Ethereum protocol study notes

Execution, consensus, networking, specs, testing

Compiled 18 August 2026

What this is. A map of the Ethereum protocol aimed at someone who can read code but has not worked inside a client. It is organised the way the protocol is organised rather than the way tutorials usually are: what a node *is*, then each layer, then how the rules get written down and tested.

How it was written. Every factual claim here was checked against a primary source on the day of writing — the Ethereum Foundation's own repositories, the executable specification, the EIP register, and in several cases the client source itself. Facts carry a bracketed source. Where a source could not settle a question, it says so rather than guessing.

Why the warnings. A lot of the callouts are of the form "the detail people get wrong". That is deliberate. Most Ethereum explainers are accurate at the summary level and wrong one layer down, and the wrong version is usually the memorable one. Where this document contradicts something you have read elsewhere, it is because the spec was opened.

Currency. Everything is current as of 18 August 2026 — Fusaka live, Glamsterdam scheduled. Fork contents and client shares move; the structure does not. Anything with a number in it is worth re-checking.

Contents

1 What an Ethereum node actually is **4**
- 1.1 Two programs, one node . . . 4
- 1.2 The four interfaces, and who is allowed to touch them . . . 4
- 1.3 Client diversity . . . 5
  - 1.3.1 Why it is a security property, not a preference . . . 5
- 1.4 What a node stores . . . 6
- 1.5 Where bugs live — the security map . . . 7

2 The execution layer **8**
- 2.1 The state, and where the root comes from . . . 8
- 2.2 Transactions: five types, and why . . . 8
- 2.3 Gas and the fee market . . . 9
  - 2.3.1 EIP-1559: what a sender actually pays . . . 10
  - 2.3.2 EIP-2929: warm and cold . . . 10
- 2.4 The EVM in one page . . . 10
- 2.5 The state transition, and where the tests hook in . . . 11
- 2.6 Receipts, logs and the mempool . . . 11

3 The consensus layer **12**
- 3.1 Time: slots, epochs, committees . . . 12
- 3.2 Fork choice: LMD-GHOST . . . 12
- 3.3 Finality: Casper FFG . . . 13
- 3.4 Slashing . . . 14
- 3.5 Validators, balances, and what Pectra changed . . . 14
- 3.6 Deposits, withdrawals, and the EL round trip . . . 14
- 3.7 Blobs and data availability . . . 15

4 Networking **16**
- 4.1 Execution layer: devp2p . . . 16
  - 4.1.1 What `eth/69` changed, and why it is a good example . . . 17
- 4.2 Consensus layer: libp2p . . . 17
- 4.3 Eclipse and the peer table . . . 18

5 The APIs **19**
- 5.1 The Engine API . . . 19
  - 5.1.1 One slot, end to end . . . 19
  - 5.1.2 Versions, by fork . . . 20
  - 5.1.3 Authentication — the exact details . . . 20
- 5.2 The JSON-RPC API (execution layer) . . . 20
- 5.3 The Beacon API (consensus layer) . . . 21
- 5.4 Summary: the three trust models . . . 22

6 Specifications, forks, testing, disclosure **23**
- 6.1 Where the chain is . . . 23
  - 6.1.1 Fusaka — what is live . . . 23
  - 6.1.2 Glamsterdam — what is under review . . . 23
- 6.2 The EIP process . . . 23
- 6.3 EELS: the executable specification . . . 24

6.3.1 The pipeline 25
6.3.2 The version checker that already exists 25
6.4 Hive 25
6.5 Who is looking for bugs, and with what 26

CHAPTER 1

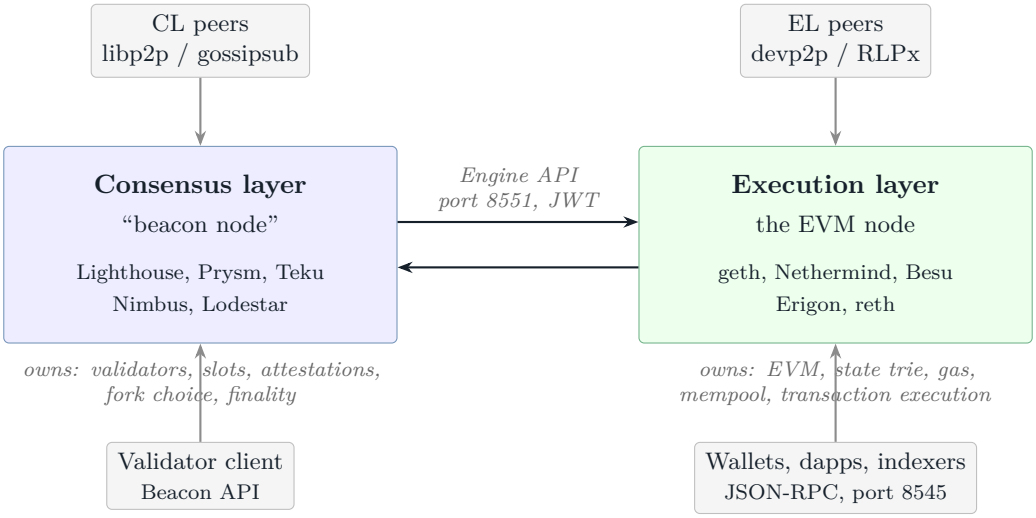
What an Ethereum node actually is

Why this chapter exists

Execution layer, consensus layer, networking, specifications, testing, client implementations. Those six words are the whole syllabus. This chapter is the map; Chapters 2–6 are the territory.

1.1 Two programs, one node

Before the Merge, one program did everything. Since 15 September 2022, running Ethereum means running **two processes** — normally, but not necessarily, on the same machine — that trust each other over one authenticated socket. Note the "not necessarily": the Engine API is authenticated precisely so it survives *not* being a loopback socket, and `authentication.md` names an internet-exposed RPC port as a threat it mitigates.



The distinction to never get wrong

The CL decides *which* chain. The EL decides *what the chain means*. The CL never executes a transaction; the EL never chooses a head. Each can be correct while the other is wrong, and the chain still breaks — which is exactly why a security team is organised around the seam between them.

1.2 The four interfaces, and who is allowed to touch them

Everything a node exposes falls into one of four buckets. Knowing which is which is most of what "attack surface" means for a node operator.

Interface	Default	Purpose	Exposure
JSON-RPC (EL)	8545 / 8546	what dapps and wallets call	public-ish
Engine API	**8551**	CL drives EL	**localhost, JWT**
Beacon API (CL)	varies	validator client, monitoring	local
devp2p (EL)	30303	peer discovery + block/tx gossip	internet
libp2p (CL)	9000	peer discovery + block/attestation gossip	internet

[**8551** is the only one that appears in a spec at all — and be precise: `execution-apis/src/engine/authentication.md` calls it "the default port", and its **MUST** is that the Engine API sit on a port *separate* from the JSON-RPC one, not that the number be 8551. The rest are client defaults. The Beacon API spec fixes no port — 5052 is Lighthouse/Nimbus/Grandine, Teku 5051, Prysm 3500, Lodestar 9596]

The Engine API is the only authenticated one. It is the only interface where a caller can *tell the node what the head of the chain is*. That is why it carries a JWT and why the spec puts it on its own port. Detail in §5.1.

The two p2p interfaces are the only ones facing strangers. Everything arriving there is adversarial by construction, and discv5 denial of service is an active research area (Chapter 4).

1.3 Client diversity

There is no "the" Ethereum client. Each layer has five *majority-share* independent implementations, and running the network means running one of each. The table below is those ten — but note it is not the whole list. There is a stable tail: **Grandine** (Rust) and **Caplin** (Go, bundled with Erigon) on the consensus side, **Ethrex** (Rust) on the execution side. Seven stable CL clients and six EL clients are listed today, so "five each" is shorthand for majority share, not a count.

Execution layer		Consensus layer	
geth	Go	Lighthouse	Rust
Nethermind	C#	Prysm	Go
Besu	Java	Teku	Java
Erigon	Go	Nimbus	Nim
reth	Rust	Lodestar	TypeScript

Why this table is the language list of the ecosystem

Whenever you see "Go, Rust, Java, C#, Nim, Python" in an Ethereum context, that is not a generic skills list — it is this table, plus Python for the executable specification (§6.3). Rust gets you Lighthouse and reth; Go gets you geth, Erigon and Prysm.

TypeScript (Lodestar) and C++ (Silkworm, pre-alpha) are the two that usually get left off such lists.

1.3.1 Why it is a security property, not a preference

Three thresholds, and they do different things. Get them in the right order, because mixing them up is a cheap way to look shaky on consensus.

Above 1/3 — finality stops. Casper FFG needs two thirds of the total *active* effective balance to justify a checkpoint. A client holding more than a third that crashes, or that votes for the wrong thing, denies that two-thirds majority. The chain keeps producing blocks; nothing finalises. **This is the number people actually watch**, because it is the lowest one that hurts.

Above 1/2 — the head follows the bug. LMD-GHOST picks the heaviest subtree. A majority of attestation weight decides which branch is canonical, so a buggy majority client's chain wins the fork choice even when the honest minority disagrees. It still cannot finalise on its own.

At or above 2/3 — an invalid chain can finalise. The spec threshold is *inclusive* (`target_balance * 3 >= total_active_balance * 2`), so exactly two thirds already clears it. Now the buggy client alone justifies, and its chain gains economic finality. This is the catastrophic case and the reason the 2/3 line exists in the argument at all.

Below all three. A client bug is an incident for that client's operators. The chain survives.

The point everyone misses about the dashboards

All three thresholds are on stake weight — sums of effective balances — *not* on validator counts or node counts. The published dashboards are a mixed bag on exactly this point, and **it is wrong to say `https://clientdiversity.org` measures headcount**:

- **Blockprint** (the CL panel's default) classifies *proposed blocks*, and `compute_proposer_index` samples by effective balance — so it is the closest thing to a *stake-weighted* estimator, not a count.
- **Miga Labs** crawls and counts *beacon nodes*.
- **supermajority.info** is entity self-reports, weighted by validator count.
- **Ethernodes** counts EL *nodes*.

Since Pectra, effective balances range from 32 to 2048 ETH (§3.5), so a headcount and a stake weight can diverge materially. **Knowing which chart is a stake proxy and which is a headcount** is the whole point: the published metric is not the metric the safety argument runs on.

Check the live numbers — and cross-check two sources

Shares move, and the dashboards disagree. Look them up before you rely on them, and **never quote a single source**: as of 18 August 2026 the Blockprint panel was showing an obviously broken **Teku 99.83% / Lighthouse 0.16%**, against Rated's Teku 53.9 / Prysm 21.2 / Lighthouse 20.6 and Miga's Lighthouse 51.0 / Prysm 20.8. Quoting the broken number would be far worse than quoting none. "I checked this week and geth was still the largest EL" is safe; a precise percentage from one panel is not.

1.4 What a node stores

Worth having in your head, because "where does the state live" underlies both sync strategies and a whole class of resource-exhaustion bugs.

EL: the state. Account balances, nonces, contract code, contract storage — a key/value store, committed into a Merkle Patricia Trie whose root goes in every block header (§2.1).

EL: the chain. Headers, bodies, receipts. Growing without bound, which is why history expiry appears in `eth/69` (§4.1).

CL: the `BeaconState`. The validator registry, balances, justified and finalised checkpoints, randomness (RANDAO), historical roots. One large SSZ object.

Blobs. Held only for a rolling window, then discarded — blob data was never meant to be permanent (§3.7).

1.5 Where bugs live — the security map

This is the frame to carry through the rest of the document. Four lanes, and almost every Ethereum security finding lands in one of them.

Execution
gas mispricing, state-root divergence, EVM edge cases, under-charged opcodes

Consensus
fork-choice manipulation, finality stalls, attestation arithmetic, slashing logic

Networking
discv5 and gossip DoS, eclipse attacks, mal-formed wire messages, sync poisoning

Cross-client
two clients disagreeing = a chain split, whatever the cause

The lane that behaves differently

The first three lanes are "one implementation got something wrong". **Cross-client is not a fourth bug type** — it is what the other three turn into once more than one client is running. A gas-accounting slip in a single client is a bug in that client; the same slip when the client holds a third of the network is a finality stall. Keep that distinction in mind while reading Chapter 3: the thresholds are what convert a defect into an event.

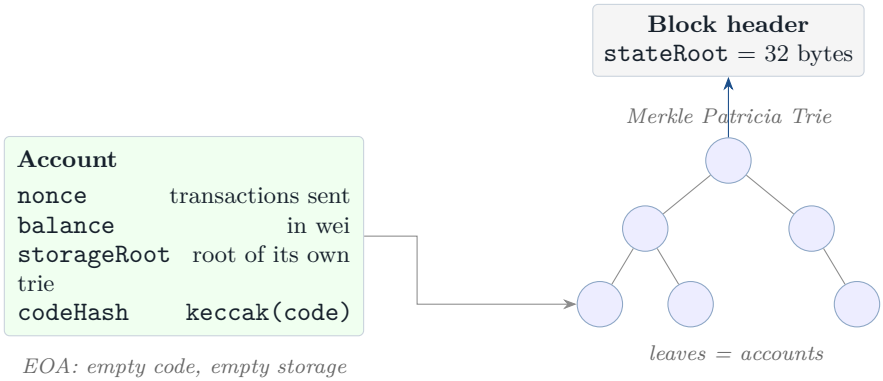
CHAPTER 2

The execution layer

The canonical source for everything in this chapter is `ethereum/execution-specs` — the executable Python implementation (§6.3). Where a claim below is precise, it was read from there or from the EIP rather than from a summary.

2.1 The state, and where the root comes from

Ethereum's world state is a mapping from 20-byte addresses to accounts. Every account, whether a wallet or a contract, has exactly four fields.



The trie is the commitment. Every account, and every storage slot inside every contract, is a leaf. The root is 32 bytes and it goes in the block header. Change one storage slot anywhere and the root changes.

Which is why state-root divergence is *the* consensus bug. If two clients execute the same block and compute different roots, they have different chains. There is no partial disagreement — the hash either matches or it does not.

The line worth having

> "The state root is a single 32-byte assertion that two implementations executed identically. That's what makes differential testing so effective here — you don't need to compare behaviour field by field, you compare one hash, and it's either equal or you have a finding."

2.2 Transactions: five types, and why

Transaction types were introduced by EIP-2718 (typed transaction envelope) so new formats could be added without breaking old ones. The first byte of a *typed* transaction is its type. **EIP-2718 defines the envelope range as `0x00`–`0x7f`** — "a positive unsigned 8-bit number between `0` and `0x7f`" — and `0x00` has never been assigned, so every type in use today is `0x01`–`0x04`. Say `0x00`–`0x7f` and then the caveat; saying `0x01` is the one number 2718's author checks.

Type	EIP	Introduced	What it added
(none)	—	genesis	legacy: bare RLP, `gasPrice`, no envelope
`0x01`	2930	Berlin	access lists (pre-declare what you touch)
`0x02`	1559	London	base fee / priority fee split
`0x03`	4844	Dencun	blob-carrying transactions
`0x04`	7702	Pectra	an EOA can delegate to contract code

[types 0x03 and 0x04 read from `spec.py` in `eip4844_blobs` and `eip7702_set_code_tx`; envelope rules from EIP-2718 and `forks/prague/transactions.py`]

The detail people get wrong

A legacy transaction has no type byte. EIP-2718 says a transaction is either `TransactionType || TransactionPayload` *or* a `LegacyTransaction`, and a legacy one is bare RLP — recognised because its first byte is an RLP list header. EIP-2718's rationale says "$\geq$ `0xc0`"; the executable spec narrows it to `[0xc0, 0xfe]` (`elif tx[0] >= 0xC0: assert tx[0] <= 0xFE`). Quote the range, not the inequality — it turns a read-the-EIP answer into a read-the-code one. The reference implementation's `encode_transaction` returns a legacy transaction unprefixed and only prepends `\x01`, `\x02` and so on for typed ones.

On `0x00`: `"type":"0x0"` is how `execution-apis`' JSON-RPC schema *represents* a legacy transaction — `TransactionLegacyUnsigned`, `pattern: ^0x0$`, which is normative OpenRPC, not merely a convention. What it is *not* is a `TransactionType` assigned under EIP-2718: the spec's `decode_transaction` raises `TransactionTypeError` on `tx[0] == 0`. Saying "legacy is type zero" is the kind of small wrongness this team notices — but so is calling the schema a convention.

2930 access lists. You declare the addresses and slots you will touch; you pay a bit up front and get cheaper access later. Relevant because **EIP-7928 (Glamsterdam) makes an access list mandatory at the *block* level** — see §6.1.2.

7702 is the strange one, and get this detail right. The EIP is titled *Set Code for EOAs* and its own description says it "**permanently** sets the code for an EOA". A transaction carries authorization tuples `[chain_id, address, nonce, y_parity, r, s]`; for each, a delegation indicator — the three bytes `0xef0100` followed by the address — is written into the authorizing account's *code*. It **persists** until another authorization replaces it or resets it to the zero address; the EIP has a section called "Persistence of code delegation".

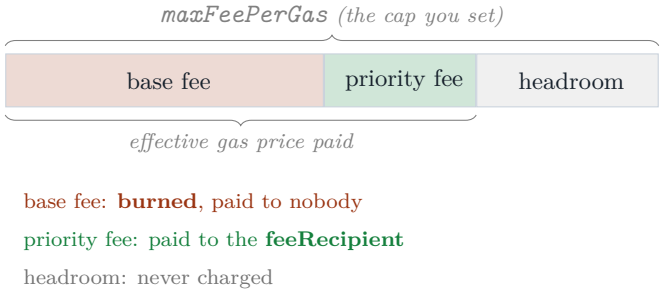
The trap in this one

The early draft was "set EOA account code *for one transaction*" — that phrasing still survives in the EIP's own `discussions-to` URL — and it is the version most people half-remember. **It is not what shipped.** Saying "for one transaction" to someone on this team marks you as having read a summary rather than the EIP. What makes it interesting to a security team is exactly the persistence: an address that was an EOA yesterday can execute code today, permanently, and every tool that treated `codeHash == keccak("")` as a stable property of an account had to be revisited.

2.3 Gas and the fee market

Gas exists to bound computation: every operation costs, and a block has a limit. Two mechanisms are worth knowing in detail because both are recurring sources of consensus bugs.

2.3.1 EIP-1559: what a sender actually pays



The base fee is set *by the protocol*, per block, from how full the previous block was: above the target it rises, below it falls. It is burned. The sender sets `maxFeePerGas` (total cap) and `maxPriorityFeePerGas` (the most they will tip); they pay base fee plus tip, and the rest of the cap is never charged.

Who actually gets the tip

EIP-1559 credits the block's `feeRecipient` (the coinbase) — *not*, strictly, "the proposer". The CL only supplies a `suggestedFeeRecipient` through `forkchoiceUpdated`, and the spec says the built payload **MAY deviate** from it. That is precisely what an external builder does: it keeps the tips and pays the proposer out of band. The distinction is small until you are talking to people working on ePBS (§6.1.2), where it is the whole subject.

2.3.2 EIP-2929: warm and cold

Since Berlin, the first access to an address or storage slot in a transaction costs substantially more than subsequent accesses — "cold" versus "warm". This was a repricing to defend against state-access DoS, and the accounting (which set is warm, when it resets, what a reverted frame does to it) is delicate.

Glamsterdam revisits this, but be precise about how. **EIP-8038** (live title: state-access gas cost *update*, though EIP-7773's index says "increase") raises `COLD_ACCOUNT_ACCESS` from 2,600 to 3,000, adds `ACCOUNT_WRITE` and `STORAGE_WRITE` surcharges, and also lifts `ACCESS_LIST_ADDRESS_COST` 2,400→2,900, `ACCESS_LIST_STORAGE_KEY_COST` 1,900→2,000 and `STORAGE_CLEAR_REFUND` 4,800→11,616, and charges `EXTCODESIZE`/`EXTCODECOPY` for their second DB read. It explicitly leaves `COLD_STORAGE_ACCESS` and `WARM_ACCESS` unchanged, and leaves the 2929 cold/warm rules themselves alone. **EIP-8037** is a different thing — it reprices state *creation*, not access. Both are scheduled for Glamsterdam and both are still status *Review*.

Why gas is a security topic, not an economics topic

Every historical DoS attack on Ethereum has the same shape: *an operation whose real cost to a node exceeds its gas price.* The 2016 Shanghai attacks were exactly this, and the fix was a repricing hard fork (EIP-150). When a fork changes gas costs — as Glamsterdam does in at least four EIPs — the question a security researcher asks is "what is now underpriced relative to what it makes the node do?"

2.4 The EVM in one page

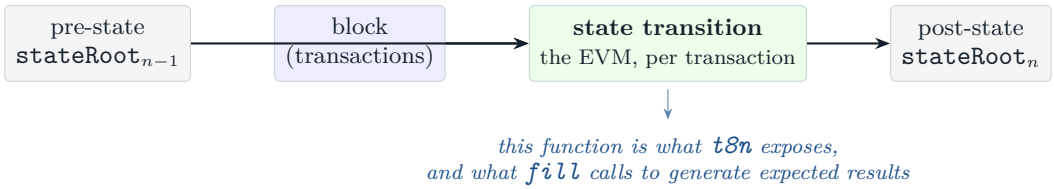
A stack machine with 256-bit words. Max stack depth 1024 (and, separately, a call-depth limit of 1024— note the spec confusingly names the *call*-depth constant `STACK_DEPTH_LIMIT`, which is a nice detail to have if the distinction comes up). The word size is usually explained as making Keccak and elliptic-curve arithmetic natural — that is the standard account, but it is design folklore rather than something a spec states, so offer it as "the usual explanation", not as fact.

Where data lives. Stack (words, cheap, transient); **memory** (byte-addressed, transient; cost is $3w + w^2/512$ for w words — the quadratic term is already 10% of the linear one by about **154** words and overtakes it at exactly **1,536** words, i.e. 48 KiB. *Do not say "it only bites past 700 words"*: at 700 the quadratic term is already 957 gas against 2,100 linear, and this is a sentence that invites a whiteboard check); **storage** (persistent, in the trie, expensive — `SSTORE`); **transient storage** (`TSTORE`/`TLOAD`, EIP-1153, cleared at the end of the transaction); **calldata** (read-only input); and the **returndata** buffer from the last call. Do not say "four regions" — transient storage and returndata are both live on mainnet.

Call frames. `CALL`, `DELEGATECALL`, `STATICCALL` create nested frames with their own memory. `DELEGATECALL` runs the callee's code against **the calling contract's own address and storage**, with `msg.sender` and `msg.value` inherited unchanged and no value transfer (`caller=evm.caller`, `to=evm.current_target`, `should_transfer_value=False`). *Say "the calling contract's"*, not "the caller's" — the latter reads as `msg.sender` and is the ambiguity that makes people think they misunderstand proxies. Reverts unwind the frame's state changes but not the gas already burned.

Determinism is the whole point. Given the same pre-state and the same transaction, every correct implementation must produce a byte-identical post-state. There is no room for "implementation defined".

2.5 The state transition, and where the tests hook in



`t8n` — "transition" — is the standard CLI shape for this function: give it a pre-state, a set of transactions and an environment, get back a post-state and receipts. **Most** major clients ship one — EEST's supported list is `evmone`, go-ethereum (`evm t8n`), Besu (`evmtool t8n-server`), nimbus-eth1, ethereumjs, and the specification itself via `ethereum-spec-evm`. **Nethermind, Erigon and reth are not on that list**, so do not say "every major client". **That uniformity is what makes cross-client differential testing possible at all**, and it is the mechanism underneath the proposal in Chapter **??**.

2.6 Receipts, logs and the mempool

Receipts record the outcome of each transaction: status, cumulative gas used, and the logs it emitted. Note from §4.1 that `eth/69` **removed the bloom filter from receipts sent over the wire**, because it is derivable and was pure bandwidth.

Logs are how contracts emit events. EIP-7708 (Glamsterdam) would make plain ETH transfers emit a log too — a change that sounds cosmetic and is not, because every indexer's assumptions shift.

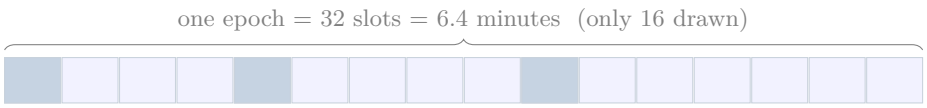
The mempool is the set of pending transactions each node holds. It is *not* consensus — nodes may hold different sets — but it is a real attack surface: eviction policy, replacement rules, and resource limits are all places where a cheap message can cost a node a lot. Mempool eviction bypasses are a recurring bug class across every chain that has one.

CHAPTER 3

The consensus layer

Canonical source: `ethereum/consensus-specs`. One directory per fork under `specs/`: `phase0`, `altair`, `bellatrix`, `capella`, `deneb`, `electra`, `fulu`, `gloas`, `heze`. [gh api repos/ethereum/consensus-specs/contents/specs]

3.1 Time: slots, epochs, committees



each slot = 12 s, one proposer, the validator set split into committees that attest

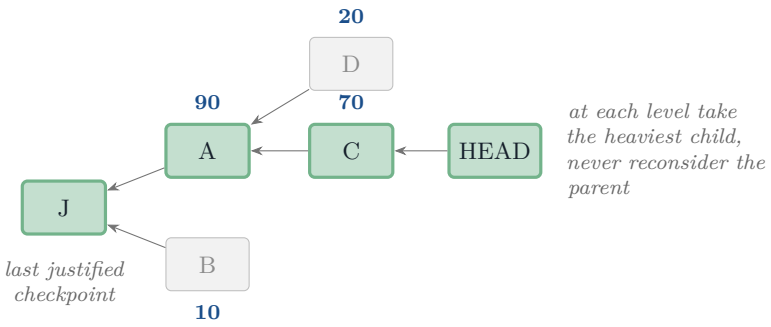
Slot. 12 seconds. Exactly one validator is selected to propose. It may propose nothing (a missed slot) — that is normal and not an error. *Naming note:* the constant is no longer `SECONDS_PER_SLOT` — the mainnet config now has `SLOT_DURATION_MS: 12000`, with the neighbouring timing expressed in basis points (`ATTESTATION_DUE_BPS`, `PROPOSER_REORG_CUTOFF_BPS`). Older material will use the old name.

Epoch. 32 slots. The unit on which the protocol does its bookkeeping: rewards, penalties, activations, exits, and finality.

Committee. Each epoch the validator set is shuffled and divided so that every validator attests exactly once per epoch, in exactly one slot.

Attestation. A vote with three parts: the **head** it thinks is canonical, and a **source** → **target** checkpoint pair for finality. One message serving both the fork choice and finality is why the two mechanisms interlock.

3.2 Fork choice: LMD-GHOST



LMD — latest message driven: only each validator's *most recent* attestation counts. **GHOST** — greediest heaviest observed subtree: start at the last justified checkpoint and repeatedly walk to the child with the most accumulated attestation weight.

Two refinements the diagram leaves out

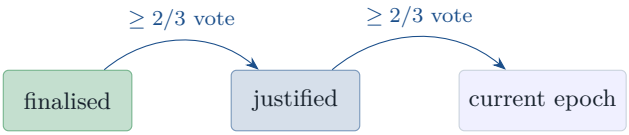
The walk is over a filtered tree, not the raw one: `get_head` walks `get_filtered_block_tree`.

And **weight is not only attestations** — `get_weight` is *attestation score plus proposer score*, so **proposer boost is part of the fork choice**, with ties broken lexicographically on block root. Fork choice weight also *excludes* slashed validators — the opposite convention to the FFG denominator below. Two mechanisms, two conventions, one spec: that asymmetry is the thing to remember.

Two properties that generate bugs

Weight is a sum of effective balances in Gwei, so it is large-integer arithmetic over attacker-influenceable quantities. Any place where a supermajority threshold is computed in a narrower type than the total it is derived from is a bug — and the dangerous half is not the overflow panic, it is the silent wrap in a release build, because that produces a wrong answer instead of a crash. And **"latest message" means state per validator**, which is memory an adversary can try to grow. Fork-choice implementations have historically been a rich source of both correctness and resource bugs.

3.3 Finality: Casper FFG



a justified checkpoint whose direct child is also justified becomes finalised — so finality normally lags the head by two epochs, about 13 minutes

Epoch boundaries are *checkpoints*. Two thirds of the total *active* effective balance voting for the same source→target pair *justifies* the target; when a justified checkpoint's child is justified in turn, the parent becomes *finalised*.

Slashed validators stay in the denominator

The membership test for the denominator is `is_active_validator`: `activation_epoch <= epoch < exit_epoch`. **There is no `slashed` check in it.** Pending and exited validators are outside the denominator; a *slashed* one is not — `slash_validator` calls `initiate_validator_exit`, which sets a *future* exit-queue epoch, so it keeps counting in `get_total_active_balance` for several epochs. Meanwhile `get_unslashed_participating_indices` drops it from the *numerator* immediately.

Numerator and denominator use different filters, and that asymmetry is deliberate — a slashed validator should stop *helping* justify immediately, but removing it from the denominator early would make the remaining honest stake look like a larger share than it is.

Why that is economic, not just logical. Reverting a finalised block requires validators controlling at least a third of the stake to have made provably contradictory votes — and those votes are slashable. The cost of reversal is denominated in ETH, not in hash power.

Inactivity leak. If the finality delay *exceeds* four epochs — `is_in_inactivity_leak` is `get_finality_delay(state) > MIN_EPOCHS_TO_INACTIVITY_PENALTY`, strictly greater, with the constant `Epoch(2**2)` — validators that are not attesting start losing stake, until the *participating* remainder is again two thirds. The chain heals by shrinking the denominator. This is the mechanism a liveness attack has to fight. (You make a point of $> 1/3$ versus $\geq 1/3$ elsewhere; the same precision applies here.)

3.4 Slashing

Two offences, both provable from signed messages alone:

Proposer slashing. Signing two different blocks for the same slot.

Attester slashing. A *double vote* — two *different attestations* sharing a target *epoch*; the predicate is `data_1 != data_2 and data_1.target.epoch == data_2.target.epoch`, so two votes with the same target but a different head still qualify — or a *surround vote* (one attestation's source/target span strictly containing another's). Surround voting is the one that matters: it is the move you would have to make to revert finality.

3.5 Validators, balances, and what Pectra changed

Constant	Value	Meaning
`MIN_ACTIVATION_BALANCE`	32 ETH	still the minimum to run a validator
`MAX_EFFECTIVE_BALANCE_ELECTRA`	2048 ETH	Pectra, **0x02 compounding only**

[EIP-7251, parameter table: `Gwei(2**5 * 10**9)` and `Gwei(2**11 * 10**9)`]

Before Pectra every validator was capped at 32 ETH of *effective* balance, so large operators ran thousands of redundant validators. EIP-7251 raised the cap to 2048 ETH so they can consolidate, shrinking the validator set and with it the number of P2P messages and BLS signatures per epoch.

But the raise is opt-in. 2048 applies only to validators holding `0x02` compounding withdrawal credentials — `get_max_effective_balance` branches on exactly that, and a `0x00` or `0x01` validator is still capped at `MIN_ACTIVATION_BALANCE`, 32 ETH. Do not state 2048 flatly; the conditional is the part that gets misremembered.

Why a security researcher cares

Raising `MAX_EFFECTIVE_BALANCE` means **attestation weight per validator is no longer near-uniform**. Every piece of arithmetic that implicitly assumed "one validator, one unit of weight" had to be revisited — and so did every piece that sized a variable for a 32-ETH ceiling. A 64× increase in the per-validator maximum is exactly the kind of change that turns a comfortably-wide integer into a marginal one.

3.6 Deposits, withdrawals, and the EL round trip

Deposits (EIP-6110). Deposits are read from the execution layer's block directly, rather than being voted on — removing a whole polling mechanism and its delay.

Withdrawals (Capella). Automatic and push-based: the protocol sweeps validator balances into execution-layer accounts, no transaction required.

Triggerable exits and partial withdrawals (EIP-7002). Not *any* execution-layer address — only the address embedded in the validator's **own execution withdrawal credential**. `process_withdrawal_request` requires `has_execution_withdrawal_credential(validator)` **and** `validator.withdrawal_credentials[12:] == request.source_address`. **Say "`0x01` *or* `0x02`"**: the EIP's title says `0x01` only because it was written before EIP-7251, and as merged the predicate accepts both prefixes — `0x02` compounding validators are the main consumer of the partial-withdrawal path. That restriction is the point of the EIP: the cold key that owns the stake gains power over the hot active key. It covers partial withdrawals too, not just full exits.

Note the pattern: these EIPs make the CL and EL talk about the *same* objects. Each one adds a place where the two layers must agree, and therefore a place where they can disagree.

3.7 Blobs and data availability

EIP-4844 (Dencun). Introduced blob-carrying transactions (type `0x03`): large data attached to a block, priced by a separate blob fee market, and **deleted after a rolling window**. Rollups post data here.

EIP-7594 PeerDAS (Fusaka, live). Nodes no longer download every blob. They sample portions and rely on erasure coding to conclude, with high probability, that the whole is available.

BPO forks (EIP-7892, Final — and *Informational*, not Standards Track). Scheduled forks that change *only* blob parameters, so capacity can be raised without a full upgrade. `bpo1`–`bpo5` exist as fork packages in `execution-specs` (*not* in `consensus-specs`, which mentions BPO only in prose). **But only two are scheduled on mainnet**: `BLOB_SCHEDULE` has exactly two entries — epoch 412672 → 15 blobs, epoch 419072 → 21 — matching EIP-7607, which defines only BPO 1 and BPO 2. Saying "five BPO forks" would be wrong; say "five fork packages exist, two are scheduled".

The security shift worth naming out loud

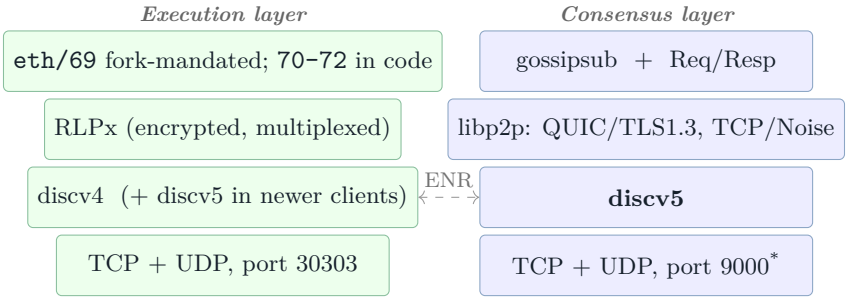
Under 4844 availability was a fact you verified. Under PeerDAS it is a *probability you estimate*. Attacks stop being "make the node accept invalid data" and become "make the node's sampling conclude available when it is not". That is a different discipline, and it is the newest large surface on the network.

CHAPTER 4

Networking

This is usually the least-studied of the four lanes and the one where the attack surface is most obviously adversarial: everything arriving here comes from strangers.

The two layers do *not* share a network stack. They evolved separately and only discovery has a common ancestor.



[CL stack read from `consensus-specs/specs/phase0/p2p-interface.md`, which has sections titled "The gossip domain: gossipsub", "The discovery domain: discv5", "SSZ-snappy encoding strategy" and "Why are we using Noise?". *__The ports are not in that document__ — it specifies only ENR fields. 9000 is the Lighthouse/Teku/Nimbus/Lodestar default; Prysm uses TCP 13000 / UDP 12000. 30303 is likewise convention, not spec.]

4.1 Execution layer: devp2p

Discovery: discv4. A Kademlia-style DHT over UDP. Nodes are identified by their **secp256k1 public key** — the "node ID" — and the distance metric is `keccak256(n1) XOR keccak256(n2)`. Each node *also* maintains an **ENR** (Ethereum Node Record), a signed key/value record served on request; that was bolted on later by EIP-868, and it is the one piece both layers share. The ENR is not the identifier.

Transport: RLPx. An encrypted, authenticated, multiplexed session over TCP. Capabilities are negotiated in a handshake.

`eth` wire protocol. Block and transaction propagation, header and body requests. `eth/69` is **the last version a network upgrade required** — Fusaka mandated it. *Do not say "eth/69 is what is deployed"*: the spec head and the shipped code have both moved past it. `devp2p` master documents `eth/72` and `snap/2`, and the defining EIPs — **7975** `eth/70` (partial block receipt lists), **8159** `eth/71` (block access list exchange), **8070** `eth/72` (sparse blobpool), **8189** `snap/2` (BAL-based state healing) — are all status Review, targeting Glamsterdam. [EIP-7773 "Other EIPs / Networking EIPs"; `devp2p/caps/eth.md` "The current protocol version is eth/72". Note the `devp2p` documents carry no status field of their own — Review is the *EIP* status]

The sentence that shows you read code, not just specs

"`eth/69` is the last fork-mandated version — but geth already negotiates `eth/72`." Shipped state, August 2026: go-ethereum v1.17.5 advertises `[ETH72, ETH71, ETH70, ETH69]` with 72 first and no fork gating, so geth-to-geth speaks 72 today; Nethermind implements through `eth/70`; **reth is the outlier**, still topping out at `eth/69`. A wire-version spread across live clients is a compatibility

surface, and naming who is where is a genuinely networking-flavoured answer from the layer you called your weakest.

`snap`. A separate protocol for fast state sync — serving flat ranges of accounts and storage rather than trie nodes.

4.1.1 What `eth/69` changed, and why it is a good example

EIP-7642, *"eth/69 — history expiry and simpler receipts"*, status Final, category Networking. Three changes, quoting the abstract: [eips.ethereum.org/EIPS/eip-7642]

1. Nodes **announce the historical block range they serve**, so peers stop asking for history that was expired away.
2. The handshake **drops total difficulty**, which has been meaningless since the Merge.
3. Receipts sent over the wire **drop the bloom filter**, because the receiver can recompute it.

Why to mention this one

It is a compact example of protocol hygiene: remove a field that is derivable (bandwidth), remove a field that is obsolete (post-Merge), and let nodes state what they can actually serve. If asked "what have you read recently", this is a small, real, verifiable answer — and item 3 is the kind of change where a security reviewer asks "who was trusting that bloom, and do they now recompute it correctly?"

The forward-looking version: "Glamsterdam has `eth/70`, `71`, `72` and `snap/2` scheduled — three wire-protocol bumps in one fork, and two of them (`eth/71`, `snap/2`) exist only because of block-level access lists. That's a lot of new parsing on the one interface that faces strangers." A networking observation with a security point attached.

4.2 Consensus layer: libp2p

Discovery: discv5. Same ENR idea, newer protocol. Unauthenticated UDP reached by strangers, so it is the textbook home of the three DoS shapes below.

Transport: libp2p, and QUIC is primary. Clients MUST support both QUIC (UDP) and TCP, and SHOULD prefer a peer's QUIC addresses. QUIC is secured with **TLS 1.3** and multiplexes natively; the *TCP fallback* uses the **Noise** handshake with `secp256k1` identities, with mplex/yamux for multiplexing. Messages are SSZ serialised and snappy compressed (`ssz_snappy`). Saying "libp2p with Noise" describes the fallback, not the primary path.

Gossip: gossipsub. Publish/subscribe over topics, with peer scoring. Topics visible in the spec's own table include `beacon_block`, `beacon_aggregate_and_proof`, `voluntary_exit`, `proposer_slashing`, `attester_slashing`, plus attestation subnets advertised via the `attnets` ENR field.

Req/Resp. Direct request/response streams for syncing blocks and blobs, separate from gossip.

Where the DoS lives, on both sides

Three recurring shapes, and they generalise well beyond Ethereum:

- **Unauthenticated work.** A stranger sends one cheap message that makes a node do something expensive. This is the entire discv5 DoS class, and it is the same shape as every unauthenticated-endpoint finding on every protocol: the asymmetry between what the message costs to send and what it costs to process.

- **Unbounded state.** Peer tables, gossip caches, pending queues — anything an adversary can grow.
- **Amplification.** A small request producing a large response, especially over UDP where the source is not verified.

4.3 Eclipse and the peer table

An **eclipse attack** surrounds a victim with attacker-controlled peers so it sees a filtered view of the network. On a proof-of-stake chain the payoff is usually not a double spend — it is making a validator attest to the wrong head, or miss its slot. Defences are all about making peer selection expensive to influence: identity derived from a key, buckets in the DHT, diversity requirements, peer scoring in gossipsub.

Where to start if you only read one thing

discv5. It is unauthenticated UDP that makes a node do work, which is the purest form of the first shape above, and it is small enough to read end to end in an afternoon. Everything else in this chapter is easier once you have one protocol at that level of detail rather than five at summary level.

CHAPTER 5

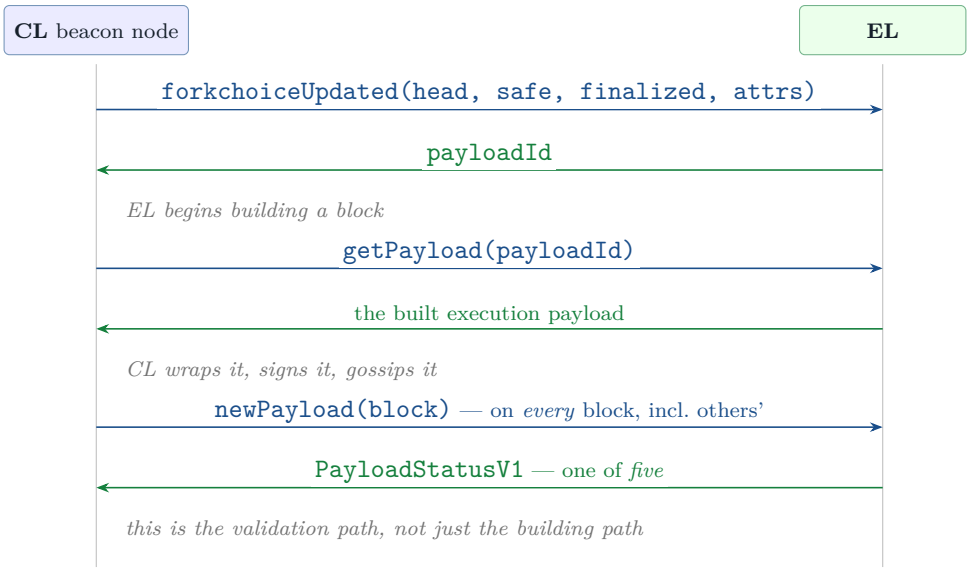
The APIs

Three API surfaces, three different trust models. Canonical sources: `ethereum/execution-apis` for the first two, `ethereum/beacon-APIs` for the third. [both verified live on 18 August 2026; `beacon-APIs` is active, described as "Collection of RESTful APIs provided by Ethereum Beacon nodes"]

5.1 The Engine API

The contract between the two halves of a node. Narrow, versioned per fork, and the only interface that can tell a node what the canonical chain is.

5.1.1 One slot, end to end



`engine_forkchoiceUpdated` tells the EL the head, the *safe* block and the finalised block — `ForkchoiceStateV1` carries **three** hashes, and `safeBlockHash` must be the head or an ancestor of it. If `payloadAttributes` is present it *also* means "start building", and returns a `payloadId`.

`engine_getPayload` collects the block the EL built.

`engine_newPayload` asks the EL to execute and validate a block — including blocks received from the network, not just ones it built.

`PayloadStatusV1` has *five* statuses, not three

`"VALID" | "INVALID" | "SYNCING" | "ACCEPTED" | "INVALID_BLOCK_HASH"`. The two interesting ones are the middle pair: `SYNCING` = "I cannot judge this yet", `ACCEPTED` = "valid, but I cannot attach it to a chain I know". Naming only three is a visible gap to anyone who has implemented against this. **And it changes in Bogota**: `PayloadStatusV2` drops `INVALID_BLOCK_HASH` and adds `inclusionListSatisfied`.

5.1.2 Versions, by fork

Spec file	Upgrade	Methods introduced there
`osaka.md`	Fusaka (live)	`getPayloadV5`, `getBlobsV2`, `getBlobsV3`
`amsterdam.md`	Glamsterdam	`newPayloadV5`, `forkchoiceUpdatedV4`, `getPayloadV6`, `getBlobsV4`, `getPayloadBodiesBy{Hash,Range}V2`
`bogota.md`	after that	`newPayloadV6`, `forkchoiceUpdatedV5`, `getInclusionListV1`

[directory listing + headings of `execution-apis/src/engine/`]

Why versioning per fork is itself a surface

Every fork adds a method version, and both clients must agree on which version to speak at which timestamp. A CL calling `V5` against an EL expecting `V6`, or either mis-detecting the fork boundary, is a node that stops following the chain at exactly the worst moment.

The sharper version of the question: **what happens if a reorg crosses a fork-activation timestamp?** Both layers have to decide which rules applied to a block that is being re-evaluated, and they have to agree. That is a genuinely under-tested corner on most chains.

5.1.3 Authentication — the exact details

From `execution-apis/src/engine/authentication.md`:

- Default port **8551**, exposed under the `engine` namespace.
- JWT. The EL **MUST** support at least **HS256** (HMAC + SHA256).
- A shared `jwt-secret`: a file holding a hex-encoded **256-bit** key. If not supplied, the client **SHOULD** generate one and **SHOULD** store it as `jwt.hex` for the counterpart to use — *SHOULD*, not MUST; every other MUST/SHOULD in this list is exact, so keep this one exact too.
- The EL **MUST** also reject `alg: none`. And `inproc` / raw IPC requires *no* authentication at all, on the assumption that local file access is already sufficient permission — an even sharper version of the WebSocket point below.
- Required claim `iat` (issued-at); the EL **SHOULD** only accept timestamps **within ±60 seconds** of now — a replay window bound.
- Optional claims `id` (a unique identifier for the individual CL client) and `clv` (CL client type/version). Unrecognised claims **MUST** be ignored.
- Over HTTP every request is authenticated individually; over WebSocket only the opening handshake is.

The WebSocket asymmetry is worth noticing

HTTP authenticates each message; WebSocket authenticates once and then trusts the stream. That is a deliberate trade-off in the spec, and the kind of detail that shows you read the source rather than a summary. If someone gets that socket, the ±60-second replay bound no longer helps.

5.2 The JSON-RPC API (execution layer)

What wallets, dapps, block explorers and indexers actually call — port 8545 (HTTP) / 8546 (WebSocket). Organised into namespaces. Only some are actually specified in `execution-apis` — `eth`, `debug`, `txpool` and `engine`, plus a `testing` namespace, and `net_version`, which *is* specified, inside

`src/eth/client.yaml`. Only `web3_` and `admin_` are pure client convention. `debug_` and `admin_` are the operationally sensitive ones.

The `testing` namespace is worth a clause

`src/testing/testing_buildBlockV1.yaml` is specified as a "testing-only method… **MUST NOT** be exposed on public-facing RPC APIs". A spec-mandated method that must never be reachable is exactly the kind of thing a security reviewer asks about — who enforces the MUST NOT, and what happens on a node that ignores it?

Security posture. Unauthenticated by design. Exposure is managed by *which namespaces are enabled* and by not putting it on the internet. `debug_` and `admin_` in particular do work that is expensive or privileged.

`eth_config` (EIP-7910, Final, shipped in Fusaka). A method that reports "node-relevant configuration data for the current, next, and last known forks". Useful precisely because fork configuration mismatches are a classic operational failure — now a node can be asked what it thinks the schedule is. [eips.ethereum.org/EIPS/eip-7910]

Why this surface is the easy one to fuzz

`execution-apis` ships **OpenRPC** documents, and a machine-readable interface description is exactly what lets a fuzzer generate valid-but-hostile inputs instead of noise — it knows the shape of a legal request, so it can spend its budget on the interesting deviations. The same trick works with OpenAPI and Schemathesis on any REST surface. The practical catch is that these documents rot: an endpoint whose schema is wrong or whose examples are stale silently narrows what the fuzzer will ever try, so cleaning the spec is usually the first half of the work.

5.3 The Beacon API (consensus layer)

A standardised REST interface — `ethereum/beacon-APIs` — so that a validator client from one project can drive a beacon node from another. Endpoints cover beacon state and blocks, validator duties, and node health.

There is no standard port here

Unlike the Engine API, **the Beacon API spec fixes no port** — the OpenAPI document just uses `http://localhost/`. **5052** is Lighthouse's, Nimbus's and Grandine's default; **Teku** uses **5051** — and Teku's *validator* REST API is the one on 5052, which is the detail that catches people; **Prysm** uses 3500 and **Lodestar** 9596. Quoting 5052 as "the" beacon port is a small error that would land badly with people who run all of them — so do not write a sentence daring them to ask about Teku unless you know the answer is 5051.

Nimbus's own source makes the no-standard-port point for you: `defaultEth2RestPort* = 5052` carries the comment *"This is not part of the spec! But it's port which Lighthouse uses"*.

Why standardising it mattered. It decouples the validator client from the beacon node, which is a client-diversity mechanism: an operator can mix implementations rather than being locked to one vendor's pair.

Trust model. Local. A validator client asking a beacon node "what should I sign?" is trusting it completely — which is why the interface is not meant to face the internet.

5.4 Summary: the three trust models

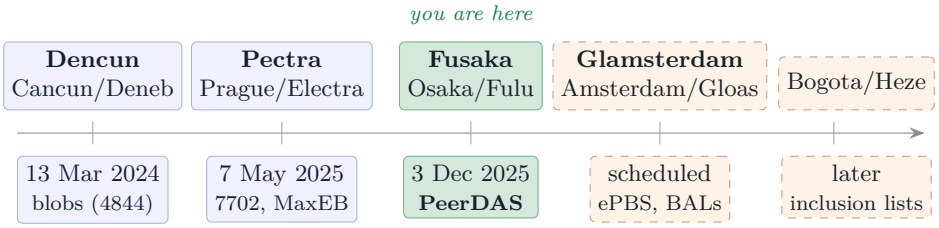
API	Who may call	If compromised
JSON-RPC	anyone permitted to	node resource abuse; wrong data to apps
Engine API	only the paired CL (JWT)	**you control what the node thinks the chain is**
Beacon API	only the operator	you control what the validator signs

CHAPTER 6

Specifications, forks, testing, disclosure

Where the rules are written down, how they get turned into tests, and where the chain is on the fork line.

6.1 Where the chain is



[ethereum.org/en/history for dates; EIP-7607 (Fusaka meta); EIP-7773 (Glamsterdam meta); `execution-apis/src/engine/` for Bogota]

6.1.1 Fusaka — what is live

EIP-7607 lists it. Core: **7594 PeerDAS**, 7823 MODEXP upper bounds, 7825 transaction gas limit cap, 7883 ModExp gas increase, 7917 deterministic proposer lookahead, 7918 blob base fee bounded by execution cost, **7934 RLP block size limit**, 7939 CLZ opcode, 7951 secp256r1 precompile. Plus 7892 BPO forks, 7642 `eth/69`, 7910 `eth_config`, 7935 default gas limit 60M.

7934 is the one to notice

An RLP *block size* limit exists because **a block can be cheap to produce and expensive for everyone else to process**. Gas bounds computation, but it does not bound the bytes a node has to hold and parse, so a block that is perfectly legal under the gas limit can still be a resource problem for every node that receives it. Ethereum shipped a consensus-level cap for that class in the last hardfork. Once you have that idea, "what work does a node do that nothing charges for?" becomes the most productive question you can ask about any protocol.

6.1.2 Glamsterdam — what is under review

EIP-7773, status Draft, activation timestamps still blank. Headline items: **EIP-7732 enshrined proposer/builder separation** (CL) and **EIP-7928 block-level access lists** (EL). Also on the list: 2780, 7610, 7688, 7708 (ETH transfers emit a log), 7778, 7843 `SLOTNUM`, 7954, 7976, 7981, 7997, 8024, **8037 and 8038** (state-creation and state-access gas increases), 8246, 8282; CL-side 8045 and 8061.

6.2 The EIP process

Types. Standards Track (Core / Networking / Interface / ERC), Meta, Informational. **Core** is broader than "needs a fork": EIP-1 defines it as improvements requiring a consensus fork *"as well as changes that are not necessarily consensus critical but may be relevant to core dev discussions"*. And a fork can ship non-Core EIPs — Fusaka shipped 7642 (Networking) and 7892 (Informational).

Status. Idea (pre-draft, not in the repo) → Draft → Review → Last Call → Final; plus Stagnant, Withdrawn, and **Living** — EIP-1 itself is *Living*. Live examples: 7928 is *Review*, 7892 and 7642 are *Final*, 7773 is *Draft*.

Scheduling. All Core Devs calls decide inclusion; a **meta EIP** tracks the set as proposed / considered / declined / **scheduled for inclusion** (SFI).

The operationally important fact: **an EIP is a markdown file with a git history, and it changes after tests are written against it**. That mutability is the entire premise of Chapter **??**.

Two live examples of that drift, in the meta EIPs themselves

You do not have to argue this abstractly — it is visible in the two documents this section just cited:

- EIP-7607 lists 7642 as *"eth/69 - Drop pre-merge fields"*. The EIP's actual current title is *"eth/69 - history expiry and simpler receipts"*.
- EIP-7773 lists 8038 as a state-access gas *"increase"*. EIP-8038's current title says *"update"*.

Neither is important on its own. What is important is that `spec_version_checker` catches **neither** — it hashes the *referenced* EIP, not the meta EIP's description of it. Two concrete instances beat a general argument every time.

6.3 EELS: the executable specification

The correction — know this cold

`ethereum/execution-spec-tests` is **archived** (`archived: true`), last push 2 July 2026. Everything moved into `ethereum/execution-specs`, which is active and pushed to constantly. **Most tutorials and blog posts still point at the old repo**, so check the date on anything you read about EEST. [gh api repos/ethereum/execution-spec-tests]

Careful with the date: 2 July 2026 is only the *final push* to a repo that was already being wound down — the merge was announced in the STEEL team's post of **4 November 2025**. "Archived in July 2026, folded into `execution-specs`" is accurate; "it moved in July 2026" is not.

One more current-repo detail worth having, because it signals you opened it rather than read about it: `execution-specs`'s **default branch is `forks/amsterdam`**, not `main` — the repo tracks the next fork on its default branch.

`ethereum/execution-specs` is the **Ethereum Execution Layer Specifications** (EELS, plural) — "an executable Python reference implementation of Ethereum's execution layer, along with the test cases that verify it". Maintained by the EF's **STEEL** team, in the repo's own words: "This repository is maintained by the STEEL Team at the Ethereum Foundation" (`https://steel.ethereum.foundation/`). [execution-specs README.md]

`src/ethereum/forks/` one package per fork: `frontier` through `prague`, `osaka`, `amsterdam`, plus `bpo1`–`bpo5`.

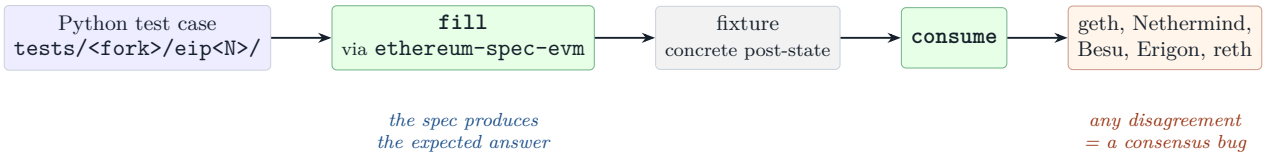
`tests/` one directory per fork, then per EIP — `tests/prague/eip7702_set_code_tx/`. 58 EIP directories today.

`packages/testing/` the test framework. **It is still called EEST** — see the warning below.

Do not say "formerly EEST"

EEST is not a former name. It is the live, everyday name inside `execution-specs`: the docs say "EEST has two commands, `consume` and `execute`", the navigation has an "EEST CLI Tools" section, `pyproject.toml` defines an `eest` CLI entry point, and the documentation URL is `https://eest.ethereum.org`. What changed is the *repository* it lives in and the PyPI distribution name (`ethereum-execution-testing`). Saying "formerly EEST" to a STEEL engineer who types `eest` every day is a small, immediate tell.

6.3.1 The pipeline



`ethereum-spec-evm` the reference EVM: a `t8n` transition tool, a `b11r` block builder, and a state-test runner. The oracle.

`fill` runs the Python cases through it to produce fixtures with real post-state roots. "Filling" = asking the specification for the answer.

`consume` replays fixtures against a production client.

`execute` runs against a live client or network.

`check_eip_versions`, `checkfixtures` hygiene — §6.3.2.

[`packages/testing/pyproject.toml`, `[project.scripts]`]

6.3.2 The version checker that already exists

`packages/testing/src/execution_testing/cli/pytest_commands/plugins/spec_version_checker/spec_version_checker.py` is a pytest plugin. For every module whose path matches `eip\d{1,4}` it injects a test that fetches the current blob hash of the referenced EIP from the GitHub API and fails on *any mismatch* — not specifically "behind". It also hard-exits if no `GITHUB_TOKEN` is supplied. The message, in full:

> *"The version of the spec referenced in {module} does not match that from ethereum/EIPs, tests might be outdated: Spec: {name}. Referenced version: {known}. Latest version: {latest}. The version was retrieved from {api_url}."*

Note what it does and does not do. It prints *both* versions, so it is more informative than a bare "out of date" alarm — but it compares **hashes, never content**. It can tell you the EIP moved. It cannot tell you *what* moved, whether any transcribed constant now disagrees with the text, or whether a test covers the delta. Chapter **??** is about that gap.

6.4 Hive

`ethereum/hive` — "a system for running integration tests against Ethereum clients", active.

Simulator a program in any language holding the test logic; it launches clients and reports results. `./simulators/<name>/Dockerfile`.

Client also Docker: `./clients/<name>/Dockerfile`. Some client builds may fail as long as one succeeds.

Simulation API the simulator container gets `HIVE_SIMULATOR`, the controller's HTTP URL, and uses it to start clients and report results. Logs in `./workspace/logs`.

Suites The `simulators/` tree has six top-level entries: `devp2p`, `ethereum`, `eth2`, `portal`, `smoke` and — worth noticing — `lean`. `devp2p` registers **five** suites in its `main.go`: `discv4`, `discv5`, `eth`, `snap`, `snap2`. Under `ethereum/` there are exactly six directories: `consensus`, `eels`, `engine`, `graphql`, `rpc-compat`, `sync`. [gh api repos/ethereum/hive/contents/simulators; simulators/devp2p/main.go]

Two names that will waste an afternoon

`ethereum/rpc` does not exist. It was removed on 9 September 2025 ("remove rpc in favor of rpc-compat"), and `--sim ethereum/rpc` fails. *But* Hive's own `docs/overview.md` **still documents it**, so you will find it in the docs and it will not run.

`ethereum/eels` is not a suite either. It has no `Dockerfile`; it is a directory of *five* simulators — `consume-engine`, `consume-enginex`, `consume-rlp`, `consume-sync`, `execute-blobs`. `--sim ethereum/eels` fails for exactly the same reason `ethereum/rpc` does. The real invocation is `--sim ethereum/eels/consume-engine`. This is the one that runs the reference-spec fixtures inside Hive.

General lesson: in this repo the `simulators/` tree is authoritative and the prose docs lag it. Check for a `Dockerfile` before assuming a path is runnable.

The `lean` simulator

A `simulators/lean` directory sitting next to `ethereum` and `eth2` is Hive being pointed at the Lean Ethereum track — the post-quantum / beam-chain research line. Nothing to study yet, but it is a useful thing to have noticed exists.

```
./hive --sim ethereum/engine --client go-ethereum,besu,nethermind
```

Public instances: https://hive.ethpandaops.io. **The distinction:** EELS fixtures test one client against the spec; Hive tests clients against each other and against the network protocols.

6.5 Who is looking for bugs, and with what

Fuzzing. Two tools to know by name — **and to distinguish, because they are not the same kind of thing**. **goevmlab** is differential fuzzing *across EVM implementations*: its `evms/` package drives geth, besu, nethermind, erigon, nimbus, evmone and the reference spec, with a generic fuzzer, a trace differ and a test minimiser. **tx-fuzz** (Marius van der Wijden) is a *live-network transaction* fuzzer: it sprays valid-but-hostile transactions at a running node. Calling tx-fuzz a differential fuzzer is a common slip.

Manual review. EIP review and PR review on the client repositories. Both are public and both are a realistic way in for someone learning — reading a fork's worth of EIP discussion is genuinely how people get up to speed.

Audits and audit competitions. The EF commissions external audits before forks and runs competitions. Public write-ups from these are some of the best study material available, because they show what a reviewer actually looked at.

Coordinated disclosure. https://bounty.ethereum.org redirects to `ethereum.org/en/bug-bounty/`: critical payouts up to $1,000,000, and **specification bugs are in scope**, not just implementation bugs. Worth reading the scope page carefully — it is effectively a list of what the people who wrote the protocol think can go wrong with it. https://security.ethereum.org is the Protocol Security team's own site (JavaScript-rendered, so it has to be opened in a browser).

One number never to quote from memory

Client diversity shares move, and the dashboards disagree with each other — see the warning in §1.3. Always look them up, and always cross-check two sources.