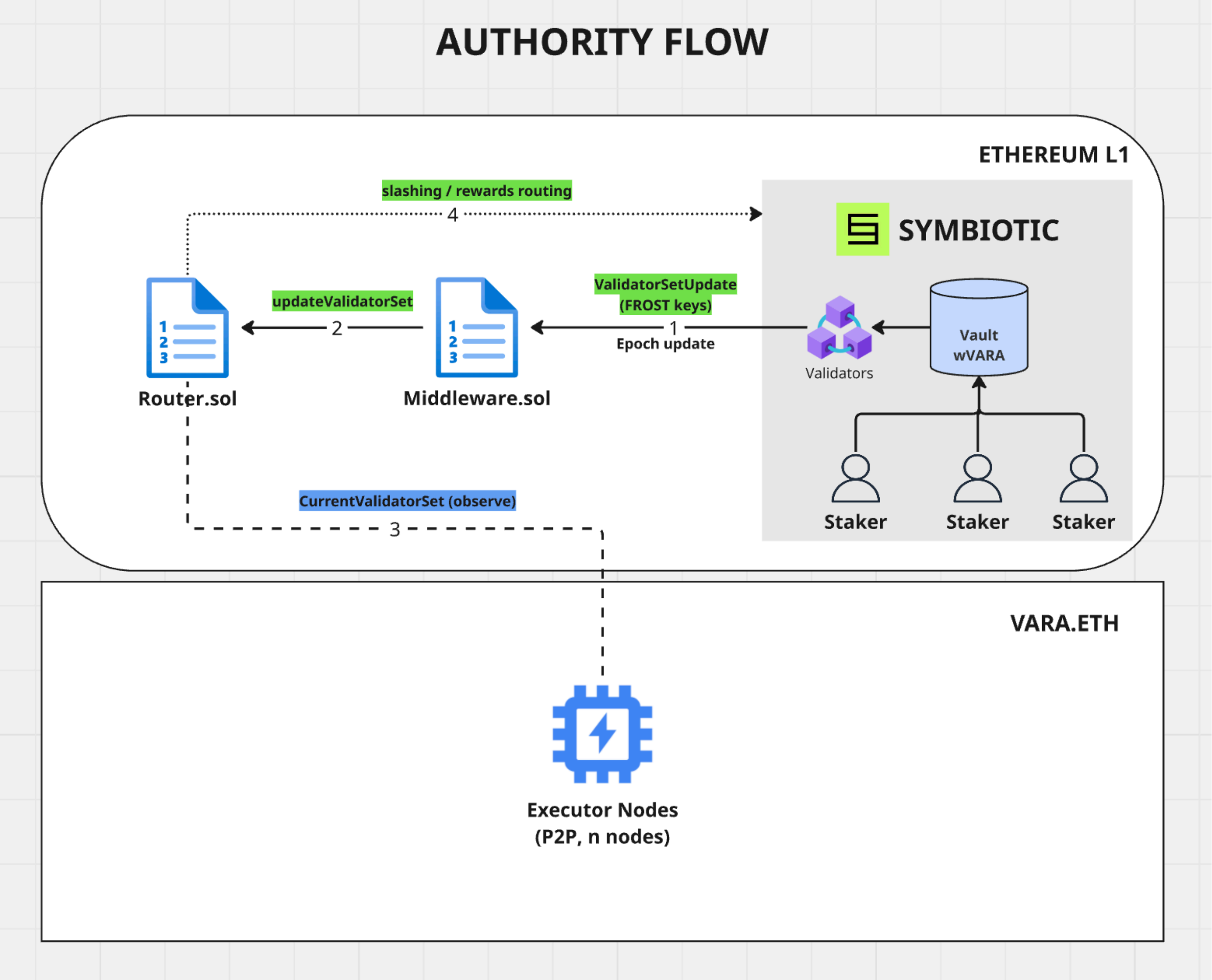
4.2 Authority Flow (Symbiotic → Router)

The Authority Flow explains how economic stake from Symbiotic restaking transforms into active validator authority on Ethereum L1.

It covers: how stakers fund vaults, how operators register, how elections produce the next validator set with cryptographic key material, how this commitment is activated by the Router, and how authority is exercised through signing, rewards, and slashing.

Involved Components

- **Symbiotic Vaults** — hold wVARA collateral and delegate stake to operators.
- **Middleware.sol** — registry for operators and vaults; provides stake views, election helpers, rewards/slashing routing. [source]
- **Router.sol** — accepts validator commitments inside batches, activates new sets, enforces signatures. [source]
- **FROST/DKG tooling**



Stakers → Vaults → Operators → Validators → Router (era activation) → Batch signing

Step 1: Stake & Opt-In

Stakers deposit wVARA into Symbiotic vaults, which aggregate collateral and delegate it to chosen operators. To participate, operators must opt in to the Vara network and register in Middleware using `Middleware.registerOperator(...)`. Vaults must also register with `Middleware.registerVault(...)`, specifying parameters such as token type, epoch length, and slashing configuration. Once registered, operators become eligible for validator elections and staker collateral is linked to operator performance.

Sources: `Middleware::registerOperator`, `Middleware::registerVault`

Step 2: Election

At election time, Middleware provides snapshots of operator stakes through functions such as `getActiveOperatorsStakeAt(ts)`. A deterministic selection process (`makeElectionAt(ts, maxValidators)`) chooses the top operators by stake. In parallel, executors perform distributed key generation (DKG) to produce the aggregated FROST public key and verifiable secret sharing (VSS) commitment for the next era. Together, these outputs define the validator set for the upcoming era.

Sources: `Middleware::makeElectionAt`, `Middleware::getActiveOperatorsStakeAt`, FROST/DKG tooling

Step 3: Validators Commitment

Executors package the new validator set into the next batch. The commitment includes:

- validator addresses
- the aggregated public key
- the VSS commitment
- and the target era index (`currentEra + 1`)

This commitment is hashed together with other batch sections (chain, code, rewards) and submitted to the Router for finalization.

Sources: `Router::_commitValidators`

Step 4: Activation

When `Router.commitBatch(...)` is called inside the election window, the Router verifies the proposal: the era index must equal `currentEra + 1`, the timing must fit the election window, and the validator list must be non-empty. If valid, the Router stores the set, locks in the FROST material, and emits `NextEraValidatorsCommitted(startTimestamp)`. This locks the new validator set, which activates at the beginning of the next era.

Sources: `Router::commitBatch`, `IRouter::NextEraValidatorsCommitted`

Step 5: Signing & Finality

For each batch in the current era, executors gather FROST signatures from the active validator set over the batch hash. The Router verifies chain continuity, monotonic timestamps, and ensures the signatures meet the threshold (`IRouter::validatorsThreshold`, `IRouter::signingThresholdPercentage`). On success, the Router applies all batch sections atomically. This is how validator authority is exercised: only the elected set can finalize batches, and only with threshold agreement.

Sources: `Router::commitBatch`, `IRouter::validatorsThreshold`, `IRouter::signingThresholdPercentage`

Step 6: Rewards Distribution

If the batch includes a rewards section, the Router approves wVARA to Middleware and calls:

- `Middleware.distributeOperatorRewards(...)` for operator payouts
- `Middleware.distributeStakerRewards(...)` for staker payouts via vaults

Middleware then forwards rewards to Symbiotic's distribution contracts. This ensures both node operators and stakers are rewarded transparently.

Sources: `Router::_commitRewards`, `Middleware::distributeOperatorRewards`, `Middleware::distributeStakerRewards`

Step 7: Slashing

If misbehavior is detected, Middleware can propose a slash using `requestSlash(...)`. Each vault's VetoSlasher contract defines veto and delay windows. If no veto is made, `executeSlash(...)` finalizes the penalty. Middleware ensures vault epochs and slashing windows align with Router eras to keep elections and rotations stable. This mechanism penalizes misbehaving operators or vaults without destabilizing the system.

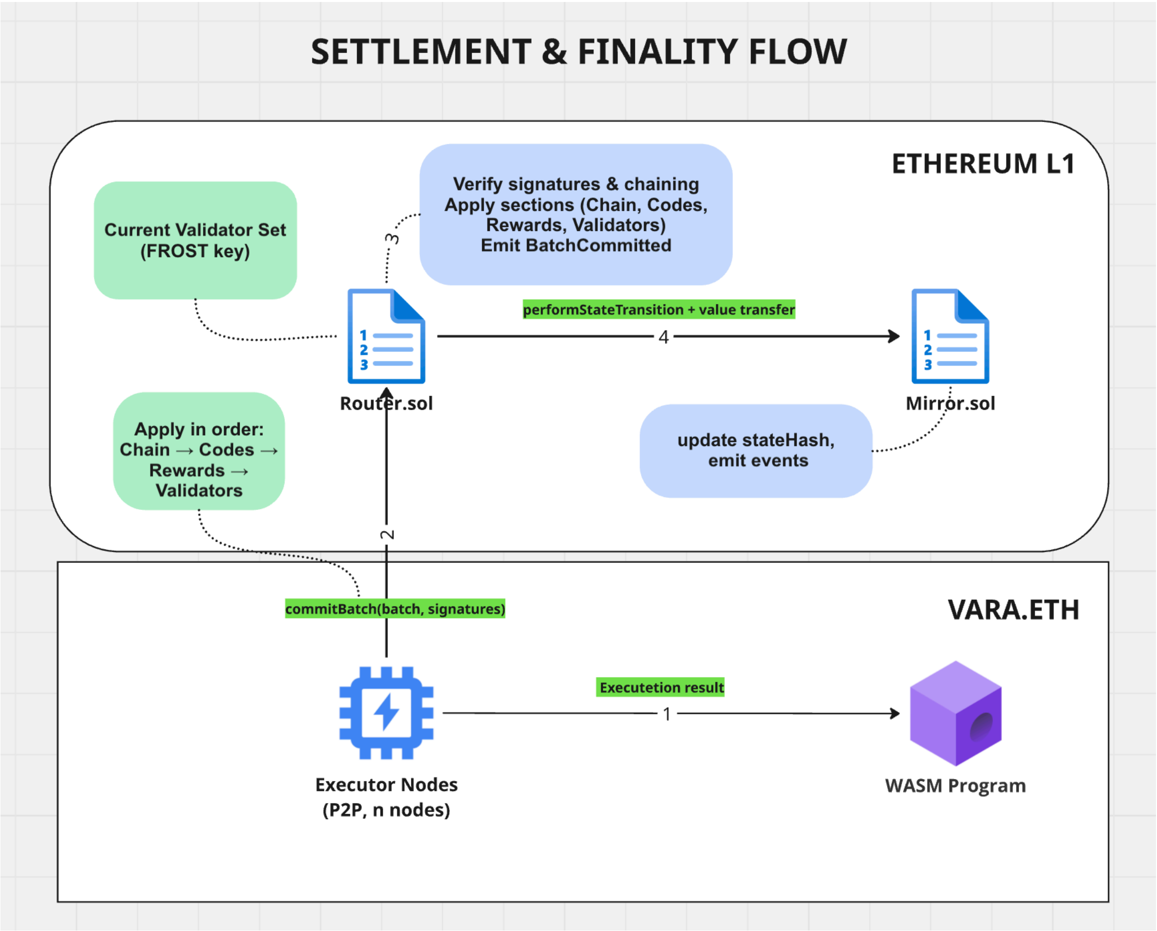
Sources: `Middleware::requestSlash`, `Middleware::executeSlash`, `Middleware::_validateStorage`

4.3 Settlement & Finality Flow

This flow explains how the validator set applies authority to make execution final on L1: what exactly is signed, how batch sections are applied, which contracts are involved, and which checks guarantee atomic "all-or-nothing" outcomes. If the Authority/Validators flow explains where authority comes from, this section shows how that authority is exercised per block.

Involved Contracts

- **Router (L1)** — central settlement entry point (`commitBatch`) [source]
- **Mirror (L1)** — per-program proxy that applies transitions (`performStateTransition`) [source]
- **Middleware (L1)** — reward/slashing routing [source]


SETTLEMENT & FINALITY FLOW
ETHEREUM L1
Current Validator Set (FROST key)
Verify signatures & chaining
Apply sections (Chain, Codes, Rewards, Validators)
Emit BatchCommitted
3
Router.sol
performStateTransition + value transfer
4
Mirror.sol
Apply in order: Chain → Codes → Rewards → Validators
update stateHash, emit events
2
commitBatch(batch, signatures)
VARA.ETH
Executetion result
1
Executor Nodes (P2P, n nodes)
WASM Program

Executors build batch → FROST signatures → Router verifies → Apply transitions → Emit finality events

Step 1: Batch Construction (off-chain)

Executors order all pending inputs (per-program queues + system requests), execute Gear WASM programs deterministically, and produce:

- `StateTransitions` (new stateHash, messages, value transfers, exits)
- `CodeCommitments` (validation results)
- `RewardsCommitment` (distribution root)
- `ValidatorsCommitment` (if inside election window)

The batch is hashed (`batchHash`) using Gear's primitives. Validators co-sign this hash with FROST keys.

Step 2: Batch Submission (on-chain entry)

Once signatures are aggregated, anyone can bring the batch on-chain by calling `Router.commitBatch(...)` . This is the single settlement entry point that hands the package to L1.

Source: `Router.sol::commitBatch`

Step 3: Signature Verification (current validator set)

Before any state changes, the Router verifies that the new batch chains to the previously finalized one, that timestamps move forward monotonically, and that signatures from the current validator set meet the configured threshold. Only a batch that satisfies these checks proceeds; otherwise it is rejected as a whole.

Sources: `IRouter::validatorsThreshold`, `IRouter::signingThresholdPercentage`

Step 4: Application

A verified batch is applied atomically. First, for each state transition the Router transfers any `valueToReceive` to the target program and invokes `Mirror.performStateTransition(...)`, which delivers messages and replies, updates the program's state hash, and emits program events.

Next, code-validation results are finalized; rewards are routed to Middleware and distributed to operators and stakers; and, if present and within the election window, the next validator set is stored for activation at the next era.

Sources: `Mirror.sol::performStateTransition`, `Router::_commitCodes`, `Router::_commitRewards`, `Middleware::distributeOperatorRewards`, `Middleware::distributeStakerRewards`, `Router::_commitValidators`

Step 5: Finality Signals

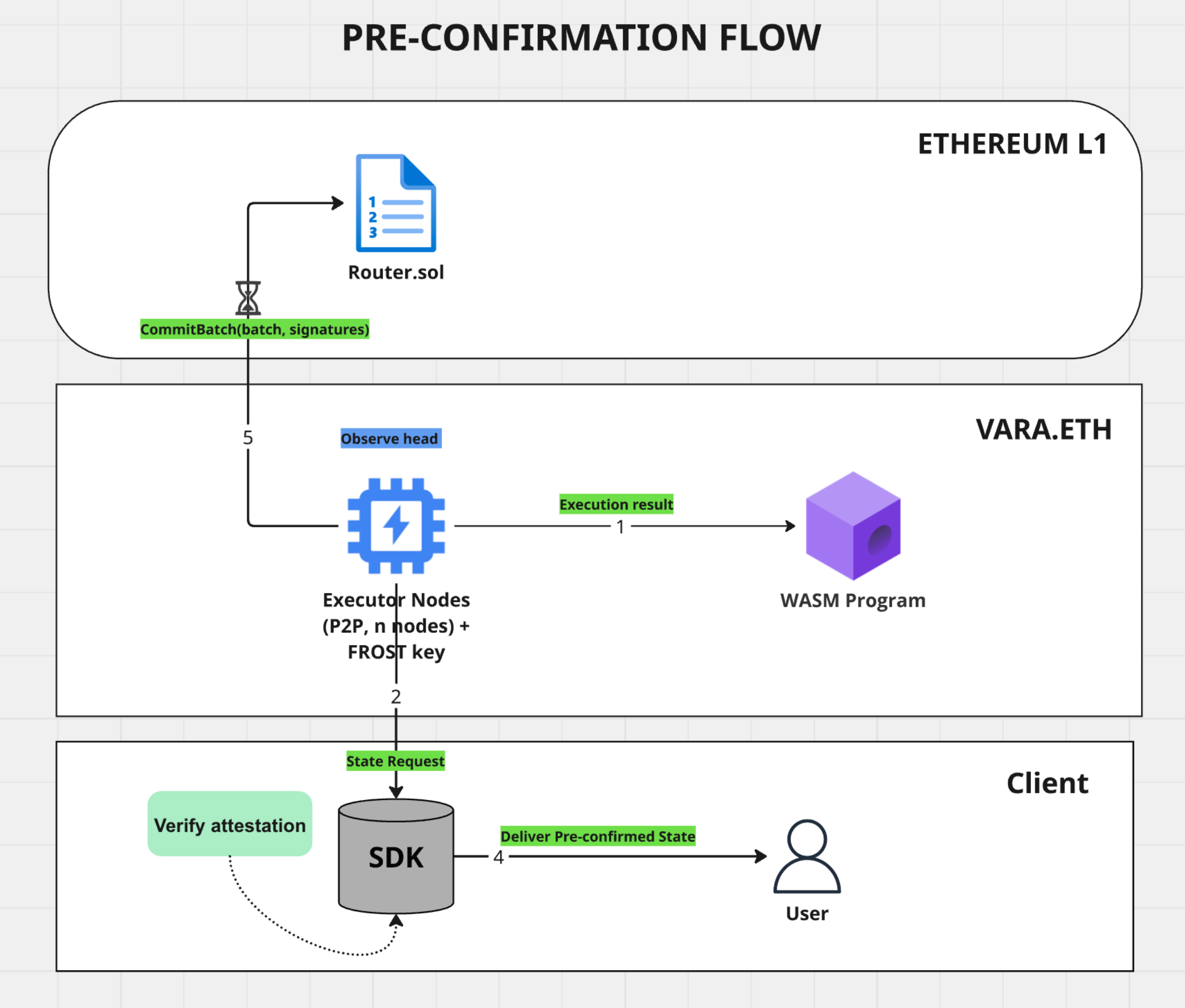
Once the batch is fully applied, the Router emits `BatchCommitted`, `HeadCommitted`, and `NextEraValidatorsCommitted`. These events mark the moment when off-chain execution has been anchored to Ethereum L1, making the results immutable and final.

4.4 Pre-confirmation Flow

Pre-Confirmation is a mechanism that allows users to see the result of program execution almost instantly, before the transaction is finalized on Ethereum L1. In practice, the user submits a transaction to a program's Mirror, and the SDK can immediately fetch the execution outcome via RPC.

The flow is the same as normal execution: executors track the best Ethereum L1 head, collect and order incoming inputs (Router and Mirror logs), and process them through the Gear WASM runtime. The resulting state transitions are signed with the current era's validator set. The SDK fetches this attestation over RPC, verifies it against the active era public key provided by the Router, and displays the program's results and events to the user.

Later, when the batch for that same head is submitted on-chain via `Router.commitBatch(...)`, the process continues in the usual way: the Router checks signatures and batch chaining, applies the state transitions, and anchors them as final Ethereum L1 state. In this way, Pre-Confirmation uses the same execution and signing mechanism as final settlement, but makes the results available to applications immediately via RPC, with finality coming later in the normal batch process.



User sends message → Executors execute immediately → Sign with FROST → SDK fetches & verifies → Later: batch commits on L1

5. Design Details

5.1 Ordering of Inputs

Ordering of inputs is the first discipline that Vara.eth enforces when consuming Ethereum L1 events. Every block on Ethereum may emit many Router and Mirror logs, and executors must turn this raw, unordered stream into a strict sequence that can be deterministically replayed by every validator. The rule is simple but absolute: all logs are sorted by `(blockNumber, txIndex, logIndex)` and then classified into system events (from Router) and program events (from Mirrors). This guarantees that no executor can reorder or skip an input — if two validators observe the same Ethereum head, they will construct the exact same sequence of inputs. This ordering discipline is what allows the system to remain leaderless, since determinism replaces the need for a sequencer.

System events such as `CodeValidation`, `BatchSettings`, and `ValidatorElection` always enter a dedicated system queue before any per-program messages. This ensures that governance changes, validator rotations, and execution policy updates are consistently applied prior to processing user-level messages in program queues.

5.1.1 FROST Threshold Signatures

Vara.eth uses FROST (Flexible Round-Optimized Schnorr Threshold Signatures) as the cryptographic foundation for validator consensus. FROST allows a group of validators to collectively sign batch commitments without requiring all validators to participate, while ensuring that no small minority can forge signatures on their own.

Key properties:

- **Threshold signing** — Only 67% of validators need to sign for a batch to be valid. This keeps the network live even when some validators are offline, while preventing minorities from approving invalid transitions.
- **Distributed key generation** — At each era start, validators jointly create a shared public key without any single party holding the full private key. Each validator gets a secret share for signing.
- **Non-interactive aggregation** — Validators create partial signatures independently. Any coordinator can collect and combine them into a single Schnorr signature that the Router verifies on-chain.
- **Efficient verification** — The Router stores the era's public key on-chain and verifies signatures with a single elliptic curve operation. No complex threshold logic needed on-chain.

FROST signatures are compact, gas-efficient, and compatible with future zero-knowledge proof systems. The threshold property ensures both security (no minority can sign invalid batches) and liveness (the network operates even if validators are offline). When eras rotate, new keys are generated in advance during the election window, ensuring seamless transitions without halting batch finalization.

5.2 Per-program Queues

Once inputs are ordered, Vara.eth separates them into per-program queues. Each program (represented on L1 by its Mirror contract) has its own isolated queue where messages, replies, claims, and top-ups are enqueued. Executors consume these queues independently, ensuring that the ordering discipline is preserved within each program but does not constrain other programs.

This isolation is the key design choice that gives Vara.eth its parallelism. Unlike rollups or L2s that enforce a single global sequence of transactions, Vara treats every program as its own self-contained unit of execution. As long as intra-program ordering is respected, executors are free to schedule different programs concurrently on separate threads or nodes. The result is horizontal scalability: more programs can run in parallel without creating contention or bottlenecks at the system level.

In other words, the Router ensures global input ordering, but Mirrors enforce per-program locality. Together, this model guarantees determinism and safety while enabling unbounded parallel execution.

5.3 Determinism

Determinism is the foundation of validator agreement in Vara.eth. Every executor that observes the same Ethereum head must produce exactly the same state transitions for a given program. To make this possible, the runtime enforces strict rules at every layer:

- **Ordered inputs** — Executors consume the same `(blockNumber, txIndex, logIndex)`-sorted stream of Router and Mirror events.
- **Deterministic WASM runtime** — Programs are executed inside a sandboxed Gear runtime, where system calls, gas metering, and message delivery are all precisely defined. No external randomness or nondeterministic syscalls are allowed.
- **State hashing** — Each transition is represented by a cryptographic hash (`stateHash`) that covers the program's full storage, outgoing messages, claims, and exit status. If two executors diverge, their transition hashes will not match and the Router will reject the batch.

Because of these guarantees, determinism replaces the need for a central sequencer. Validators do not need to "agree" on the order of execution in real time — the system ensures that if they see the same inputs, they will independently compute the same outputs. This property makes Vara.eth resilient to forks, network delays, or adversarial scheduling: finality depends only on threshold signatures, not on who executed first.

5.4 Parallelism

As previously noted, parallelism is foundational to Vara.eth's scalability. At the execution layer, each program operates within its own isolated queue, allowing multiple programs to run concurrently. Deterministic processing within each program ensures safety, while allowing horizontally scalable execution across threads.

From a performance standpoint, Vara.eth leverages multi-core hardware to accelerate computation: benchmarks—such as a Mandelbrot set simulation—demonstrated that Vara.eth can distribute workloads across multiple CPU threads, fully utilizing hardware parallelism for drastic performance gains. Gear is capable of scaling with available hardware; in many systems, up to 16 threads may be used for parallel execution, though actual limits depend on the specific hardware configuration.

Parallelism also extends to the architecture's validator and router layers, opening multiple scaling paradigms:

- **Single Router with unified validator set:** All validators serve all programs under one Router instance—simple and consistent.
- **Multiple independent Routers (clusters):** Each Router has its own validator set and serves its own program pool, enabling horizontal scaling across Vara clusters via the Ethereum L1 anchor.
- **Subgrouped validators within one Router:** Validators can be partitioned to handle subsets of programs, reducing intra-cluster synchronization and enabling finer-grained scaling within a single network.

These models can be combined, forming a scalable execution fabric: computation scales both within programs (via threads) and across Vara clusters, all while preserving Ethereum-level security and continuity.

5.5 Timing Windows

In Vara.eth, execution is not only about ordering inputs correctly but also about making sure they happen within the right time window. Timing windows define when an action is valid and how long it stays valid. This matters because Vara.eth connects two timelines:

- Ethereum L1, where blocks appear every ~12 seconds and may reorg
- Vara.eth eras, where validators execute and commit batches atomically

Each operation has its own timing rules. Validator elections must be submitted just before the next era begins — too early or too late and the Router rejects them. Batch commits are tied to Ethereum block timestamps, so if a commit arrives outside the allowed range, it cannot be finalized. Slashing challenges also have fixed veto/execute periods to make sure they only apply inside their defined era.

For executors this means they must always track both Ethereum block time and the Router's era index. For example, `Router.commitBatch(...)` will only accept a batch if the `eraIndex` matches the expected next era and the block timestamp falls inside the allowed window. Middleware applies similar checks for storage and slashing.

The effect is simple: if something comes outside its timing window, it is invalid by definition. This prevents replay of old signatures, blocks arriving out of order, or late commits slipping into the system. For clients and users, it also means predictability: once the window closes, previews can safely be treated as final.

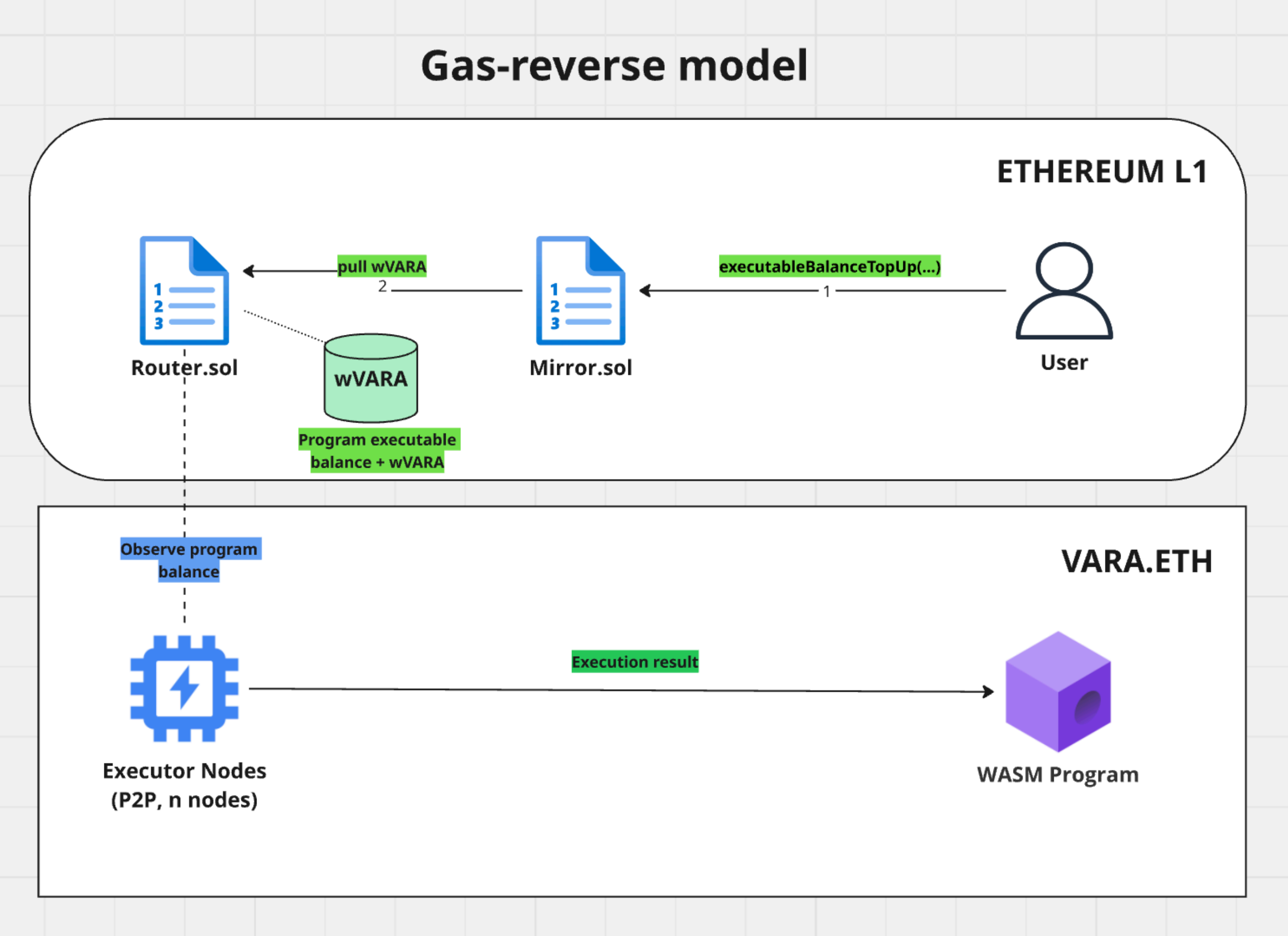
5.6 Reverse-Gas Model

In Vara.eth, transaction costs are flipped compared to the standard Ethereum gas model. On Ethereum, every caller pre-pays gas for execution, which makes even simple user interactions depend on unpredictable compute costs. Vara.eth introduces the Reverse-Gas Model, where users only pay Ethereum L1 gas in ETH to submit their message, while the actual program execution is funded from the program's own balance in wVARA.

Anyone can fund a program's execution by sending wVARA to its Mirror.sol via a top-up call via `executableBalanceTopUp(...)`.

Mirror immediately pulls the tokens into the Router during ingress. The Router acts as a central accountant: it tracks how much executable balance each program has and ensures that no computation is ever attempted without being pre-funded. In other words, the Router maintains a secure ledger of balances for all programs, and executors simply observe these events off-chain.

Executors then process program queues deterministically in the Gear WASM runtime. They know exactly how much balance is available to each program by observing ingress logs from Router, and they simulate execution accordingly. CPU time is metered with a predictable policy — every program gets a free compute threshold first, and beyond that compute is charged at a fixed `wvaraPerSecond` rate. If a program's balance is insufficient, its messages are not discarded; they simply remain queued until more funds arrive. Importantly, executors never "run on IOUs" because the Router has already pulled the wVARA upfront, guaranteeing solvency before any off-chain execution starts.



User sends message (pays ETH) → Mirror pulls wVARA to Router → Executors observe balance → Execute program → Consume executable balance → Distribute to validators

Final settlement on Ethereum L1. When executors submit a batch via `commitBatch(...)`, the Router enforces chaining and signatures and then applies the batch atomically. As part of settlement the Router debits the program's Executable Balance (on the Router) for the metered compute.

This is why Vara.eth calls the mechanism the Reverse-Gas Model. Instead of users being forced to pre-pay gas for every execution, programs pay for themselves from their pre-funded balances. Users enjoy a smooth experience — they only spend ETH for the L1 transaction — while programs retain control over how their compute is funded, whether through prepaid deposits, value attached to messages, or external top-ups. At the same time, executors are protected from unfunded execution because Router enforces strict balance discipline at the point of settlement.

The result is a model that combines better UX, predictable costs, and protocol-level safety. Users interact with Vara programs just like any Ethereum contract, but under the hood, balances are managed in reverse — ensuring execution is always prepaid and never dependent on the caller's willingness or ability to fund gas.

Why Mirror Does Not Maintain Execution Balance

In Vara.eth, program execution balances are never stored on the program's own Mirror contract. Instead, all wVARA for compute lives in the Router. This makes life easier for executors: they all look at one place, the Router's ledger, and always see the same balances. If Mirrors held their own balances, cross-program calls could end up in different orders on different nodes, breaking determinism. With a single accountant, everyone processes the same numbers in the same order.

It also keeps settlement safe. A batch on Ethereum L1 must either apply fully or not at all. If Mirrors moved funds before a batch was finalized, money could "leak" and then be rolled back inconsistently. By holding execution balances in the Router until finality, the system guarantees atomicity: nothing moves until the batch is proven valid. Only then, if the program is meant to receive value for its logic, does the Router transfer it to the Mirror. This way the Router handles compute funding, and the Mirror only ever holds tokens that are safe for the program to spend.

5.7 Reorgs & Batch Chaining

In Vara.eth, history must stay linear even though Ethereum itself can reorder blocks during short-range reorgs. Executors therefore treat Ethereum as a moving target: they always follow the node's current best head and rebuild their previews if that head changes.

Whenever a new Ethereum head arrives, executors collect and order Router/Mirror logs up to that point and replay programs deterministically. The result is assembled into a batch — a package of state transitions, code updates, rewards, and validator changes. Every batch carries the hash of the previous committed batch, so the Router can only accept it if it chains directly from the last finalized step.

This chaining rule makes the on-chain record strictly linear. Even if executors worked on a preview for a block that later got reorged out, they simply throw away that preview, recompute from the new head, and resubmit. From the Router's perspective, there is no fork: only a single linked chain of batches.

Finalization is all-or-nothing. If the batch's signatures, timing, or chaining checks fail, the commit reverts as a whole. If it passes, the Router commits the new hash, applies all state transitions atomically, and emits a fresh `HeadCommitted` event. No partial counters, no half-applied top-ups — either the entire batch lands, or nothing does.

Pre-confirmations fit naturally into this model. Executors can expose off-chain results to users right after local replay, giving applications instant feedback. But these results are always provisional: if Ethereum reorgs before the batch is finalized, the preview may be replaced by a recomputed one.

5.8 Safety Constraints (Summary)

This section serves as a brief recap of the previous design details. It highlights the core invariants that explain why Vara.eth can guarantee safe execution and final application of results without requiring a trusted sequencer or coordinator.

- **Deterministic programs** — All executors consume the same ordered input stream and run it inside a sandboxed WASM runtime. No nondeterminism, no external syscalls. If two executors diverge, their transition hashes will not match and the Router will reject the batch.
- **Isolated accounting** — Mirrors enforce ingress/egress balance discipline. Value is always credited before execution, charged deterministically during runtime, and applied atomically on settlement.

Executors never rely on IOUs.

- **Atomic batches** — The Router only accepts batches that chain correctly, carry threshold signatures from the current validator set, and pass timing checks. If any check fails, the entire batch reverts.
- **Timing discipline** — Elections, commits, and slashing all happen inside explicit windows, aligned with Ethereum blocks and Router eras. Anything outside these windows is invalid by definition.

Together, these constraints summarize why Vara.eth maintains safety even under reorgs, validator rotations, or adversarial inputs: determinism ensures agreement, accounting ensures solvency, and atomic batch rules ensure consistency of state.

5.9 ZK-Verification Mode

ZK-Verification Mode is a future security enhancement that uses zero-knowledge proofs to validate state transitions. While the current system relies on economic security—validators risk slashing if they sign incorrect batches—ZK mode provides cryptographic guarantees that execution was performed correctly, regardless of validator behavior.

Why ZK?

The FROST-based model assumes at least 67% of validators act honestly, enforced through economic stake. For most applications, this is sufficient. However, certain high-stakes use cases—financial settlements, regulatory compliance, critical infrastructure—require absolute cryptographic certainty rather than economic deterrence. Zero-knowledge proofs offer this: they mathematically prove correct execution, making it impossible for even a fully colluding validator set to finalize invalid state transitions.

How It Works

Executors process programs and generate state transitions as usual. Before a batch reaches Ethereum L1, a zero-knowledge proof is created demonstrating that all transitions were computed correctly. This proof is submitted with the batch to the Router, which verifies it on-chain. The proof is succinct—a single proof can cover thousands of program executions—keeping gas costs manageable through recursive aggregation.

Operating Modes

Hybrid Mode combines speed and security. Executors provide instant pre-confirmations using FROST signatures (sub-second latency), while zero-knowledge proofs verify correctness before L1 finalization. Users get immediate feedback; the blockchain gets cryptographic security.

Pure ZK Mode uses only zero-knowledge proofs, eliminating validator signatures entirely. Maximum security with no trust assumptions, but proof generation adds latency (30 seconds to minutes). Ideal for applications where absolute correctness is paramount.

For Developers

Programs work identically in both modes. The verification mechanism is transparent to program logic—whether validated by signatures or proofs, state transitions follow the same format. Developers choose their security model at deployment: Hybrid for performance-critical apps, Pure ZK for security-critical ones. The current FROST system remains the default for speed and efficiency, with ZK mode as an opt-in enhancement.

6. Economics

The economic model of Vara.eth is built around the wVARA token. It acts both as the fuel for program execution and as collateral for validators via Symbiotic. This section explains how wVARA is issued, how rewards are funded and split, how slashing ensures accountability, and how parameters are configured.

6.1 Token and Bridge Scope

The core asset of the system is **wVARA**, a wrapped version of VARA issued on Ethereum L1 through a two-way bridge with Vara Network. When users move VARA from Vara, the native tokens are locked and an equivalent amount of wVARA is minted on Ethereum; the reverse process burns wVARA and unlocks VARA back on Vara.

All system contracts — Router, Mirror, and Middleware — operate exclusively with wVARA. Each program on Ethereum maintains an **Executable Balance**, which is consumed to pay for execution under the reverse-gas model.

The same token also secures the validator set: stakers deposit wVARA into Symbiotic Vaults and delegate it to operators. This way, computation and security are tied to one unified asset, avoiding liquidity fragmentation.

6.2 Rewards Funding & Splits

Validator and staker rewards are funded from a dedicated wVARA rewards pool. At batch finalization, the Router triggers distribution by calling into Middleware, which splits rewards across operators and their delegating stakers.

Operators earn rewards directly in proportion to their activity and role in finalizing batches. Stakers earn their share via Vaults, proportional to the stake they delegated. The exact parameters — emission rate, operator vs. staker share, distribution cadence — are governed and can be updated via Router governance.

6.3 Slashing Policy

To enforce correct behavior, Vara.eth integrates slashing hooks. Misbehavior — such as invalid attestations or malicious execution — can trigger a `requestSlash` through Middleware. The corresponding operator's Vault receives the request, and the VetoSlasher module enforces windows for veto and execution.

If the challenge stands, an `executeSlash` call burns or confiscates a portion of the operator's and delegators' wVARA stake. This direct economic penalty ensures that operators remain accountable and ties network safety to financial risk.

6.4 Parameterization

Key system parameters are configurable and updated via Router governance. Critical parameters include: the validator signing threshold (`signingThresholdPercentage`), the era length (`eraDuration`), the base execution fee rate (`wvaraPerSecond`), and the free compute allowance (`freeThreshold`).

Middleware enforces additional guardrails on Vault registration: for example, veto and slashing windows must fit within Router-era boundaries, and era durations must align with Router cycles. These constraints prevent timing mismatches between economic and consensus layers.

Importantly, parameter changes only take effect at era boundaries, ensuring consistent state across executors and predictable economics for programs and users alike.

7. Glossary and Terms

Actor Model

A computational model where individual components, called actors, operate independently and communicate with each other through messages. This approach enables parallel processing and high scalability, which are integral to Vara.eth's architecture.

Archive Node

An Ethereum node that stores the complete history of the blockchain, including all past states and transactions. Unlike full nodes, which only maintain the current state and recent transaction data, archive nodes retain historical data that allows developers and applications to access detailed information about any block or state from the chain's entire history. Archive nodes are essential for tasks like querying historical balances, accessing older smart contract states, or retrieving blobs uploaded for off-chain processing, as utilized by Vara.eth.

Based Rollups

A type of Layer-2 scaling solution that relies directly on Layer-1 protocols for sequencing and data availability. Unlike traditional rollups that use dedicated infrastructure, based rollups integrate deeply with the underlying blockchain, leveraging its decentralization and security guarantees. This alignment with Layer-1 simplifies operations by removing the need for native tokens or separate trust assumptions. While based rollups benefit from Ethereum's censorship resistance and robust consensus, they inherit its limitations, such as slower transaction finality and shared scalability constraints. Additionally, transaction flexibility is often reduced because sequencing and execution must conform to Layer-1 rules.

Batch / BatchHash

An atomic package of state transitions, code updates, rewards, and validator changes. Every batch links to the previous `BatchHash`, ensuring strict linearity.

Blob

A large binary object stored on the Ethereum network as part of a transaction. In Vara.eth, Wasm code is uploaded as a blob, which resides outside Ethereum's main state but is accessible via archive nodes.

Clones / ClonesSmall (EIP-1167)

Minimal proxy contracts deployed deterministically (via salt). Vara.eth uses them to spawn Mirror contracts with low gas costs and predictable addresses.

dApp (Decentralized Application)

A software application that runs on a blockchain or decentralized network. dApps are powered by smart contracts and provide users with transparent and trustless interactions without relying on centralized servers.

DKG (Distributed Key Generation)

A protocol that lets a group of validators jointly generate a public key and private shares without any trusted dealer. In Vara.eth, DKG is run at each era to derive the new FROST key.

Era (Validator Era)

A fixed-time period during which a validator set remains active. At the end of each era, new validators and FROST keys are elected and committed.

Executable Balance

A per-program account (in wVARA) that covers compute costs in the Gear runtime. Execution halts if balance is insufficient until more wVARA is topped up.

Executor

A decentralized node within the Vara.eth network responsible for executing Wasm programs. Executors retrieve programs, perform computations, and generate signed results, ensuring the network's reliability and scalability.

Finality

The point at which a transaction or computational result is considered immutable and irreversible. On Ethereum, finality typically occurs after ~12.8 minutes, but Vara.eth enhances this by providing pre-confirmation mechanisms for near-instant feedback.

FROST (Flexible Round-Optimized Schnorr Threshold Signatures)

A threshold signature scheme that allows a validator group to jointly produce a single Schnorr signature. In Vara.eth it is used for co-signing batch commitments.

Gear Protocol

The foundational framework behind Vara.eth that supports the creation and execution of Wasm programs. It provides the tools and runtime environment necessary for decentralized computation.

IDL (Interface Definition Language)

A file that describes the structure and interface of a Wasm program. Developers use IDL files to define how their Gear programs interact with external systems or smart contracts.

Middleware

A specialized contract that links the Router Contract with the Symbiotic Protocol. It manages responsibilities such as key registration, updates, and validator/operator accountability, including slashing requests. Middleware ensures secure coordination between execution results and staking economics. [source]

Mirror Contract

A smart contract deployed on Ethereum to act as an interface for a Gear program. Mirror Contracts enable communication between the Ethereum blockchain and off-chain computations performed on Vara.eth. [source]

Mirror ABI Interface

The ABI (Application Binary Interface) exposed by Mirror Contracts on Ethereum. This interface allows Ethereum tools and dApps (e.g., MetaMask, Etherscan) to interact with Gear programs natively, without the need for additional encoding/decoding, by treating them as standard Ethereum smart contracts. Mirror ABI Interfaces act as a transparent proxy layer for Gear programs.

Operator

An entity responsible for running Executor nodes within the Vara.eth network. Operators are elected through the Symbiotic Protocol based on delegated stake and are accountable for reliable execution. They represent the economic and operational counterpart to Executors.

Optimistic Rollups

A type of Layer-2 scaling solution that processes transactions off-chain and periodically posts summarized data (state roots) back to the Ethereum mainnet. Optimistic Rollups operate under the assumption that transactions are valid ("optimistically") unless proven otherwise. To ensure security, they include a challenge period during which anyone can submit fraud proofs to contest invalid transactions. This mechanism provides scalability but introduces delays in transaction finality due to the need for a dispute resolution window.

Pre-confirmation

An off-chain, signed preview of a batch at a given Ethereum head. It gives clients off-chain results until `Router.commitBatch(...)` applies the batch on-chain.

Restaking / Symbiotic

Symbiotic is the restaking protocol that provides validator security for Vara.eth. Stakers deposit wVARA into Vaults, operators register via Middleware, and the Router activates elected validator sets on Ethereum L1.

Reverse-Gas Model

A model where program execution is funded by the program's own Executable Balance (wVARA), not by the caller. Users only pay L1 gas in ETH to submit a message; execution costs are charged against the program's balance.

Router Contract

The central smart contract in Vara.eth's architecture that coordinates interactions between Ethereum and the Vara.eth network. It handles program uploads, execution results, and state transitions. [source]

Shared Storage

A blockchain or Layer-2 design feature where all participating nodes or entities share access to a unified state, including data and smart contract storage. This approach ensures consistency and transparency across the network but can limit scalability due to bottlenecks in data retrieval and update operations. Shared storage is a hallmark of traditional blockchains like Ethereum and many rollup solutions, where all transactions and state changes must be reflected across the network. Vara.eth avoids shared storage, instead decentralizing computations and managing state transitions dynamically through its architecture, enabling greater efficiency and scalability.

Slashing

A mechanism that penalizes Executors for malicious behavior or poor performance by reducing their staked collateral. This process ensures the economic accountability of Vara.eth participants and maintains the network's integrity.

Solidity

A high-level, object-oriented programming language specifically designed for writing smart contracts on blockchain platforms like Ethereum. It allows developers to define and implement the logic that powers decentralized applications (dApps).

SSTORE2

An Ethereum storage pattern that stores large data (such as validator key material) in minimal contracts instead of regular storage slots. Used by Router to persist aggregated keys and VSS commitments.

Symbiotic Operator Registry

The on-chain registry that tracks operators eligible to run Vara executors and receive delegated stake.

Symbiotic Protocol

A decentralized restaking system used by Vara.eth to select and manage Executors. It facilitates staking, distributes rewards, and enforces penalties, ensuring a secure and scalable compute network.

Vault (Symbiotic Vault)

A staking contract in Symbiotic where delegators deposit wVARA. The Vault aggregates stake, delegates it to chosen operators, and routes rewards and slashing penalties.

VetoSlasher

A Symbiotic module that enforces delayed slashing: slashing requests are opened with a veto/execute window to allow for checks before collateral is penalized.

VSS (Verifiable Secret Sharing)

A cryptographic primitive for splitting a secret into shares that can be verified individually and reconstructed only in quorum. Vara.eth uses VSS to distribute validator signing keys securely.

Wasm (WebAssembly)

A high-performance, lightweight binary format for executing code. Vara.eth uses Wasm programs to run decentralized computations efficiently and securely.

ZK Rollups (Zero-Knowledge Rollups)

A Layer-2 scaling solution that uses zero-knowledge proofs to validate transactions off-chain and post verified summaries on-chain. ZK Rollups employ cryptographic proofs (such as zk-SNARKs or zk-STARKs) to ensure the correctness of the batch without revealing the underlying transaction data. This approach enhances scalability, reduces gas costs, and offers faster finality compared to Optimistic Rollups, but at the cost of higher computational demands for generating proofs.

ZK-Verified Mode

A security mode in Vara.eth that uses zero-knowledge proofs to maintain full Ethereum-equivalent security. In this mode, every computation is cryptographically verified on mainnet, providing the highest level of security guarantees for applications requiring trustless verification.