Université de Montréal

State of Software Diversity in Ethereum

par

Nadia Gonzalez Fernandez

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en informatique

August, 2026

Université de Montréal

Faculté des arts et des sciences

Ce mémoire intitulé

State of Software Diversity in Ethereum

présenté par

Nadia Gonzalez Fernandez

a été évalué par un jury composé des personnes suivantes :

Houari Sahraoui
(président-rapporteur)

Benoit Baudry
(directeur de recherche)

Philippe Langlais
(membre du jury)

Résumé

La diversité logicielle contribue à améliorer la fiabilité et la sécurité des systèmes depuis les années 1960, notamment grâce à des modèles dans lesquels une autorité centrale mandate et certifie des implémentations indépendantes.

Au cours de la dernière décennie, Ethereum a déployé la diversité à grande échelle sous une forme nouvelle : 11 clients indépendants y coexistent sans être mandatés ni certifiés par une autorité centrale, tout en bénéficiant de la diversité déjà présente dans l'écosystème open source.

Cette thèse étudie Ethereum comme un exemple de ce modèle émergent de diversité logicielle, en examinant son adoption, son étendue dans le code des clients et les défis qu'elle soulève. Nous combinons des entretiens semi-directifs avec sept contributeurs impliqués dans le développement, la coordination et le suivi de la diversité des clients, avec la première analyse statique complète de la chaîne d'approvisionnement logicielle des 11 clients les plus utilisés.

Deux principaux résultats se dégagent. Premièrement, la communauté Ethereum veille activement au maintien de cette diversité au moyen d'une spécification commune, d'un suivi continu de la distribution des clients, de financements dédiés et d'interventions visant à limiter leur concentration. Deuxièmement, notre analyse montre que cette diversité s'étend au code : huit moteurs de bases de données indépendants sont utilisés par les 11 clients, tandis que les équipes convergent volontairement vers les composants les plus difficiles à implémenter correctement et privilégient leur audit plutôt que la multiplication d'alternatives non vérifiées.

Ainsi, Ethereum constitue un exemple de diversité logicielle étendue et activement entretenue par sa communauté.

Mots-clés: diversité logicielle, Ethereum, chaîne de blocs, programmation N-version, chaîne d'approvisionnement logicielle, systèmes décentralisés, code source ouvert, tolérance aux pannes

Abstract

Software diversity has improved reliability and security in systems since the 1960s, through models in which a central authority commissions and certifies independent implementations. Over the last decade, Ethereum has deployed diversity at scale in a new form: 11 independent clients coexist without any central authority commissioning or certifying them, while also leveraging the diversity already present in the open-source ecosystem. This thesis studies Ethereum as a case of this emerging software diversity model, asking how such diversity is adopted, how far it reaches into client code, and what challenges it introduces. We combine semi-structured interviews with seven contributors who build, coordinate, and monitor client diversity with the first comprehensive static analysis of the software supply chain of the 11 most widely used clients.

Two results follow. First, the Ethereum community actively cares for and sustains its client diversity: interviewees describe coordination through a shared specification, continuous monitoring of client distribution, dedicated funding, and intervention against client concentration. For example, when Prysm exceeded 60% of the consensus layer, the community organized a campaign to redistribute validators away from it. Second, our supply chain analysis confirms that this diversity extends into the codebase: eight independent database engines back the 11 clients, and teams deliberately converge on the components hardest to implement correctly, funding their audit rather than multiplying unverified alternatives. Ethereum exhibits extensive software diversity that its community actively cares for and maintains. This sets an example of a new model for decentralized software diversity.

Keywords: software diversity, Ethereum, blockchain, N-version programming, software supply chain, decentralized systems, open source, fault tolerance

Contents

List of tables

List of figures

Glossary

Beacon Chain	Ethereum's proof of stake consensus layer coordinating validators.
Block	Collection of transactions and other data that is added to the Ethereum blockchain and linked to the preceding block.
Client Diversity	Coexistence of multiple independent client implementations of the Ethereum protocol.
Consensus Layer	Layer of the Ethereum protocol responsible for handling block proposal, attestation, and finality.
ETH	The native cryptocurrency of the Ethereum blockchain, used to pay transaction fees and to participate in proof-of-stake consensus.
Ethereum Improvement Proposal (EIP)	Describes standards for the Ethereum platform, including core protocol specifications, client APIs, and contract standards.
Ethereum Virtual Machine (EVM)	Runtime environment that executes smart contract bytecode.

Execution Layer	Component handling transaction execution and state management.
Finality	Guarantee that a confirmed block cannot be reverted without a large stake penalty.
Hard Fork	Protocol change that makes previously valid blocks or transactions invalid unless nodes upgrade to the new rules.
Mainnet	The primary public, production Ethereum network.
Peer-to-Peer (P2P)	Network model where nodes communicate directly without a central server.
Proof of Stake (PoS)	Consensus mechanism where validators stake ETH to propose and attest blocks.
Proof of Work (PoW)	Consensus mechanism where miners compete by solving computational puzzles.
Smart Contract	Program that executes on the EVM, can store data and interact with other smart contracts.
Slashing	Penalty that destroys part of a validator's stake for malicious or faulty behavior.

Software Bill of Materials (SBOM)	Structured inventory of the software components, libraries, and dependencies included in a software system.
Stake	The amount of ETH a validator deposits and locks in Ethereum to participate in proof-of-stake consensus and earn rewards.
State	The current set of data of the blockchain, including the block number, the hash of the previous block, and the transactions included in the current block.
Validator	Network participant staking ETH to propose and attest blocks.

Acknowledgements

To Laila, Mario and Leo, who make Montreal feel more like home. To Mom and Dad, for always supporting me, even from afar.

To Yogya, who started this journey with me and always has a helping hand. To Roxana, Madjda, and Alireza, for making the lab a fun place to work in.

To Amalia, Ernesto, Andrea, Daniela and Jose, for becoming my second family in Montreal.

To my supervisor, Benoit Baudry, who has guided me with patience through this master's degree.

Introduction

Context

Modern society relies on large software systems, where failures can result in significant economic losses or even loss of human lives. Flight control computers keep aircraft in the air, signaling systems keep trains apart, and payment infrastructures settle the transactions of entire economies. In these settings, a design fault or an exploited vulnerability would result in catastrophe. In these critical systems, reliability and security are properties that have to be obtained by construction [**42**].

Software diversity was proposed as an answer to this problem in the 1970s. The idea is to build the same function several times, independently, so that the resulting versions are unlikely to fail under the same conditions. It has evolved considerably since then, from multiple versions written by hand to tolerate accidental design faults in critical systems, to variants generated automatically to resist deliberate attacks [**62, 4, 43, 49, 21, 35**].

A standard practice for software diversity is entirely manual, centrally controlled, and expensive. In aerospace, railway signaling, and nuclear instrumentation, a specification is defined and then implemented several times by separate teams, often in different programming languages and on different hardware, under a certifying authority and safety standards [**75, 69**]. This model yields genuine design diversity, but it requires conditions that are rarely available: a single owner able to commission N implementations of one specification and sufficient funding to pay N times for one function.

Another practice for software diversity is fully automated and inexpensive. Tools in the build and deployment process can generate multiple variants of a program that behave identically but differ in their internal representation. For example, randomization of memory layout at load time to prevent attacks like buffer overflows [**35**] or the insertion of non-functional or redundant code to increase program complexity [**21, 35**]. Execution environments then run several such variants at once to detect changes in behavior or tampering in their build pipeline [**23, 76, 44**]. Because no human is involved, these techniques cost

almost nothing per deployment and are applied at scale [**41**]. What they diversify is the representation of a program rather than its design.

Alongside these two models, a third form of diversity has emerged spontaneously. Open source communities implement the same functionality repeatedly, without anyone commissioning it: fifteen operating systems implement the same portable operating system interface (POSIX) system-call interface [**46**], twenty Java libraries read and write the same JSON format [**40**], and several database servers implement the same query language SQL [**36**]. Whole package ecosystems accumulate the same redundancy, as Maven Central hosts families of artifacts that provide the same functionality under different names [**70**]. Because each implementation is designed independently, guided by its own design preferences and performance goals, these variants differ in design and almost none of their faults are shared [**36**]. This diversity is spontaneous and its results are publicly available for reuse.

Research Statement

Software diversity is a core design principle of Ethereum, advocated from the start by its creator, Vitalik Buterin. Ethereum deliberately has no reference client, the protocol exists as a collaboratively maintained specification, and what users run are independent implementations of it [**12**]. Buterin defends this choice on two grounds, technical and political: no single bug in one implementation can bring down the whole network, and no single team can dictate the protocol [**12**].

In more than a decade, this ecosystem has never had any downtime, avoiding external attacks and continuously executing transactions. None of it is automated: every client is designed and written by hand. None of it is commissioned either there is no owner that pays for 11 implementations of one specification, no certifying :authority, and no standard that mandates how many implementations must exist. Ethereum is, to the authors' knowledge, the only large-scale production system in which software diversity has been achieved and maintained through purely open source development under decentralized governance.

This arrangement is also fragile: nothing obliges a client team to exist or prevents node operators from converging on whichever client is fastest or best documented at a given moment. Ethereum has nonetheless sustained 11 implementations and a healthy ecosystem.

Ethereum is therefore a case study of an emerging fourth model of software diversity: deliberate, manual, and sustained without a central authority. The objective of this thesis is to identify the conditions under which diversity is introduced and sustained in a system that has no central authority to impose it, and we measure how far the resulting diversity reaches into the code.

Novelty Statement

Research on Ethereum is abundant, but few works study software diversity in the blockchain. Ron et al. design and implement an N-version node that runs multiple clients in parallel, so that a bug in one implementation does not bring the node down [**67**]. In a later work, the same authors propose a framework that economically rewards operators who run a minority client, promoting a healthier and more diverse network [**66**]. Ranjan et al. survey Ethereum node operators and measure deployment diversity, analyzing the distribution of client usage, operator preferences, and attitudes toward client diversity across the ecosystem [**63**]. Closer to the code, Soto-Valero et al. analyze the software supply chain of the two Java clients, Besu and Teku, and find that they share more than half of their dependencies [**71**].

Existing research documents aspects of Ethereum's client ecosystem [**67, 66, 63, 71**], but none examines how the community arrived at 11 independently developed implementations or what sustains them despite the costs of maintaining protocol compatibility. The motivations behind creating a new client, selecting a programming language, and continuously tracking protocol upgrades remain largely undocumented. To the authors' knowledge, no study spans all 11 clients across their six software ecosystems, nor investigates the software supply chains and storage engines on which they depend, even though such shared components are a vulnerability for the entire network. This thesis addresses this gap along two dimensions that have not been studied together: the practices through which the community deliberately introduces and maintains it and the extent to which the diversity claimed at the level of clients holds inside their codebases.

Research Method

We study software diversity in Ethereum from two complementary perspectives: qualitative and quantitative evidence [**24**]. From the perspective of the people who build the clients, the question is why independent teams and organizations keep investing in separate implementations and how they reconcile building a robust product with preserving diversity in the network. From the perspective of the code, the question is how implementations remain different beyond their programming language.

In chapter 3, we conduct a qualitative assessment of how the community introduces and maintains client diversity. We run semi-structured interviews with 7 participants, covering execution and consensus client developers, a governance and Ethereum Improvement Proposals (EIPs) coordinator, a validator operator, and two client diversity analysts, all with three to ten years of experience in the ecosystem. With the information collected, we analyze the adoption, benefits, and challenges of software diversity in Ethereum.

In chapter 4, we perform a quantitative assessment of how independent the 11 clients are. Six programming languages guarantee that the clients differ in their source code, but they say nothing about what each client reuses from elsewhere. We therefore measure convergence in two aspects of the codebases that language barrier doesn't prevent them from sharing. Since no canonical blueprint prescribes how an Ethereum client must be built, we select these two aspects from the documentation and our knowledge of the domain, because they are critical to node operation and are likely sites of unintended convergence: third-party dependencies and database backend.

Key Takeaways

This thesis provides the first combined qualitative and quantitative account of software diversity in Ethereum, identifying the conditions under which independent implementations emerge and persist without a central authority and assessing how far that diversity reaches into the code.

Our interviews show that the Ethereum community actively maintains software diversity. Client developers implement from a shared specification rather than from one another's code and coordinate upgrades in public all-core-devs meetings. The community monitors the client distribution across the network and intervenes when one client approaches a dangerous share, as it did when Prysm exceeded 60% of the consensus layer. Dedicated funding from the Ethereum Foundation and public-goods programs sustains minority client teams that would otherwise lack the resources to keep up with protocol upgrades.

Our software supply chain analysis shows that this diversity extends beyond language choice. The 11 clients reuse a vast open-source supply chain that was not written for blockchain: eight embedded database engines and thousands of third-party libraries across six language ecosystems. Cross-ecosystem overlap is limited, confirming that language choice is the most effective source of supply chain diversity. Within a language, overlap is higher—roughly 14% of Go modules, 27% of Rust crates, and 59% of Java artifacts—yet even there active divergence is possible, as Erigon and Go Ethereum demonstrate. Part of the remaining convergence is deliberate: the ecosystem knowingly shares cryptographic primitives such as `blst` and KZG, and funds their audit rather than multiplying unverified alternatives.

Together, these results describe what we call *decentralized software diversity*: a fourth model that takes the independent implementations of commissioned software diversity and the open-source reuse of natural software diversity, and sustains both without a central authority.

All resources of this study are publicly available in our GitHub repository[1]. This includes the interview questions, the codebook and themes derived from the interviews. It also contains the software supply chain analysis scripts, and the extracted dependency and database data. Every result we report can be reproduced.

[1] https://github.com/nala7/eth_client_diversity

Chapter 1

Background and State of the Art

In the following chapter, we present the background and state of the art of the thesis. We start by defining software diversity as a technique for reliability and security. We then present the Ethereum blockchain, how a node is built, how the network reaches agreement, how the protocol is specified and changed, and what a client defect can cost. Finally, we show how Ethereum implements software diversity as client diversity.

1.1. Software Diversity

Software diversity is the use of several functionally equivalent versions of a program, obtained through different designs, implementations, or development processes. The property that makes it valuable is failure independence: versions built independently are unlikely to fail under the same conditions or to expose the same vulnerabilities [**5, 35**].

Diversity is pursued for two distinct reasons. The first is reliability. Replicating a program on redundant hardware protects against physical faults [**62, 49**] but not against design faults, because every replica contains the same bug. Running several independently written versions and comparing their results, through a vote or an acceptance test, allows an incorrect result produced by one version to be detected and masked by the others [**4**]. The second reason is security. An exploit is written against the specific memory layout, interfaces, and behavior of one implementation, so it does not necessarily carry over to another. When deployed systems differ, an attacker has to repeat the work for each target [**35**]. Laprie et al. treat the two cases as facets of one problem, since diversity addresses the faults that redundancy alone cannot, whether they are accidental or deliberate [**49**].

What has changed over four decades is how diversity is engineered. Baudry et al. survey the field, separating diversity that is managed by hand, diversity that is generated automatically, and diversity that already exists and is merely exploited [**6**]. These three categories

are also three periods, and the remainder of this section follows them in order: design diversity for fault tolerance in critical systems, automated diversification for security, and natural diversity in open source.

1.1.1. Design software diversity for fault tolerance

The first form of software diversity was manual. It emerged in the 1970s, in response to the observation that redundant hardware tolerates physical faults but not design faults, and that testing alone cannot establish the reliability required by systems whose failure is unacceptable.

Randell proposed the recovery block, the first structured mechanism for tolerating design faults in software [**62**]. This method structures code into blocks equipped with an acceptance test and one or more spares (alternates). If the primary alternate fails the acceptance test or a crash occurs, the system state is reset and a spare is executed. The alternates are the same function written independently and are expected to fail on different inputs.

Avizienis formulated the complementary approach, N-version programming, in which N independently written programs execute the same specification and a decision algorithm selects the result by majority vote [**4**]. These versions run in parallel, and a voting mechanism determines the correct output based on consensus. Avizienis defines an N-version programming process that prescribes an initial specification precise about function but deliberately leaving implementation unspecified. This strategy forbids communication between the development teams, and mandatory diversity in the choices that shape a design, including algorithms, data structures, programming languages, and tools [**4, 5**].

Design diversity became standard practice in the safety-critical industries. Voges surveys its use in nuclear instrumentation, railway signaling, avionics, and aerospace, and describes the forms it takes: diverse specification languages, diverse programming languages, diverse teams, and diverse hardware [**75**]. The PODS project is representative of how such a program is organized [**10**]. Three teams in three countries independently implemented the same reactor over-power protection function, in three languages and from two different specification forms, under a controlled protocol. Back-to-back testing of the resulting versions revealed faults that had survived each team's own verification, including faults traceable to ambiguities in the specification itself. Knight later reviewed four decades of this practice and concluded that diversity works in engineering, but only under conditions that must be paid for [**42**].

Those conditions are the limitation of this model. It presupposes a single owner able to commission N implementations of one specification, a certifying authority that mandates how many implementations must exist and audits their independence, and funding sufficient

to pay N times for one function. Outside aerospace, rail, and nuclear instrumentation, these conditions are rarely available.

1.1.2. Automated software diversity for security

The second form of software diversity is automated. It emerged in the 1990s and shifted the motivation for diversity from accidental design faults to deliberate attacks. The motivating observation is the software monoculture: when millions of machines run bit-identical binaries, one exploit written once works everywhere, and the attacker's cost per target approaches zero [**41**]. Laprie et al. view accidental and deliberate faults as two manifestations of the same underlying problem, for which diversity provides a common solution [**49**]. Thus, the transition from reliability to security is not a change in principle, but a change in the source of the fault: from accidental failures to deliberate attacks.

Cohen introduced program evolution as a defense [**21**]. A program is transformed automatically into a large set of semantically equivalent variants through instruction reordering, insertion of redundant or non-functional code, and equivalent instruction substitution, so that an attack designed against one variant does not transfer to another. Forrest et al. made the argument explicit by analogy with biological populations, where genetic diversity is what prevents a single pathogen from destroying an entire species, and proposed randomizing the memory layout of programs at compile and load time so that the addresses an exploit depends on differ from one machine to the next [**35**]. Their randomization of stack frame padding is the direct ancestor of address space layout randomization, now deployed by default in every mainstream operating system as Address Space Layout Randomization (ASLR) to mitigate memory corruption exploits.

Later, moving target defense generalized the idea in time rather than in space: instead of producing one fixed variant per machine, the system continuously reconfigures its own attack surface so that the information an attacker gathers becomes stale before it can be used [**50, 17, 19, 80**].

Multi-variant execution (MVE) environments use this same principle for intrusion detection. In this setup, several diversified variants of a program run in parallel, processing identical inputs. A monitor compares their behavior to that any execution divergence immediately signals a potential attack or a corrupted build. Cabrera-Arteaga et al. developed Wasm-Mutate, a diversification engine that generates functionally identical WebAssembly binary variants to mitigate external attacks [**16**].

Because no human is involved, automated diversification costs almost nothing per deployment and is applied at internet scale, which is precisely what design diversity could never do. Its limitation is symmetric. Every variant is derived by transformation from one

source, so every variant implements the same design and inherits the same design faults. Automated diversification diversifies the representation of a program, not its design, and offers no protection against a specification error or a flawed algorithm.

1.1.3. Natural software diversity

The third form of software diversity is neither commissioned nor generated, it already exists. Baudry and Monperrus call it natural or managed diversity: the redundancy that accumulates when independent communities implement the same functionality repeatedly, and which can be exploited by whoever assembles a system out of these components [**6**]. Its cost of production is zero for the party that uses it, because someone else already paid it for their own reasons.

We observe this redundancy in practice. Koopman and DeVale tested fifteen POSIX operating systems against the same system call interface and found that each fails on a different set of exceptional inputs [**46**]. Harrand et al. analyzed twenty Java libraries that read and write the same JSON format and observed systematic behavioral differences in how they handle inputs at the boundary of the specification [**40**]. Gashi et al. studied the bug reports of four different SQL servers implementing the same query language and found that no single fault was common to all four, and that the vast majority of faults affected only one product [**36**]. Redundancy also accumulates at the scale of whole package ecosystems: Soto-Valero et al. mined Maven Central and identified large families of artifacts providing the same functionality under different names [**70**].

What makes this diversity valuable is that it is design diversity obtained from the open-source community. Each implementation was designed independently, by a different team, guided by its own performance goals and architectural preferences, and under no obligation to resemble the others. The isolation that N-version programming had to enforce by protocol is here a consequence of how open source is produced. The errors shared by equivalent implementations have little overlap, which is the property the manual model paid for and the automated model cannot provide [**36, 46**].

The consequence is a change in how a diverse system can be built. Rather than commissioning N implementations, an architect can select components that already differ, and much of the applied work surveyed by Baudry and Monperrus does exactly this: assembling detection systems from several existing antivirus engines or intrusion detection systems, running several database servers behind one interface, or deploying routers from different vendors on the same network [**6, 36**]. Diversity becomes an assembly decision rather than a development program.

1.1.4. The state-of-the-art models of software diversity

Software Diversity Model	Financial Model	Software Variants Independence	Software Diversity Sustainability
Commissioned software diversity	A single owner, N times	Enforced isolation of teams, verified by a certifying authority	Regulation and safety standards
Automated software diversity	Almost nothing per deployment	Guaranteed by construction, but limited to the same design	Build and deployment tooling
Natural software diversity	Someone else, for their own reasons	A by-product of how open source is produced	Nothing in particular; the model reuses what exists

Table 1.1. The three state-of-the-art models of software diversity.

Commissioned diversity is the classical N-version approach: a central authority funds independent teams to implement and maintain the same specification by hand. Independence is enforced and audited. The result is high reliability, but the cost is prohibitive for most systems [**62, 4, 5**].

Automated diversity is cheap because it is generated from a single program. Compilers and loaders randomize instruction encoding, stack padding, and memory layout. All variants share the same design, so they do not protect against design-level faults [**35, 50, 17**].

Natural diversity is a by-product of open source: equivalent programs already exist, written by different teams and languages, and for different reasons. An assembler can reuse them. There is no owner of that diversity, and the resulting system depends on external maintainers [**6**].

1.2. The Ethereum Blockchain

Ethereum is a public, open-source blockchain launched in July 2015 by Vitalik Buterin and a group of co-founders [**14**]. Ethereum operates on a peer-to-peer (P2P) network, where participants communicate directly without centralized servers or intermediaries [**54**]. The individual devices connected to this network are called nodes. Each node maintains the integrity of the system by storing a local copy of the shared ledger, validating incoming transactions, and broadcasting state changes to its peers [**73**].

Transactions are signed data packages containing messages broadcast across the network. These messages contain user interactions such as cryptocurrency transfers or smart contract executions [**18**]. For secure storage, these transactions are aggregated into blocks. Each block contains a bundle of validated transactions, along with a timestamp, a unique cryptographic hash that identifies the block, and the hash of the directly preceding block. [**47**].

Because thousands of independent nodes operate without a central authority, a consensus framework is required to reach a single, unified agreement on the validity of transactions and the general state of the ledger [**55**]. It is a method for selecting which participant gets to add the next block and a rule for deciding which chain is the correct one if temporary conflicts occur [**77**].

Ethereum originally used proof of work, in which participants called miners competed by solving computational puzzles, and the chain with the most accumulated work was taken as canonical [**78**]. In September 2022, in an upgrade known as *The Merge*, this mechanism was replaced by proof of stake [**1**]. The chain that implements it is called the Beacon chain, and it is run in the consensus clients [**33, 68**].

Under proof of stake, the right to participate in consensus is bought with a deposit. A participant locks 32 ETH into the protocol and thereby registers a validator: an identity that is expected to be online and to sign messages according to the rules [**39**]. Validators are operated by consensus clients and they are the ones responsible for proposing, voting and agreeing to the creation of new blocks.

1.2.1. Node architecture

An Ethereum node is composed of two main components, as shown in Figure 1.1, which run side by side and communicate locally: an execution client and a consensus client [**74**]. The two clients are typically written by separate teams in different languages, and the protocol does not tie them together: any execution client can be paired with any consensus client to form a node.

The consensus client is responsible for the chain's consensus logic: it runs Ethereum's proof-of-stake protocol, which decides which blocks belong to the chain and in what order. It receives new blocks from its peers through block gossip and hands the block's transactions to the execution client for validation through the Engine API. Once it obtains the result from the execution client, it either attests to the block or, when selected to do so, proposes a new block of its own [**15, 39**].

The execution client validates and executes the transactions in a block, then returns the execution result to the consensus client. To validate each transaction, it verifies the sender's signature and, if the transaction invokes a smart contract, executes the contract's bytecode

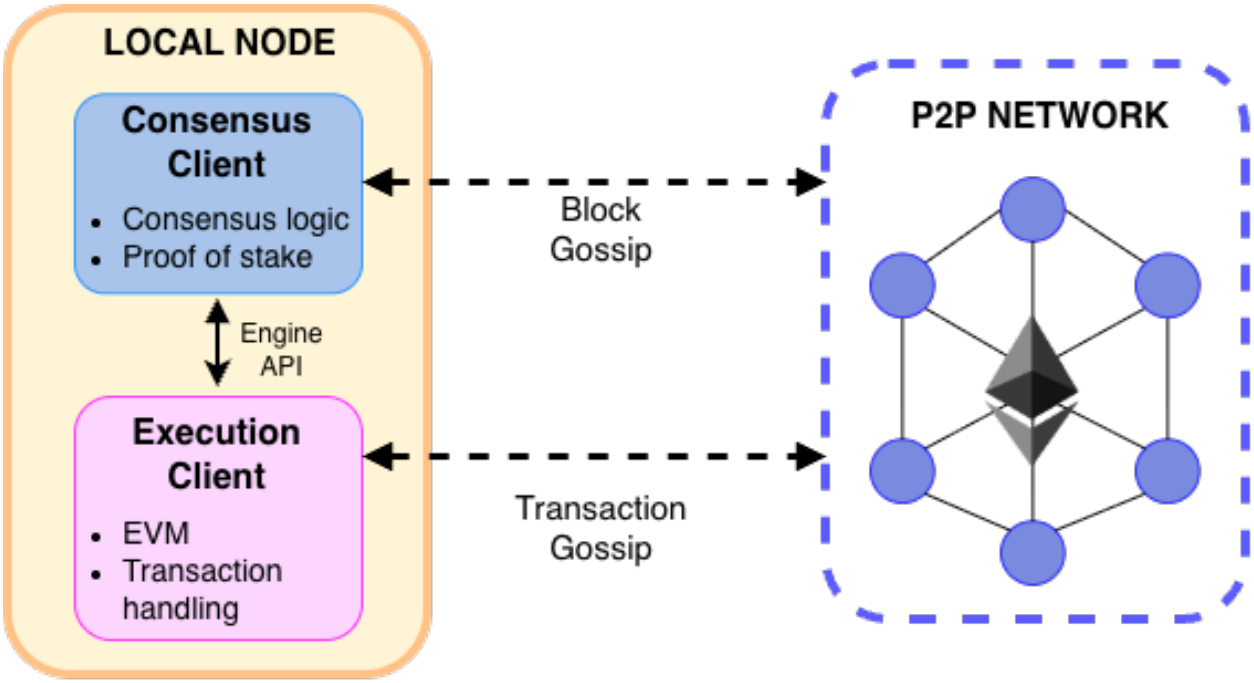

Fig. 1.1. A local Ethereum node is composed of both a consensus client and an execution client that communicate internally through the Engine API. Each client also maintains its own connection to the broader peer-to-peer network: the consensus client participates in block gossip to propagate and receive new blocks, while the execution client engages in transaction gossip to relay pending transactions.

in the Ethereum Virtual Machine (EVM), updating the state accordingly. The execution client also keeps a pool of transactions that have been submitted but not yet included in a block, and relays them to its peers through transaction gossip [**78, 39**].

1.2.2. Protocol evolution and coordination

Ethereum has no reference implementation whose behavior defines the protocol, instead it has a set of documents maintained by the community. The White Paper of 2013 described the vision and the Yellow Paper of 2014 gave the first formal definition of the execution layer [**14, 78**], but neither is used as reference today. The active references are the Ethereum Improvement Proposals (EIPs), which are numbered documents that each describe one change to the protocol or one standard, and two specification repositories: the execution specifications, an executable reference description of transaction execution and the EVM, and the consensus specifications, which define the Beacon chain and validator behavior [**3, 33**]. Alongside them, the community maintains large suites of conformance tests that every client is expected to pass. Table 1.2 summarizes these sources and their authority.

Source	Authority	Description
EIPs	Ethereum community (EIP editors)	Ethereum Improvement Proposals define protocol changes, standards (e.g., ERCs), and APIs. Primary mechanism through which Ethereum evolves over time.
Execution Specs	Client teams and Ethereum Foundation	Readable and up-to-date reference implementation of the execution layer.
Consensus Specs	Client teams and Ethereum Foundation	Formal specification of Ethereum 's proof of stake consensus (Beacon chain), including fork upgrades and validator behavior.

Table 1.2. Main sources of Ethereum specifications and their roles in defining the protocol.

Changes reach the network in batches called network upgrades, or hard forks. An upgrade bundles a set of accepted EIPs and activates them at a predetermined point in time, identical for everyone [**8**]. Every client team must therefore implement the same changes and ship a release before that point; a node that keeps running the old software will reject the blocks that the rest of the network produces and incur penalties.

Because an upgrade cannot be undone, it is rehearsed before it reaches the live network. The live network is called Mainnet, and it is the one that holds real value. Around it the community runs Testnets, which are separate networks that follow the same protocol with valueless funds used by client teams to test a specific upgrade against each other. Upgrades are activated on devnets and testnets first, so that a defect is discovered where nothing is lost.

Coordinating all of this is a public process rather than an authority. The main venue is the all-core-devs (ACD) calls, recurring open meetings in which client teams, researchers, and EIP editors decide which proposals go into the next upgrade and confirm that every client is ready [**8**]. Discussion also happens on public forums, and anyone may attend or propose. The Ethereum Foundation, a non-profit that funds research and development in the ecosystem, coordinates much of this work and maintains the specifications, but it cannot compel anyone: a change takes effect only when client teams implement it and node operators choose to run the result.

1.2.3. Finality and slashing penalties

When a block has been attested by at least two-thirds of validators, it becomes *finalized*: the protocol will never reorganize the chain behind it, so a finalized transaction is permanent and irreversible [**57**]. This two-thirds threshold creates two distinct risk scenarios, depending on how much of the validator set is compromised at once.

If validators holding more than one-third of the stake go offline or misbehave, the rest of the network can no longer gather the two-thirds majority that finality requires. The chain keeps producing blocks, but stops finalizing them, and recent transactions remain reversible until the situation is resolved. If, instead, validators holding more than two-thirds of the stake share a defect that makes them accept an invalid block, they finalize it — and the protocol offers no way back, since recovering would mean overriding the very rule that made the block permanent in the first place.

Ethereum discourages both scenarios through two economic penalties. A validator that proposes and attests as expected earns a reward. However, a validator that is offline loses a comparable amount for every duty it misses, and this penalty grows with the fraction of the network offline at the same time [**39**]. Moreover, validators must never support two conflicting versions of the chain — for example, signing two different blocks for the same slot. A validator that does so can be *slashed*: permanently removed from the validator set and stripped of part of its stake. Slashing penalties also scale with how many validators are slashed at once. As a result, a bug that makes many validators double-sign simultaneously is punished as severely as a coordinated attack, since the protocol cannot tell the two apart.

Both mechanisms make it far more costly for many validators to fail in the same way at the same time than to fail in isolation. This is precisely why Ethereum promotes *client diversity*: since a defect in one client implementation can affect every validator running it, no single client should ever be run by more than one-third of the stake — otherwise a bug or exploit in that client could halt finality — and none should approach two-thirds, which could let an invalid chain be finalized. Client diversity is therefore treated as a security parameter in its own right.

1.3. Software Diversity implemented in Ethereum

Software diversity in Ethereum takes one specific form: client diversity. Ethereum has multiple clients that implement the same protocol but with different internal designs, programming languages, and optimizations, developed by different teams. This enables client

diversity while preserving interoperability [**78, 37, 31, 32**]. Having multiple client implementations allows for a healthy level of competition and encourages innovation among client developers [**63**].

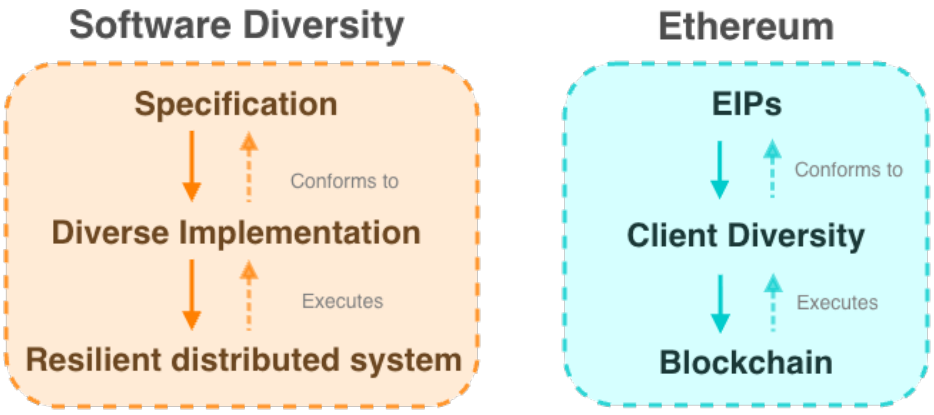

Fig. 1.2. Software Diversity has a layered structure. Starts with a formal specification, followed by multiple independent implementations that are then deployed in a distributed system. Ethereum has the EIPs that defines the protocol and enables multiple independent client implementations. These diverse clients later interoperate in the blockchain. As a result, diversity at the implementation level translates into a fault-tolerant and resilient distributed blockchain network.

As illustrated in Figure 1.2, software diversity originates from a common specification and propagates into multiple implementations that are later deployed across the distributed system.

The specification is intentionally precise in defining the protocol's expected behavior, yet it leaves a number of implementation decisions open to developers. For example, while the specification assumes that an implementation maintains the mapping between addresses and account states using a modified Merkle Patricia trie, it does not describe how this structure should be stored or managed internally. The choice of database is left as an implementation detail.

Similarly, the specification does not mandate a programming language, concurrency model, or how developers should optimize performance or ensure efficient data access. Instead, it outlines what must be achieved while leaving flexibility in how it is achieved. This balance between precision and freedom is what enables meaningful software diversity.

Chapter 2

Empirical Analysis of Diversity Engineering In Ethereum

This thesis is the first empirical study of the engineering of software diversity in Ethereum. In this chapter, we first present the overall empirical design: a mixed-method case study that examines the same phenomenon from two perspectives, drawing on both the people who develop and maintain clients and the code they produce. We then introduce the 11 execution and consensus clients included in both studies.

2.1. Thesis Methodology

This thesis studies the Ethereum blockchain as a case study for software diversity without a central authority. The aim is a general characterization of the state of diversity: how independent implementations are introduced and maintained, and how far that diversity reaches into the code. A thoroughly executed case can contribute to scientific development; one can generalize from a single case, and the force of example is often underestimated relative to formal sampling [**34**].

Ethereum is, to the authors' best knowledge, the only large-scale production system in which software diversity has been achieved and maintained through open source development under decentralized governance. Diversity in this setting is both a community practice and a property of artifacts, so neither interviews nor repository analysis alone is sufficient. Field studies in software engineering collect data either directly from people or independently from work artifacts such as source code and documentation [**27**]. We therefore study the

same phenomenon along both routes. Semi-structured interviews recover intent, coordination, funding, and trade-offs that never appear in repositories. Code analysis of the client implementations measures independence.[1]

Figure 2.1 summarizes this pipeline. What already exists in Ethereum is shown in blue: the community around the clients and the artifacts they produce. What this thesis collects is shown in orange. In chapter 3, we report on semi-structured interviews, through which we collect first-hand knowledge about the adoption, benefits, and challenges of software diversity. In chapter 4, we extract dependency trees, software supply-chain overlap, and the inventory of storage engines, to quantitatively assess the level of diversity beyond different programming languages. The two strands combine into an analysis of the state of software diversity in Ethereum.

The 11 clients that are used in both studies are listed in the next section.

2.2. Study Subjects

	Repository	Commit	Language	Ecosystem	Commits	Since	Contribs.	LOC
Execution	Besu	`983c53f`	Java	JVM	6974	10/2018	294	734k
	Erigon	`7e8c1ef`	Go	Go modules	31428	12/2013	992	1.88M
	Go Ethereum	`36a7dc7`	Go	Go modules	17133	12/2013	1289	741k
	Nethermind	`9c36577`	C#	.NET	10989	08/2017	233	548k
	Reth	`9384bc5`	Rust	Cargo	14597	09/2022	772	450k
Consensus	Grandine	`70c81c5`	Rust	Cargo	766	03/2024	16	160k
	Lighthouse	`120c3c6`	Rust	Cargo	7535	07/2018	123	306k
	Lodestar	`5cb87b7`	TypeScript	Node.js	11368	06/2018	249	307k
	Nimbus Eth2	`2361ee1`	Nim	Nimble	8598	07/2018	226	184k
	Prysm	`5498c83`	Go	Go modules	10480	07/2015	312	875k
	Teku	`6c8bd12`	Java	JVM	6812	09/2018	128	605k

Table 2.1. Most popular execution and consensus clients in the Ethereum network as of October 2025

[1] 

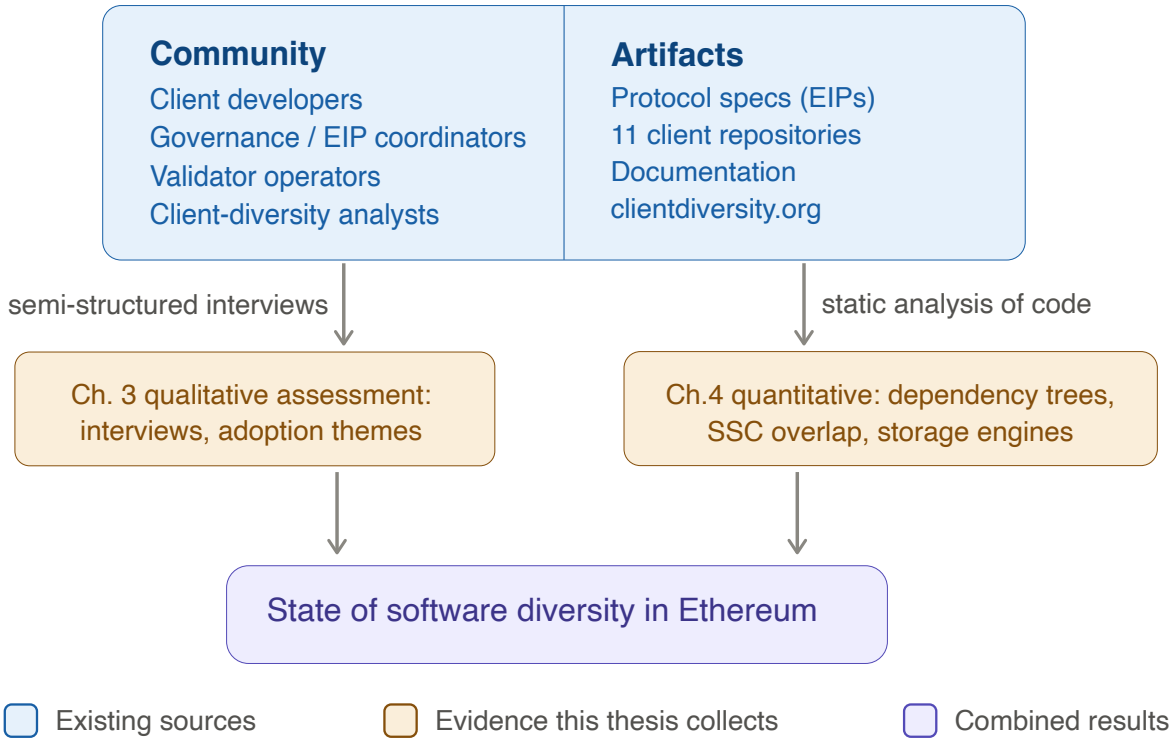

Fig. 2.1. Methodology pipeline for the state of software diversity in Ethereum. Blue boxes are sources that already exist in the ecosystem. Orange boxes are the evidence this thesis collects: interview themes in chapter 3 and repository measurements in chapter 4. The two strands combine into the results presented in this thesis.

Table 2.1 lists, in alphabetical order, the 11 most popular execution and consensus clients in the Ethereum network as of October 2025. The table reflects a mature and broad ecosystem across several dimensions. The clients are implemented in a wide range of programming languages and ecosystems, including Java (Besu, Teku), Go (Erigon, Go Ethereum, Prysm), C# (Nethermind), Rust (Grandine, Lighthouse, Reth), TypeScript (Lodestar), and Nim (Nimbus Eth2), so that no single language dominates.

Most projects have long and active development histories: the oldest repository is Go Ethereum, which dates back to 201,3 and the majority have been under continuous development for roughly 7 to 12 years, with only the newest client (Grandine, 2024) being a recent addition. Erigon is the only client that started as a fork from another, in this case Go Ethereum. Their last commit[2] in common was in October 2019. However, since then, Erigon has drifted away from Go Ethereum, focusing on disk space and sync speed optimization.

[2]https://github.com/erigontech/erigon/commit/bd05968077f27f7eb083404dd8448157996a8788

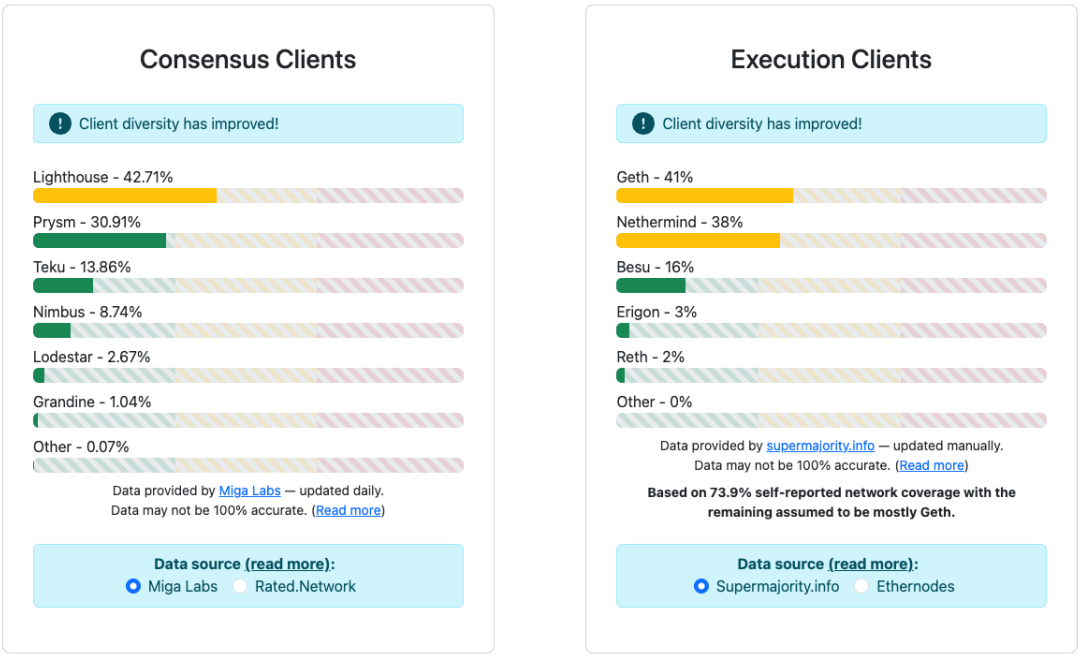

Fig. 2.2. State of client diversity on October 1st, 2025 as per `https://clientdiversity.org`.

The repositories have accumulated substantial engineering effort, ranging from thousands to tens of thousands of commits, Erigon being the project with the most commits (31,428). This is also backed by a large and diverse contributor base, with several clients counting hundreds, led by Go Ethereum with almost 1,300 contributors, consistent with this being the oldest Ethereum client.

The scale of these codebases further underlines how large full client nodes are: even the smallest client in the set exceeds 160,000 lines of code, most implementations range from roughly 300,000 to 875,000 lines, and Erigon alone approaches 1.9 million.

Together, these figures show that Ethereum is supported by multiple independent, actively maintained client implementations rather than a single reference codebase. In this thesis we study how the community sustains this diversity and how diversity permeates through the different layers of these various implementations.

Chapter 3

Qualitative Assessment of Software Diversity by the Ethereum Community

In this chapter, we present a qualitative assessment of the community building and maintaining software diversity in Ethereum. Client diversity is not a permanent property. In a distributed system with no central authority, it is the community, developers implementing independent clients and operators choosing among them, that carries the practice forward [**63**].

We conduct semi-structured interviews with client developers, governance coordinators, validator operators, and client diversity analysts to examine how diversity is introduced and coordinated. We also explore the community's perception on the benefits software diversity provides and its challenges.

The motivations, coordination mechanisms, and trade-offs behind client diversity cannot be studied in source code or documentation. They live in the intentions and decisions of the people who sustain the ecosystem, and are otherwise scattered across social media and informal channels. Interviews are therefore well suited to recover this first-hand, tacit knowledge that no artifact-based analysis could surface on its own [**27**].

3.1. Methodology

To assess the state of software diversity in Ethereum, we select the most popular clients on the Ethereum blockchain as a starting point as shown in Figure 2.2. We then conduct semi-structured interviews with contributors of the projects and the community. In this section, we describe the methodology we follow to select the projects, contact contributors, interview them, and analyze their responses.

Participant selection: We aim to conduct interviews with people in the Ethereum ecosystem, both governance and coordination contributors as well as client developers, who work

towards improving and maintaining software diversity in Ethereum. To select projects, we rely on the data of `clientdiversity.org` (Figure 2.2) as reference for the most popular client implementations in Ethereum. From this we obtain 11 open source projects as in Table 2.1.

For each selected project, we search the public GitHub profile of the 2 top contributors and send an interview invitation. As main contributors of a complex and mature project, we expect these developers to be knowledgeable about Ethereum's protocol implementation and to be familiar with software diversity practices. When developers respond positively to the initial email, we send them the outline of the interview questions in advance so they can confirm their familiarity with the topics. If the developer's contact information is not publicly available or we do not receive a response within a week, we continue iterating through the top contributors.

We ask participants who respond to recommend others who could contribute to our study, and we invite them to participate.

Participant demographics: This snowball effect led to 33 invitations in total and 7 positive responses: two execution and one consensus client developers, a governance and EIP coordinator, a validator operator, and two client diversity analysts. This diversity of roles ensures multiple perspectives on the same phenomenon. All participants had prior professional software development experience before entering the Ethereum ecosystem. They also have between 3 and 10 years of experience working on Ethereum-related projects. More information on each participant is summarized in Table 3.1.

Interviews: We conduct 30 minute online interviews with 7 Ethereum contributors. We record the interviews only upon consent from the participants. One interview was conducted without recording at the participant's request, only written notes were taken. Participants were not provided any compensation for their participation.

The main interview questions are designed to address how is software diversity introduced and maintained in Ethereum and the benefits and challenges it introduces. Most questions are open-ended, encouraging a natural conversation and giving participants the opportunity to share experiences beyond direct responses.

The first questions are about background information on when and why the participant started working on blockchain, as well as their current role and responsibilities. The second part addresses Ethereum client diversity, covering awareness of software diversity, motivation for creating a new client, programming language choice, and use of specifications. The interviewees were also asked about the challenges of maintaining diversity, business sustainability, and strategies to encouraging adoption while preserving diversity.

The full list of questions can be found on Github [1].

Interview response analysis: We use the Whisper speech recognition tool (Radford et al. 2023[2] [**61**]) to transcribe 6 interviews audio recordings. If audio recordings are not available, we refer to the written notes. Two authors conduct line-by-line coding of the transcribed interviews or written notes. The same two authors then collaboratively create an affinity diagram, grouping the codes into emerging themes inductively. In cases of disagreement, the two authors consult a third author to help resolve the conflicts. This process resulted in 30 sub-themes and three main themes: adoption, benefits and challenges. The final codebook is available in GitHub[3].

ID	**Experience in Ethereum (Years)**	**Job Title**	**Job Description**
P1	5–10	Ethereum client developer [Execution]	Execution protocol developer; mathematics academic researcher; designer of EIPs
P2	3–5	Ethereum client developer [Execution]	Execution client developer; blockchain engineering
P3	3–5	Ethereum client developer [Consensus]	Consensus client developer; protocol engineer
P4	5–10	Ethereum governance; EIPs coordinator	Ethereum governance and enterprise alignment; ETHSecurity promoter of client diversity in community; manages coordination between different sectors of the community
P5	3–5	Ethereum validator operator	Developer of multinode validator clients
P6	–	Client diversity analyst	Develops and maintains a website that tracks live info on client diversity
P7	5–10	Client diversity analyst	Developer of crawler that monitors client diversity in the Ethereum network

Table 3.1. Overview of interview participants.

[1] `https://github.com/nala7/eth_client_diversity/blob/main/State%20of%20Diversity%20in%20Ethereum%20Interview%20Questions.pdf`

[2] https://proceedings.mlr.press/v202/radford23a.html

[3] https://github.com/nala7/eth$_c$*lient*$_d$*iversity/blob/main/codebook.xlsx*

3.2. Community's Deliberate Adoption of Software Diversity in Ethereum

In this section, we examine how the current network was created using software diversity through coordination and shared specifications. We analyze how the ecosystem actively maintains client diversity through deliberate emergence of client diversity, coordination, independent implementations, monitoring, intervention against client concentration, sustaining minorities and economic incentives.

3.2.1. The deliberate emergence of client diversity

Client diversity in Ethereum did not arise organically from market competition or independent initiative. It was set into motion through an explicit call from the Ethereum Foundation to build independent implementations of the protocol.

For example, for the creation of the Beacon Chain, the Ethereum Foundation had a hands on role in the development of the new consensus clients: *"They made this call for researchers and developers. So, like, we want to develop a new version of Ethereum, who is interested, who wants to collaborate, who wants to contribute. And, of course, there were brands behind"* (P7). In response, community members volunteered to implement clients.

The Foundation is able to motivate teams implementing clients in different languages and platforms through economic incentives. When a team makes a proposal for a new client, if there is already another client using the same programming language they will suggest alternatives. *"We already have one, maybe we're not interested, not having a second one. Are you interested in checking another language or something different, because then it's just too easy to make a fork of the open source client". "So you are making sure that you want you get the diversity that you are willing to pay"* (P7).

This effort for diversity shapes the development of Ethereum. Teams that enter the ecosystem after multiple clients already existed describe diversity itself as an explicit motivation for programming language choice. Client diversity is thus not an incidental outcome of decentralized development but a deliberately initiated and continuously reinforced design choice.

3.2.2. Coordination through specifications and shared governance

Once multiple independent clients exist, the ecosystem requires coordination mechanisms to ensure they remain interoperable without converging on a single implementation. This role is served by Ethereum Improvement Proposals (EIPs), which define the shared standard that every client must implement. As P4 explains: *"the set of standards, which is defined in*

our EIPs website that each client will take as their common code base. Like this is what is the referral code. This is what you have to implement in your client" (P4).

Network upgrades require simultaneous implementation across all clients: *"we need to select EIPs as well as make sure that these are implemented in all the clients at the time and then get it deployed on the mainnet at one day and time"* (P4).

The original Yellow Paper, once the foundational reference for the protocol, has been replaced by specifications that evolve with each upgrade. As one participant describes: *"We don't take yellow paper as reference anymore. We take the present code base as reference, add new proposal and go back and update yellow paper as needed"* (P4). The active references are now the consensus and execution specifications on GitHub and backed by an extensive test suite: *"30,000 tests or something or even more"* (P3) that clients run to verify correctness against the specification.

Specification development itself is bidirectional rather than top-down — *"sometimes it's first on the spec. Sometimes there's a proof of concept in a client and then brought to the spec [...] the main maintainers from the Ethereum Foundation works on the spec, but there are multiple other maintainers and sometimes even client developers do a PR and change the spec"* (P3).

Day-to-day coordination across client teams happens through all-core-devs (ACD) meetings: *"we have one common meeting called all core dev meetings in which all client teams join in. And we discussed the main spec"* (P4). Proposals are also discussed on public forums to gather broader community input: *"every individual or stakeholder who wants to use Ethereum, they propose their ideas. Their ideas are shared in the ACD meeting [...] and also shared on public forum to collect community response"* (P4).

3.2.3. Independent Implementations

Client teams deliberately avoid copying each other's implementations, even when working across different programming languages. *"Client teams do exchange information among each other but take care not to copy each other's implementations. Copying implementations, even in a different programming language, would defeat the purpose of diversity"* (P5). This ensures that coordination does not compromise implementation independence, which is essential for diversity.

3.2.4. Building infrastructure to monitor client distribution

Maintaining diversity requires knowing the current distribution of clients across the network. This information was initially unavailable: *"nobody had any idea about how many nodes were running this client [...] We had no way to know"* (P7). Without visibility into

the client distribution, the ecosystem cannot assess whether a dangerous concentration is forming.

To address this, the Ethereum Foundation commissioned the development of a network crawler: *"the foundation asks, can anyone develop a crawler? A crawler [...] connects with as many nodes as possible [...] and gets information about what client they are using"* (P6). The data that this tool produces is the foundation for ongoing education and awareness campaigns: *"they were relying very heavily on the data that we published, open for free to the community [...] to educate everyone in the community and explain the client diversity is extremely important"* (P6). A dedicated website was later created to keep this information publicly accessible and up to date.

3.2.5. Intervening against client concentration

When monitoring reveals dangerous concentration, the ecosystem activates crisis-response mechanisms. A well known example is when Prysm consensus client controlled a supermajority of the network: *"Prysm had over 60% of the network, actually way more. And we could even finalize an invalid chain. So there was an active campaign to drive users outside of Prysm"* (P1).

These interventions operate through multiple channels simultaneously. The Ethereum Foundation adjusts its official guidance: *"the Ethereum Foundation website said explicitly when you were spinning up a validator, please choose a different client"* (P1). Community members organize social media campaigns: *"there was a group of people in the community that will go on Twitter and spam everyone saying like guys, we need to change. People please change from this client to another client"* (P7). Others focus on direct education: *"just educating people explaining why this is important, and when we were approaching the limit. Then, just go out and say to everyone, please change. Because this is dangerous"* (P7).

The primary lever, however, is pressure on staking pools and large infrastructure operators, who control significant portions of the network's validator set: *"mostly it was ETH staker [...] gathering most stakers and staking pools and companies. And there was a strong drive to force pools to move away from Prysm"* (P1). Interviewees describe this mechanism as reliable and repeatable: *"If we get to a critical situation, it's not so hard to pressure, as we did publicly, pools to try to force client diversity"* (P1).

Minority client teams also engage in promotional efforts beyond crisis periods. These include sharing performance metrics from mainnet validators and engaging with the community: *"Go to conferences, talk to people there, have telegram chats, ask them why they are not running the client"* (P3).

Underlying all of these efforts is a shared understanding of the risks that motivate diversity: *"I think their main motivation is their awareness of the risks associated with client bugs"* (P5).

3.2.6. Funding and tooling that sustain minority clients

Diversity cannot persist without material support for the teams that build and maintain minority clients. The Ethereum Foundation provides initial grants to new client projects: *"if there is a new client which is trying to come up in the ecosystem [...] Ethereum Foundation also sometimes support them by providing some grants"* (P4), and has formalized this through a dedicated Client Incentive Program.

Client teams also receive economical incentives to start up: *"we are also getting public good funding from the ecosystem in order for us to support the network upgrades"* (P4). However, this funding is structured as seed investment rather than ongoing subsidy: *"this is an investment for one client to grow them. And now it is the responsibility of the client to make them profitable on their own"* (P4).

Beyond funding, new tools are developed to help validators benefit and contribute to diversity. Multi-client validator tools apply the N-version programming principle at the level of an individual validator: rather than trusting a single implementation, the tool runs several independent clients in parallel and only signs a message when they agree, so that a bug in any one client is caught by the others instead of causing a faulty attestation [**67**].

Such tools have been created: *"I was looking for a simple way to protect validators from client bugs, and that did not exist". "I implemented [it] to protect validators from the kind of dangerous client bugs that can lead to the loss of significant amounts of staked ETH"* (P5).

Multi-node infrastructure options also provide additional mechanisms for distributing client risk across implementations. Tools like Vero[4] and Vouch[5] facilitate individual validators adopting minority clients and reduce the costs of maintaining a diverse setup.

3.2.7. Economic incentives for diversity

The Ethereum protocol's penalty structure creates a direct financial incentive for validators to avoid majority clients: *"validators are penalized for downtime — if their chosen client has a bug and connected validators go offline, they get financially penalized until they come back online again. The more of the network is offline, the harsher the penalties"* (P5). This means that running the same client as the majority exposes validators to correlated risk: *"if people are following client with 70 plus percent of adoption, then if that client goes*

[4]https://github.com/serenita-org/vero

[5]https://github.com/attestantio/vouch

down, there would be heavy penalty as well" (P4). In the worst case, *"client bugs that result in validators voting for an invalid fork [...] can lead to the loss of the full amount of ETH staked by a validator"* (P5).

Diversity is deliberate, not spontaneous. Client diversity was set in motion by an explicit call from the Ethereum Foundation and is actively maintained through coordination, shared specifications, monitoring, targeted intervention against concentration, and sustained funding of minority clients.

3.3. Community's Perceptions on Benefits Software Diversity provides Ethereum

Interviewees consistently identify client diversity as one of Ethereum's core strengths. The benefits they describe fall into five interconnected areas: network robustness and resilience, security against targeted attacks, bug discovery through independent implementation, governance robustness, and greater innovation freedom for minority clients.

3.3.1. Network robustness and resilience

The most frequently cited benefit of client diversity is its contribution to network resilience. The central argument is that relying on a single codebase creates a single point of failure: *"we have less clients, more people will rely on same code base. And that will become a single point of failure in worst case. So more diverse clients, heavy testing"* (P4). A monoculture could leave the entire network vulnerable: *"If Ethereum only had a single client implementation, an attacker could take down the entire network by finding a single vulnerability in that client"* (P5).

With multiple independent implementations, a bug in one client affects only a fraction of the network while the remaining clients continue operating. As P4 summarizes: *"not all clients can go down at any point of time, right? So that is the strongest point of Ethereum"* (P4). Interviewees point to the Prysm outage as an example: *"after Fusaka, the Prysm client had an outage, which was like 30% of the network went down or 25%. And yeah, if Prysm was the only client, network would be offline"* (P3). EIP-7607: Fusaka[6] was an upgrade that went live on December 3, 2025, where the client Prysm had a performance bug. This led to excessive CPU and memory consumption and temporary unresponsiveness from the validator [**2, 60**].

[6] https://eips.ethereum.org/EIPS/eip-7607

Despite individual client failures over the years, the network as a whole has maintained continuous operation: *"Ethereum has zero downtime in the past 10 years"* (P4).

3.3.2. Security against attacks

Beyond accidental failures, client diversity provides defense against deliberate attacks. This benefit has historical precedent in Ethereum itself: *"the importance of software diversity became apparent years ago when clients were facing DoS attacks and Ethereum stayed online thanks to differing client implementations which were not vulnerable to the same attack"* (P5). Because different clients are written in different languages and employ different architectures, a single exploit is unlikely to compromise all implementations simultaneously.

This protection matters because vulnerabilities in complex software are essentially inevitable: *"Ethereum clients are complex pieces of software spanning tens of thousands of lines of code, often with further dependencies, therefore the likelihood of them containing some vulnerability is high"* (P5). The risk is not hypothetical — *"a very popular client, Lighthouse, just had a high-priority release 2 days ago that fixed a critical vulnerability"* (P5). Without diversity, such a vulnerability could have affected the entire network rather than a single client's share.

Interviewees also highlight the insider threat as a motivation: *"Malicious developer injects some malicious code in this client, then basically the blockchain goes down"* (P7). With multiple independent codebases, the repercussions of any single compromised client is limited to its proportion of the network.

3.3.3. Bug discovery through independent implementation

Client diversity does not only contain the damage from bugs — it also helps surface them earlier. Because each client implements the same specification from scratch using different architectures and design decisions, they tend to expose different types of issues: *"clients find different issues when you implement them in your code, because they're different architectures"* (P3). This creates a form of cross-implementation review where *"A bug in one implementation can give other client implementers a hint at what areas of their own codebases could be vulnerable to similar bugs"* (P5).

The benefit extends beyond code to the specification itself. *"Multiple different client teams all agree on a specification, and by independently implementing this specification, many people get to review it and discover potential blind spots and areas of ambiguity"* (P5). The act of implementing the same protocol in different languages functions as a distributed audit: *"client diversity helps test the piece of code in different ways. Like it gives flexibility to test the code in different environment, different language and how they are performing"* (P4).

Diversity also creates a positive feedback loop at the implementation level: *"Not only does the broader system's (Ethereum) reliability increase, it also helps each implementation become more robust"* (P5). Each client team learns from failures in other implementations, raising the quality of every client over time. The overall assessment is concise — *"More diverse, more better and robust system"* (P4).

3.3.4. Governance robustness

Client diversity also strengthens the governance process by distributing influence across multiple independent teams. As P3 explains, contrasting Ethereum with a single-client ecosystem: *"it also makes the governance process more robust because you don't have a, like you see this a lot in Bitcoin, where Bitcoin Core has too much power of what gets implemented and what doesn't. So, since we have different teams [...] different teams have a say in the process and you can't capture the governance that easily"* (P3).

When new protocol changes must be agreed upon and independently implemented by multiple teams. This structural pluralism makes it harder for any single actor to capture governance and ensures that proposed changes receive scrutiny from multiple independent actors.

3.3.5. Innovation freedom for minority clients

A less obvious benefit of diversity is that it creates space for experimentation, particularly among minority clients. When a single client dominates the network, its team faces intense pressure to prioritize stability over innovation: *"you become really conservative with doing experimental stuff, doing features that might be unstable [...] So if you have more clients, each client can experiment more"* (P3).

The relationship between market share and conservatism is observable: *"when they were a majority, they felt a lot of pressure and once they weren't a majority, they could, they felt like they didn't have to do that much work on maintenance and sometimes work on new features"* (P3). This pattern is currently playing out with one of the larger consensus clients: *"I know from Lighthouse that they are getting more conservative [...] I think their share is now 40%-ish"* (P3).

Minority client teams confirm this dynamic from their own experience: *"since we are a minority client, it's easier for us to experiment. Although [...] we still don't want to break on maintenance [...] but yeah, you have less pressure"* (P3).

Diversity strengthens the network. It improves robustness and resilience against single-client failures, increases security against attacks, surfaces bugs through independent implementations, and distributes governance power across teams.

3.4. Community's Perceptions on Challenges software Diversity introduces in Ethereum

While the preceding sections describe how client diversity is maintained and the benefits it provides, interviewees are equally candid about its costs. The challenges they identify fall into six areas: coordination overhead that scales with the number of clients, governance complexity arising from a large and heterogeneous group of stakeholders, financial sustainability pressures on minority client teams, protocol-level complexity that can undermine diversity's protective value, the relationship between diversity and Ethereum's finality mechanism, and a persistent pattern in which users migrate between majority clients rather than distributing evenly.

3.4.1. Coordination overhead

The most frequently cited challenge is the coordination burden that grows with each additional client.

As P4 states plainly: *"in Ethereum, the biggest challenge with diversity is coordination"* (P4). The cost is not abstract — it manifests directly in the resources required for network upgrades: *"More clients, more coordination. And that increases the cost of manpower and the time that goes into an upgrade. We want that all clients are ready in order to proceed with an upgrade"* (P4).

The scale of this coordination has grown substantially. With multiple execution and consensus clients that can be paired in any combination, the ecosystem now faces *"42 client combinations between consensus and execution clients"* (P1), and on governance calls, participants must evaluate proposals that *"essentially no one has read except the people that wrote it and two or more"* (P1). The result is a sense that the process has become unwieldy: *"client diversity has made the process of choosing how Ethereum will continue progressing quite difficult because there's just too many cooks in the kitchen"* (P1), and that *"it's hard to make any decisions because we're just too many"* (P1).

The burden falls disproportionately on majority clients, whose failures would have the greatest impact on finality. One interviewee observes that this pressure is currently affecting the largest consensus client: *"this is probably now hitting Lighthouse. That they have now*

a large majority. They could actually prevent validation. And they are being stressed out. That's why they're hiring a lot" (P1).

Not all participants view the slowdown as purely negative. As P3 notes, the deliberation that accompanies coordination also brings scrutiny: *"it makes the whole process slower, but that can be positive and negative because you get so many different perspectives on the code you ship"* (P3).

3.4.2. Governance complexity

Beyond the pace of decision-making, interviewees describe structural tensions in how the ecosystem governs itself. Influence is unevenly distributed despite the large number of participants: *"even though it's a very large group, not everyone's voices is the same. And even non-clients and non-researchers' voices have very heavy weight. If Vitalik*[7] *comes to any of their meetings, or any of our meetings, his voice is going to be heard more than anyone else's"* (P1).

The challenge intensifies as the ecosystem grows. Historically, governance was manageable when client teams were few: *"Historically, this role belongs to Ethereum core developers who have generally been accepting of new client teams joining discussions once those new clients matured. But the more people are part of those discussions, the more challenging it is to come to consensus and make decisions"* (P5).

There is also a tension between the centralised and decentralised dimensions of the network. On one hand, a small number of large staking pools hold enough influence to direct client choices; on the other, those pools are not monolithic: *"we are centralized enough in which a few owners of pools or big voices in staking pools can mandate particular clients or force people out of some clients. And decentralized at the same time in which those pools are not being operated by a single person"* (P1). This duality complicates governance: the same concentration that enables rapid intervention against client monoculture also means that a few actors can disproportionately shape outcomes.

3.4.3. Financial sustainability

Maintaining multiple independent client teams is expensive. The direct costs of diversity are inherent to the model: *"The direct costs are obviously high — every implementation requires a team to work on it, and while those teams may get smaller as development gets more efficient with the help of AI, there will always be some amount of direct implementation overhead for each client"* (P5).

[7]Vitalik Buterin is the founder and main architect of Ethereum

While the Ethereum Foundation and ecosystem grants provide initial support (as discussed in Section 3.2), sustaining a client over the long term requires additional revenue. Community donations alone are insufficient: *"Public donations are in my experience insufficient to sustain a project like this"* (P5). As a result, client teams must develop their own business models. One interviewee describes a dual strategy: *"1) growing our staking business, effectively subsidizing Vero as a public good; 2) selling a Vero version with advanced features for professional node operators, bundled with priority support"* (P5). The tension between building a public good and finding a sustainable revenue model is a recurring concern for minority client teams.

3.4.4. Protocol complexity undermining diversity

Perhaps the most fundamental challenge is that client diversity does not protect against bugs that originate in the protocol specification itself. As P1 explains: *"most bugs that we see actually [...] leak from the protocol implementation into all clients. So even client diversity is lost when everyone is having the same exact same bug"* (P1). For this participant, *"protocol complexity is worse than client diversity"* (P1) as a source of systemic risk.

Writing clients in different programming languages does not guarantee that they will fail differently. As P3 observes: *"just because you use a different programming language or framework doesn't mean you don't have the same issues"* (P3). The Holesky testnet incident provided a concrete demonstration of this risk: *"We have seen a very bad situation play out on a testnet last year (the 'Holesky incident'), where 3 clients shared the same kind of bug, and it resulted in a very bad situation on that testnet. If the same thing were to happen on Ethereum mainnet, it would be a catastrophe"* (P5).

Even at the individual client level, the complexity of the specification can create hidden vulnerabilities. A client can appear to function correctly while cutting corners on edge cases: *"you can have a client that enforces everything on the happy case of the spec. It works perfectly fine. It's much more performant than any other client because it's greedy on everything else. It doesn't do anything that doesn't have a specific enforcement in the spec"* (P1). Such shortcuts may go undetected until adversarial conditions expose them.

3.4.5. Diversity as a consequence of finality

Several interviewees frame client diversity not as an intrinsic good but as a necessary response to a specific architectural choice: finality. As P1 puts it: *"I have mixed feelings about client diversity. It's a problem that is self-imposed by having finality"* (P1).

The argument is that finality — the property that makes confirmed transactions irreversible — transforms implementation bugs from recoverable incidents into potential catastrophes. Without finality, the community could simply reorganize the chain to undo an invalid block: *"if we didn't have finality, then we could just reorganize an invalid chain. Just go back. And this we could easily do via just the community getting together"* (P1). With finality, there is no such fallback: *"If we didn't have finality, this would not be an issue at all. You could just have a single client. The fact that we have finality and [...] it's just a catastrophe to try to recover from finalizing an invalid chain is what makes this drive to have client diversity"* (P1).

This framing does not reject finality itself. As P1 acknowledges, finality is a valued property of the network: *"Finality at the same time is something good. It's something we have. But it was a decision [...] And having finality forces us into having to deal with the fact that we might finalize an invalid chain"* (P1). Client diversity is thus understood as the cost of an architectural decision that the ecosystem has deliberately chosen to retain.

3.4.6. The majority migration problem

Even when the community successfully reduces the dominance of one client, the underlying dynamic tends to reassert itself. Users select clients based on popularity metrics rather than technical evaluation: *"people always pick the second best or 'best' in quotes client based on just the percentage, because it's hard to reason about the details [...] that's why people switch from Prysm to Lighthouse. And then now Lighthouse is becoming too big"* (P3).

This is not a one-time event but a recurring pattern. After the campaign to move users away from Prysm (subsection 3.4.1), the majority simply migrated: *"now something similar might happen with Lighthouse, which is another consensus client that actually took most of our users"* (P1). The gap between economic theory and observed behavior is clear: *"In theory this should lead to an equal distribution of clients used as everyone attempts to minimize the chances of correlated downtime. Practice so far has turned out differently though"* (P5).

Diversity is costly but accepted. The community bears governance complexity, the financial burden of sustaining minority clients, growing protocol complexity, and the risk of a majority client causing incorrect finality, because the alternative—a single point of failure in an irreversible system—is far worse.

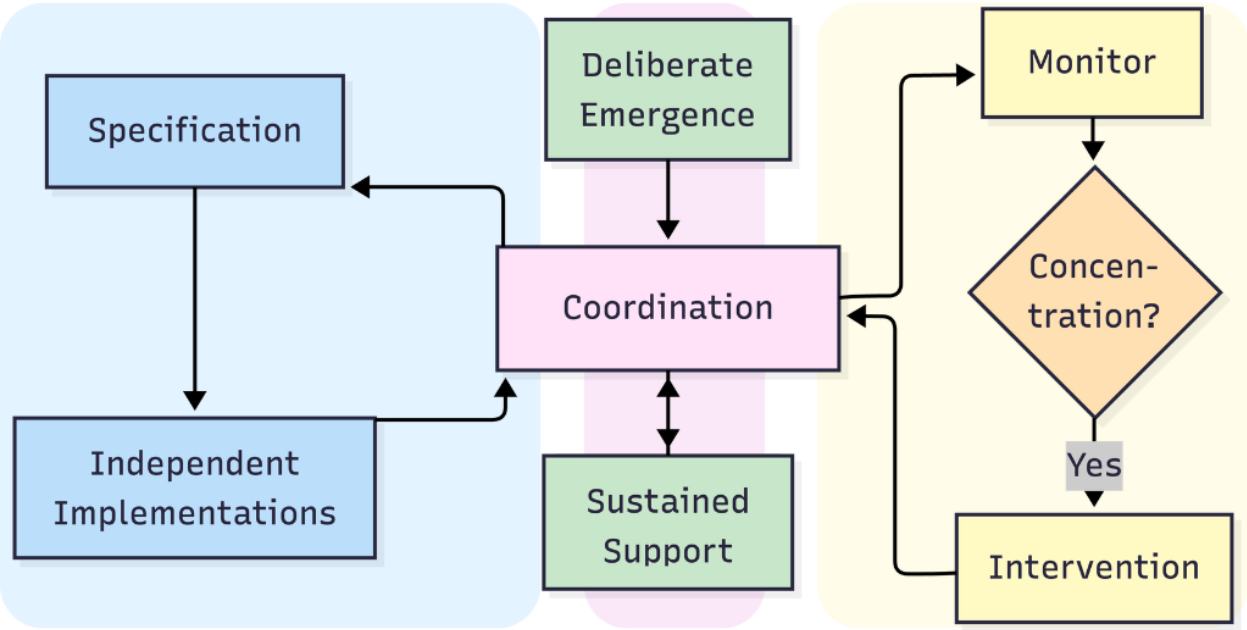

Fig. 3.1. Overview of the governance mechanisms sustaining client diversity in Ethereum. Three interconnected regions illustrate the governance foundation (center), the specification–implementation loop (left), and the monitoring–intervention loop (right), all mediated by coordination.

3.5. Overview of Sustainable Software Diversity in Ethereum

The interviews with 7 key persons involved with software diversity for Ethereum provide novel, qualitative insights about the diversification process and its challenges. In Figure 3.1, we summarize this process and discuss it further in this section.

Figure 3.1 is organized into three interconnected regions, all mediated by *Coordination* (center). Coordination does not reside in a single central authority, it is a distributed function carried out by the entire community. An interviewee refers to this from a developer's point of view: *"since we have different teams, [. . .] the coordination over it also means like different teams have a say in the process and you can't capture the governance that easily"* (P3).

The Ethereum Foundation's explicit call for independent implementations (*deliberate emergence*) feeds into *coordination*, which in turn channels resources and economic incentives back to client teams through *sustained support*, an external input to this loop. This support consists of seed grants to new client teams and the Client Incentive Program funded by the Ethereum Foundation [**29**]. The Foundation's treasury is held mostly in ETH and is spent to fund public goods; it periodically sells ETH to maintain a multi-year operating runway [**30**].

Software diversity in Ethereum is therefore intentional rather than organic: coordination preserves it and sustained support makes it viable.

Coordination connects the two other regions that sustain diversity in practice:

The specification–implementation loop shows how *Specification* and *Independent Implementation* form a cycle. Initially, the relationship is top-down: the specification defines what each client must implement. Over time, implementations introduce new features into the specification and vice versa. "Sometimes it's first on the spec. Sometimes there's a proof of concept in a client and then brought to the spec" (P3), coordination ensures that this flow remains orderly.

The monitoring–intervention loop illustrates the ecosystem's mechanism for detecting and responding to dangerous concentration. *Coordination* triggers *Monitoring* of client distribution; if concentration is detected, *Intervention* is carried out through social-media campaigns, adjusted guidance, and pressure on staking pools: "when we were approaching the limit. Then, just go out and say to everyone, please change" (P7).

Although the right and left regions can operate independently, they are interconnected through Coordination: insights from monitoring can influence specification priorities, and implementation experience can reshape the criteria used for intervention.

3.6. Conclusion

The interviews reveal how Ethereum introduces and actively maintains client diversity, together with the benefits and costs this brings. The main contribution of this chapter is Figure 3.1, which identifies the mechanisms that maintain multiple implementations in a decentralized system with no central authority and how they reinforce each other. Ethereum's 11 clients cannot be transferred to another system, but the mechanisms behind them can, which makes this guide directly applicable to any project that wants software diversity by design rather than by accident.

A system that intends to reproduce Ethereum's outcome has to take four steps. It must set an explicit goal, in this case that no single implementation controls a share of the network large enough to threaten the system. It must maintain a specification precise enough for teams to implement it independently, without reading each other's code. It must build visibility into how implementations are distributed across the network, because concentration cannot be corrected before it is measured. And it must educate its community about the risks, since every intervention our participants describe worked by persuading operators rather than by enforcing a rule in the protocol.

None of these steps is free. A community that adopts them accepts coordination overhead, slower decisions, and minority teams that must fund a public good. Ethereum pays these

costs because the alternative, a single point of failure in a system whose transactions are irreversible, is far worse.

Chapter 4

Quantitative Assessment of Software Supply Chain Diversity Among Ethereum Clients

In this chapter, we analyze the diversity within the codebase of the 11 major Ethereum clients, execution and consensus, listed in Table 2.1. Client-distribution surveys measure diversity at the product level. Here we look underneath that surface, at the shared components where independence can be undermined.

Open source software has amplified the availability of code for reuse, increasing development productivity and preventing teams from reinventing the wheel. The counterpart of this reuse is that every project inherits the code, and the faults, of the packages it depends on, together with those of their transitive dependencies. This turns the software supply chain into a large and distributed attack surface, and attacks that exploit it are now a well-documented class of threats [**48**]. The SolarWinds Orion breach illustrates what is at stake: attackers inserted malware into SolarWinds' build process, and the compromised update was distributed to approximately 18,000 customers, including major corporations and U.S. government agencies [**58**]. A single trusted component was enough to bypass the defenses of every organization downstream of it.

In Ethereum, a supply-chain compromise carries a second cost beyond the security of any individual client, it erodes the very property that client diversity is meant to guarantee. Diversity protects the network because a bug in one implementation is not a bug in the others, so a faulty client cannot take the chain down as long as it stays below a critical share of the network. That argument holds only if the implementations fail independently. Two clients written in different languages by different teams still fail together if they depend on the same library, and the transactions they settle are irreversible, which makes the consequences of a correlated failure permanent [**71**]. Shared dependencies therefore act as hidden couplings between nominally independent clients, and client-distribution figures such as Figure 2.2 do not reveal them.

We therefore study the software supply chain of the Ethereum clients, the first systematic comparison of this kind across all clients. External dependencies make up a large share of each codebase, and independently implemented clients can still share them, within and across language ecosystems, so a single faulty or compromised package can affect more of the network than client-share percentages alone suggest. We compare clients within each language ecosystem, where overlap can be analyzed systematically, and then identify libraries that appear across all ecosystems.

We then zoom in on one dependency that no client can avoid: the database backend used to store chain and state data. Storage is central to any blockchain: every node must keep a growing copy of the chain and its state, write to it constantly, and stay available. Clients do not implement their own, custom storage, they rely on existing databases, and only few databases are fast enough and can hold enough data for a system the size of Ethereum. Language diversity therefore does not automatically imply storage diversity: clients can converge on the same engine and reintroduce a shared failure point.

These analyses delve into how 11 independent projects interpret the same specification and implement it differently, where they succeed in diverging, and where software diversity is limited. In the following, we first describe our methodology, then examine the software supply chain of third-party dependencies and database backends. Finally, we discuss the implications for diversity.

4.1. Methodology

We analyze the 11 major Ethereum clients: the five execution clients Besu, Erigon, Go Ethereum, Nethermind, and Reth, and the six consensus clients Grandine, Lighthouse, Lodestar, Nimbus Eth2, Prysm, and Teku. Each repository is studied at the commit recorded in Table 2.1, so all measurements refer to a fixed snapshot of the ecosystem as of October 2025.

There is no canonical reference architecture, or *blueprint*, establishing how an Ethereum client must be built. Client teams share protocol specifications but remain free to choose languages, storage engines, library stacks, and internal designs. For that, it is difficult to compare and study diversity among the clients. Therefore, the authors, using the documentation available and their expertise in the field, hand pick two key components critical in node operation and likely sites of unintended convergence: database backends and third-party dependencies.

In order to assess the diversity of the software supply chain of Ethereum, we extract each client's dependency tree with the native tooling of its language ecosystem, including transitive dependencies. We resolve Go client dependencies with `go list -m all`, Rust

client dependencies with `cargo tree`, and Java client dependencies with Gradle's runtime classpath resolution. Nethermind and Lodestar are studied using the respective package managers `dotnet` and `npm`, and dependencies are analyzed in Nimbus Eth2 from its vendored Git submodules. Resolved dependencies are then aggregated by organization and repository URL.

Within each language that hosts more than one client (Go, Rust, and Java), we compute set intersections to measure shared packages. Cross-ecosystem comparison cannot rely on package names alone: naming conventions differ across ecosystems, and bindings or wrappers of the same native library often use different identifiers. We therefore manually identify libraries that recur across ecosystems, with particular attention to networking and cryptography. The analysis scripts and raw outputs are available online.[1].

Among the dependencies studied, database engines stood out as a category warranting closer inspection. We identified all third-party database engines and traced their usage in the source code with the assistance of LLMs. With the collected information, for each client we identify the persistent store used for chain and state data—and, where relevant, secondary roles such as validator slashing protection—by inspecting documentation, configuration defaults, and source code at the pinned commit. We record the default engine, any operator-selectable alternatives, and engines used for distinct purposes. The resulting inventory is reported in Table 4.2.

4.2. Software Supply Chain: Third-Party Dependencies

A large share of any project's codebase is typically drawn from external packages, which accelerates development by allowing teams to reuse mature implementations rather than reimplement common functionality. Open source also brings transparency and the possibility of community scrutiny.

These benefits, however, come with trade-offs: projects inherit the vulnerabilities of their dependencies, and when independently developed systems rely on the same third-party packages, software diversity is undermined. In Ethereum, clients that appear independent in surveys such as Figure 2.2 may still share common supply-chain components, so that a single compromised or faulty library can affect a larger fraction of the network than client-distribution figures alone suggest.

In this section, we address that gap by studying the third-party dependencies of the 11 clients in Table 2.1 implemented in six different programming languages: Go, Rust, Java, TypeScript, C and Nim. We analyze overlap within each language ecosystem that has more

[1] `https://github.com/nala7/eth_client_diversity`

than one client (Go, Rust and Java) and later identify common libraries across all ecosystems. Fully automated comparison is not reliable: package managers and extraction tools differ by ecosystem, which can yield different naming standards and sources, and language-specific bindings or wrappers of the same underlying library are not detected by name-based matching. The analysis is therefore conducted manually.

4.2.1. Overlap of Go clients third-party dependencies

Three of the clients in the Ethereum ecosystem are implemented in Go: the execution clients Go Ethereum and Erigon, and the consensus client Prysm. Go is a general-purpose programming language designed at Google to address the complexity of large-scale concurrent software systems [**26**], which makes it a natural fit for a complex decentralized system such as Ethereum.

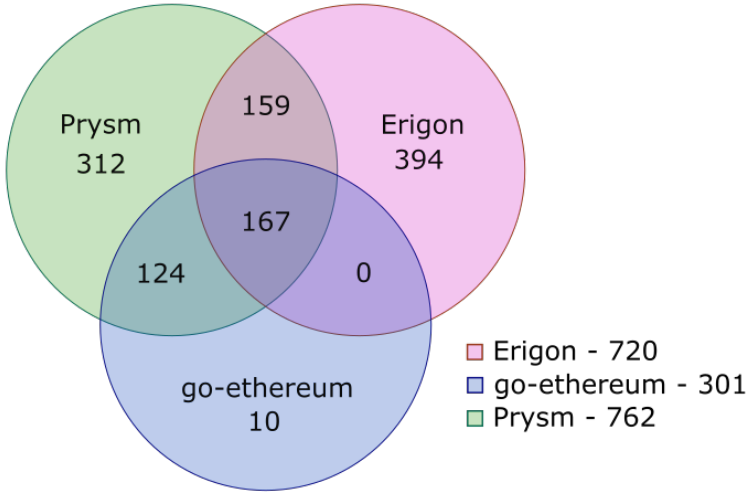

Fig. 4.1. Overlap of third-party dependencies among the three Go-based clients: Erigon, Go Ethereum, and Prysm

Figure 4.1 shows the overlap of third-party dependencies among the three Go clients. In total there are 1168 distinct Go modules, of which 167 (roughly 14%) are common to all three and 450 (38%) are shared by at least two clients. Go Ethereum declares 302 dependencies and has only 10 that are unique to it. Erigon and Prysm are considerably larger, with 721 and 763 dependencies, and retain 395 and 312 client-specific modules, respectively. This pattern is consistent with Go Ethereum being the earliest client and a reference implementation whose libraries are widely reused, while later clients have grown larger dependency trees around specialization.

The shared modules include several low-level building blocks that are central to client operation. The core numeric primitive `uint256` provides the fixed 256-bit integers that

match the native EVM word size and is authored by a Go Ethereum core developer. The library `gohashtree`, implemented by the Prysm developers, is used in all Go clients and handles hashing. It is written largely in assembly and is a SHA-256 library highly optimized for Merkle-tree computation. `Snappy` is a Go implementation of Google's Snappy algorithm used to quickly compress and decompress data shared in all 3 clients.

Beyond individual libraries, some GitHub organizations recur across common dependencies. Among the most frequent are Prometheus and Pion, each contributing four modules to the set shared by all three clients. Prometheus is a widely used monitoring and alerting toolkit designed to collect metrics, while Pion is an organization that develops networking libraries in Go.

A notable result is that there are no dependencies exclusive to both Erigon and Go Ethereum: every module they share is also a dependency of Prysm. After Erigon forked from Go Ethereum in October 2019, its dependency set has diverged to the point that the residual overlap with Go Ethereum is precisely the core stack that an independently developed Go client (Prysm) also needs.

By contrast, Erigon and Prysm share 159 modules that Go Ethereum does not, including 15 modules from libp2p responsible for the peer-to-peer communications in the clients. Meanwhile, Go Ethereum and Prysm share 125 modules absent from Erigon. Interestingly, Prysm depends directly on Go Ethereum to interoperate with the execution layer and reuse low-level Ethereum protocol utilities, and thereby inherits much of Go Ethereum's transitive dependency tree. This represents approximately 97% of Go Ethereum's listed modules. This is a serious limitation to the diversity of Ethereum, if a malicious actor tampers with Go Ethereum, it would affect 41% of the execution clients but also 31% of the consensus clients, as shown in Figure 2.2.

Where clients diverge, they often solve the same problems with different third-party libraries. For storage, Go Ethereum defaults to Pebble, Erigon to MDBX, and Prysm to bbolt, as described in Table 4.2. In-memory caching follows a similar pattern: Go Ethereum relies on fastcache and bigcache, Erigon on golang-lru and freelru, and Prysm on ristretto.

Although Erigon forked from Go Ethereum, it shares fewer third-party dependencies with Go Ethereum than Prysm does. This shows that consistent and active action to diversify from other implementations is needed, although they are in different layers of the Ethereum node and at a surface level they are considered different, they share the same vulnerabilities in their external source code.

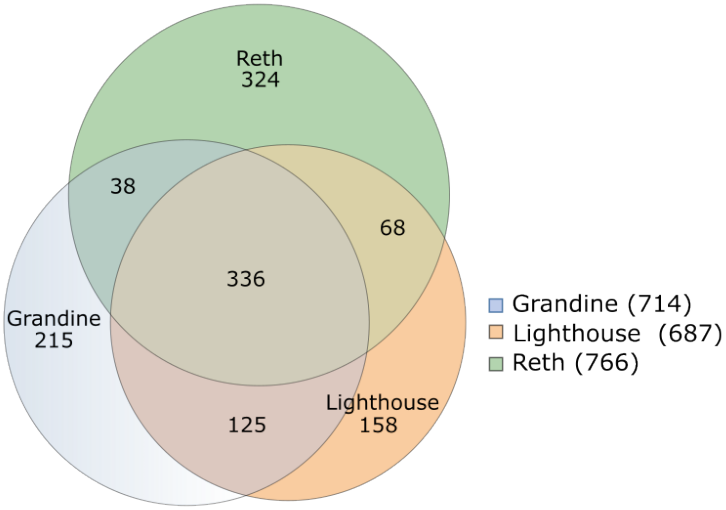

Fig. 4.2. Overlap of third-party dependencies among the three Rust-based clients: Grandine, Lighthouse, and Reth

4.2.2. Overlap of Rust clients third-party dependencies

Three Ethereum clients are implemented in Rust: the execution client Reth and the two consensus clients Grandine and Lighthouse. Rust is a general-purpose programming language that emphasizes performance, type safety, and concurrency. It also enforces memory safety—meaning that all references point to valid memory—without a garbage collector [**11, 45**]. These qualities make Rust a popular language in the Ethereum blockchain.

Figure 4.2 shows the overlap of the third-party dependencies of the three Rust clients. In total there are 1264 distinct Rust libraries, of which 336 (roughly 27%) are common to all three. 55% of the all the libraries are client-specific libraries. The two consensus clients, Grandine and Lighthouse, share 125 libraries that are not used by Reth, a higher number than the other pairwise combinations. This suggests that dependency convergence is driven not only by the use of the Rust ecosystem, but also by whether a client implements the same node layer.

The shared libraries fall into three groups. Runtime and networking rest on `tokio` together with a common HTTP/TLS stack (`hyper`, `rustls`). Cryptography includes `k256` (secp256k1) and crates from dalek-cryptography. Ethereum-specific modules include `alloy-rlp` for RLP serialization and `discv5` for peer discovery.

Aggregating dependencies by their organization, the three clients reference 419 distinct owners, of which 168 (40%) are common to all three. The most frequent is rust-lang with 17 shared libraries, consistent with hosting the Rust language toolchain. Among the top 10 most frequent organizations some are dedicated to rust language related implementations

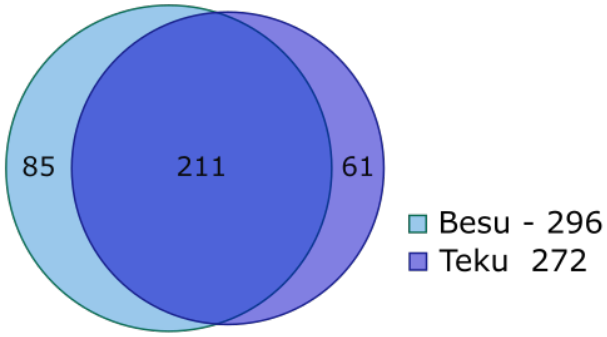

Fig. 4.3. Overlap of third-party dependencies of the two Java-based clients: Besu and Teku

like Tokio. Others develop tools and libraries for networking and security like Bytecode Alliance and wasm-bindgen.

There are also individual GitHub users who contribute extensively to the Rust ecosystem, such as dtolnay and BurntSushi. These are highly active maintainers with more than 100 repositories each and are responsible for several widely used libraries. However, reliance on individual accounts can introduce ecosystem risks. Unlike projects backed by organizations, the long-term maintenance of these libraries may depend on a single person's availability, interests, or resources. If a maintainer becomes inactive, critical dependencies may experience delayed updates, unresolved security vulnerabilities, or compatibility issues. Individual accounts may also present a greater risk of supply-chain attacks if an account is compromised.

4.2.3. Overlap of Java clients third-party dependencies

Besu and Teku are Java implemented execution and consensus clients respectively. They are both owned by Consensys, a blockchain software company that develops tools and applications for the Ethereum ecosystem. This can limit software diversity: implementing the same policies, culture, and work environment tends to produce more similar implementations.

Analyzing the emails used to commit to both repositories, out of the 294 contributors of Besu and 128 of Teku, 48 contributors appear in both GitHub repositories, representing 15% of Besu's and 34% of Teku's workforces, respectively. This is a limitation in the software diversity of these clients. Software diversity depends on the dependency of the implementation process, in this case teams are not entirely separate which can lead to similar architectures, implementations and functionalities.

Figure 4.3 shows that of 358 distinct Maven artifacts, 210 (roughly 59%) are shared: 71% of Besu's 296 dependencies and 77% of Teku's 272. This pairwise overlap is substantially higher than the all-client intersections for Go (roughly 14%) and Rust (roughly 27%), and

it confirms earlier findings that more than half of the Java clients' dependencies are shared [**71**].

The common artifacts concentrate on operationally critical stacks. Networking is dominated by Netty (43 modules in the intersection). Cryptography includes Bouncy Castle together with Consensys packages such as `tuweni` for low-level Ethereum utilities. For storage, both rely on `rocksdbjni`, consistent with Table 4.2. Teku further depends on several `org.hyperledger.besu` native modules, including `secp256k1` and `blake2bf`, reusing cryptographic native code packaged for the execution client—analogous to Prysm's dependence on Go Ethereum.

Aggregating by organization, the clients share 42 of 84 distinct owners (50%). The most frequent sources are the Java Native Runtime project (8 repositories), Netty, and FasterXML (4 each), alongside Consensys itself through in-house libraries. Although the clients implement different node layers and appear as separate entries in surveys such as Figure 2.2, they largely inherit the same third-party risk surface. A bug or malicious change in a widely shared Java library or in components maintained for both products, could therefore affect execution and consensus capacity together in the Java portion of the network.

4.2.4. Common third-party dependencies across all Ethereum clients

Clients depend on different package ecosystems, therefore dependencies rarely overlap by package identity across all of them: a Rust crate is distinct from a Maven artifact, a Go module, or a NuGet package. Meaningful cross-ecosystem coupling instead appears when clients share native libraries through bindings, or independently adopt parallel implementations of the same protocol. Those cases are supply-chain single points of failure for Ethereum as a whole.

Networking is one such case. libp2p provides modular peer-to-peer stacks in several languages: Rust implementations are used by all three Rust clients, Go modules by Erigon and Prysm, and a JavaScript stack by Lodestar. A flaw in the protocol design, or in a widely reused implementation family, can therefore affect clients that otherwise look diverse in surveys such as Figure 2.2.

The densest concentration is in cryptography: `blst` is an optimized Assembly implementation of the BLS12-381 cryptographic curve, which is used for efficient digital signatures in Ethereum's consensus layer. Clients that do not depend on it by name reach the same code through wrappers such as jblst (Teku), blst-ts (Lodestar), and blst-bindings (Nethermind), so implementations in different languages ultimately verify signatures with the same piece of assembly code. Client developers treat this as a deliberate exception to diversity rather

than an oversight: *"we have that single point of failure, all clients. So the BLS signature verification [...] every client uses the same assembly. That's why we invested millions in that assembly to get it audited correctly"* (P1, Table 3.1).

Furthermore, c-kzg-4844 provides a carefully maintained C implementation of KZG commitments—with Go and Rust counterparts go-eth-kzg and rust-eth-kzg—and is integrated into all the clients in Table 2.1, either directly or via bindings. libsecp256k1, originally developed for Bitcoin, similarly underpins transaction signatures across much of the ecosystem. Nimbus Eth2, for example, exposes both libsecp256k1 and an in-house alternative.

These cryptographic components are difficult to implement correctly and have received sustained expert review, often over many years. By reusing them, client teams concentrate their implementation and testing effort on primitives critical to chain security rather than proliferating unverified alternatives. Ethereum thus benefits from a deliberate balance: language and architectural diversity reduce failures at the client layer, while shared, heavily scrutinized cryptography reduces the risk of independent mistakes in the hardest parts of the stack.

That same concentration is nonetheless a liability. Shared libraries amplify the consequences of a bug or compromise, and several findings in this section show that diversity is already more limited than client-count surveys suggest. Within Go, Prysm's direct dependence on Go Ethereum pulls nearly the entire Go Ethereum module set into a consensus client. Within Java, Besu and Teku share roughly 59% of their Maven artifacts, 71% and 77% of each client's dependency tree, confirming prior work on their overlapping supply chains [**71**]. Cross-ecosystem reuse of `blst`, KZG, and related primitives extends that vulnerability further.

Language diversity is an effective strategy for supply-chain diversity. Ethereum clients collectively leverage a broad range of open-source dependencies across six language ecosystems, and the resulting diversity is substantial: cross-ecosystem overlap is limited to few deliberately shared cryptographic primitives. Within a single ecosystem, however, overlap grows considerably. Diversity gains of multi-language development diminish when clients share a package manager and its library pool.

4.3. Software Supply Chain: Database Backend

Persistent storage is a fundamental component of Ethereum clients, which use database engines to maintain blockchain and other node data. Table 4.1 summarizes the database engines used by Ethereum clients.

Database	Language	Storage	API	Lineage	Year	cloc
LevelDB	C++	LSM-tree	Key-Value	Independent	2011	19k
RocksDB	C++	LSM-tree	Key-Value	LevelDB	2013	320k
Pebble	Go	LSM-tree	Key-Value	LevelDB	2018	112k
LMDB	C	B+tree	Key-Value	Independent	2011	14k
bbolt	Go	B+tree	Key-Value	LMDB	2013	6k
MDBX	C	B+tree	Key-Value	LMDB	2015	49k
redb	Rust	B+tree	Key-Value	LMDB	2022	24k
SQLite	C	B-tree	SQL	Independent	2000	144k

Table 4.1. Embedded database engines used in major Ethereum clients. For each engine we report its implementation language, on-disk storage structure, the API it exposes, the project it descends from, the year of its first release, and its size in lines of code as measured with `cloc`.

Across the 11 clients, eight databases are used: LevelDB, RocksDB, Pebble, LMDB, bbolt, MDBX, redb and SQLite. All of them are embedded engines that run inside the client process rather than as a separate server, and none of them was written by a client team. They differ in three dimensions:

The first is the on-disk data structure, which splits them into two families. LSM-trees organize data by first buffering writes in memory and then storing them on disk in a structure that is optimized for efficient updates. This design favors write-heavy workloads, at the cost of the read and space amplification caused by having to consult and rewrite several runs [**56, 51**]. B-trees, by contrast, maintain a single balanced index that is updated in place, which favors read-heavy workloads and ordered scans [**7, 22**]. The B+tree is the refinement of that structure used by most modern embedded stores: internal nodes hold only separator keys while all records live in the leaves, so the fan-out is higher and the leaves can be linked to make range scans efficient [**22**].

The second dimension is the interface exposed to the client. Seven of the eight engines provide a key–value interface, which offers a simple and efficient way to store and retrieve data using unique identifiers. SQLite is the only relational database, trading some performance for a structured schema and more flexible querying capabilities.

The third is maturity: all eight are long-lived projects, from SQLite, maintained since 2000, to redb, first released in 2022, and their sizes span two orders of magnitude, from 6k lines of code in bbolt to 320k in RocksDB.

The LSM family used by Ethereum clients descends from a single project. LevelDB, released by Google in 2011, is an embedded key–value store built on an LSM-tree and

remains the reference implementation of that design [**38**]. RocksDB was forked from it by Meta shortly afterwards and extended with additional features and configuration options aimed at modern storage systems and data intensive workloads. These additions have made its codebase substantially larger than that of LevelDB [**25**]. On the other hand, Pebble is a pure-Go reimplementation of the same design, developed by Cockroach Labs to remain compatible with RocksDB on disk while avoiding the cost of calling into C from Go [**53**].

The B+tree family likewise descends from a single project. LMDB (Lightning Memory-Mapped Database) was developed for the OpenLDAP project as a memory-mapped, copy-on-write B+tree: reads are served directly from the mapped pages without locking, and a single writer at a time commits a new version of the tree, which yields low read latency and crash consistency without a write-ahead log [**20**]. bbolt is the Go-native descendant of that design, maintained by the etcd project and widely used in infrastructure software [**28**]. MDBX is the C descendant, which extends LMDB with larger database limits, stricter consistency checks, and lower, more predictable latency [**79**]. redb is the most recent member of the family, a Rust reimplementation of the same copy-on-write B+tree design, first released in 2022 [**9**].

SQLite is the outlier of the set. It is a mature, lightweight relational engine in C, maintained since 2000, that stores its tables and indices in B-trees and exposes them through SQL [**72**].

	Client	Language	Databases
Execution	Besu	Java	RocksDB
	Erigon	Go	MDBX
	Go Ethereum	Go	Pebble (default) / LevelDB
	Nethermind	C#	RocksDB
	Reth	Rust	MDBX; RocksDB
Consensus	Grandine	Rust	MDBX; SQLite
	Lighthouse	Rust	LevelDB (default) / redb; LMDB; SQLite
	Lodestar	TypeScript	LevelDB
	Nimbus Eth2	Nim	SQLite
	Prysm	Go	bbolt
	Teku	Java	RocksDB (default) / LevelDB

Table 4.2. Database backends used by major Ethereum clients. *Default* denotes the engine used by default; "/" separates an alternative or fallback for the same role; ";" separates engines used for other purposes.

As shown in Table 4.2, Ethereum clients implemented in diverse programming languages do not necessarily use diverse storage backends. Although Go Ethereum, Lighthouse, Lodestar, and Teku are implemented in different languages, they all support LevelDB. Three of the six consensus clients expose it as a backend, representing up to 60% of the consensus layer. If the validators running LevelDB-backed instances together exceed one third of the stake, a fault that takes them offline can halt finality; if they exceed two thirds, a consensus-breaking defect could finalize an incorrect chain.

Relative to the current client distribution in Figure 2.2, MDBX and RocksDB currently represent a lower concentration risk, however they still appear in three and four clients respectively, so the same threshold logic applies if their combined validator share grows. Since the share of the blockchain that a client holds is dynamic, having multiple implementations using the same storage solution is a vulnerability.

SQLite appears in three consensus clients, yet it plays different roles among these clients. In Nimbus Eth2 SQLite is the primary beacon-chain database, holding blocks, states, blobs, and related tables, and it also backs validator slashing-protection data. By contrast, in Grandine and Lighthouse, SQLite is not the chain store: both use it for a validator slashing-protection database that records previously signed blocks and attestations so the client can refuse slashable double-signing. Although this is a common vulnerability in all three clients, it is used in different functionalities, making targeted attack harder

Two databases stand out as outliers. Go Ethereum defaults to Pebble and Prysm is the only client whose primary store is bbolt. Both are Go implementations, showing that diversity in storage choice is achievable even within a shared language ecosystem, and that defaulting to the most popular engine remains a risk even there.

Storage solutions are among the most constrained components of a client: they must sustain high write throughput while storing large volumes of chain and state data, and relatively few mature embedded engines meet those requirements. The ecosystem has nonetheless achieved diversity in the solutions used. That diversity remains fragile wherever several clients converge on the same engine. Offering alternative backend, as Go Ethereum, Lighthouse, Reth, and Teku already do, provides a practical fallback if a primary store is compromised by a bug or attack, and wider adoption of such options would further reduce shared storage risk.

Storage diversity is real despite tight constraints. Only a handful of embedded engines can sustain the write throughput and data volumes a system the scale of Ethereum requires, yet the 11 clients still field eight such engines from two on-disk families. That diversity is not automatic, Go Ethereum and Prysm choose different engines despite sharing a language, but it remains fragile wherever clients converge: LevelDB alone is supported by three of the six consensus clients, and RocksDB and MDBX recur similarly.

4.4. Overview

Ethereum draws its software supply chain almost entirely from open-source code developed outside the blockchain community and for purposes unrelated to it. The 11 clients span six programming languages, eight embedded database engines from two on-disk families, and thousands of third-party libraries, none of which were written specifically for Ethereum. This breadth is the primary finding: language diversity translates directly into supply-chain diversity because different ecosystems have different package pools, compilers, and toolchains.

Cross-ecosystem overlap is confined to a small set of cryptographic and networking primitives—`blst`, KZG implementations, `libsecp256k1`, and libp2p—that recur through bindings and parallel ports. Client teams treat these as deliberate exceptions: they knowingly accept a shared implementation where correctness is critical and collectively fund its auditing, as the `blst` case illustrates.

Within a single language ecosystem, overlap is higher. The three Go clients share roughly 14% of their modules, with Prysm's direct dependence on Go Ethereum importing nearly the entire Go Ethereum module set into a consensus client; the Rust clients share roughly 27% of their crates; and Besu and Teku share roughly 59% of their Maven artifacts. Yet convergence is not inevitable even within one language: Go Ethereum defaults to Pebble while Prysm relies on bbolt, Nethermind assembles an independent C# stack, and Nimbus Eth2 builds most of its core functionality on Nim libraries maintained in-house. Erigon, originally a fork of Go Ethereum, has diverged enough that the only Go modules it still shares with Go Ethereum are also present in Prysm, suggesting they reflect the Go ecosystem baseline rather than a residual coupling.

Our findings also suggest concrete actions for the 11 teams. On technology choices, all clients should ship at least one alternative backend, as Go Ethereum, Lighthouse, Reth, and Teku already do. This provides a fallback if a primary store is compromised, and new clients can decide which engine is already dominant among their peers.

On reuse, a direct dependency of one client on another, such as Prysm on Go Ethereum or Teku on the `org.hyperledger.besu` native modules, should be a diversity decision rather

than a convenience: extracting the narrow set of protocol utilities actually needed avoids importing an entire client's transitive tree, and thereby avoids convergence in the execution and consensus layers.

On governance, the monitoring infrastructure that already tracks client distribution could be extended below the product level: publishing an SBOM per release, reviewing new dependencies against what sibling clients already use, and maintaining shared funding for auditing the primitives that are deliberately common would make convergence visible while it is still a choice.

Client diversity is not spontaneous: it is an active choice that must be maintained. For every feature of a client, teams must consciously adopt different solutions rather than converge by default, and the same discipline applies when importing third-party libraries, because shared dependencies can quietly undermine the independence that client diversity is meant to provide.

4.5. Conclusion

This chapter presented the first systematic analysis of the software supply chain of all 11 major Ethereum clients, covering third-party dependencies and database backends. The 11 clients are written in six programming languages, depend on thousands of third-party libraries, and use eight embedded database engines—all developed outside the Ethereum community and for purposes unrelated to blockchain. Ethereum thus achieves software diversity largely by leveraging the natural diversity already present in the open-source ecosystem.

Language diversity is the strongest driver of supply-chain diversity. Each language ecosystem has its own package pool, compilers, and toolchains, clients in different languages share almost no dependencies by package identity. Cross-ecosystem overlap is confined to a small, deliberate set of cryptographic and networking primitives—`blst`, KZG, `libsecp256k1`, and libp2p—where correctness is paramount and the community collectively funds auditing.

Within a single language, overlap is higher but not uniform. In Go, 14% of modules are common to all three clients, yet Go Ethereum and Prysm choose different storage engines. Furthermore, although Erigon forked from Go Ethereum, it has diverged to the point that their residual overlap matches the Go ecosystem baseline. In Rust, 27% of crates are shared, concentrated in the `tokio` runtime, cryptographic primitives, and Ethereum-specific serialization. In Java, Besu and Teku share 59% of their Maven artifacts, the highest within-language overlap, compounded by shared ownership at Consensys and overlapping contributor pools.

For storage, eight engines from two on-disk families (LSM-tree and B+tree) back the 11 clients. Diversity exists even under tight performance constraints: Go Ethereum defaults to

Pebble, Prysm to bbolt, and Erigon to MDBX. However, software diversity is fragile where clients converge: LevelDB is supported by four clients in different languages, and RocksDB and MDBX recur similarly.

These findings confirm that Ethereum's multi-language strategy is effective at producing software supply-chain diversity, but that diversity is not guaranteed within a shared ecosystem and must be actively maintained.

Chapter 5

Discussion and Conclusion

Ethereum sustains 11 independent, hand-written implementations of one specification, maintained for over a decade, without any owner commissioning them or any authority certifying their independence. This is a new model of software diversity that has not been studied before: *decentralized software diversity*. In our research we find that Ethereum adopts the multiple implementations of commissioned software diversity and the reuse of natural diversity, then replaces the central owner with a specification, distributed funding, public monitoring, and protocol penalties. We then turn these findings into recommendations for client teams and operators, and discuss how AI-assisted development changes the cost of producing further independent implementations.

5.1. A Fourth Model of Software Diversity

The literature reviewed in chapter 1 offers three ways to obtain software diversity, and Ethereum fits none of them. Commissioned design diversity, as practiced in avionics and nuclear instrumentation, produces genuinely independent designs but requires a single owner able to pay N times for one function and a certifying authority to verify that the teams worked apart [**4, 75, 10**]. Automated diversification is cheap enough to apply at internet scale, but it derives every variant from one source and therefore diversifies representation rather than design [**35, 21**]. Natural diversity is design diversity obtained for free from open source, but the party that benefits from it can only select among the variants that already exist; it cannot steer them, and nothing guarantees that a variant will still be maintained next year [**6, 36**].

Ethereum does not reapply any of these models. It takes from commissioned diversity the practice of writing independent implementations of a shared specification by hand. It also takes from natural diversity the reuse of open-source components that were never written

for blockchain. Ethereum removes the central owner, replacing it with a set of mechanisms that keep those two practices running without one.

Table 5.1 compares this new fourth model, decentralized software diversity, with the three documented in the literature.

Software Diversity Model	Financial Model	Software Variants Independence	Software Diversity Sustainability
Commissioned software diversity	A single owner, N times	Enforced isolation of teams, verified by a certifying authority	Regulation and safety standards
Automated software diversity	Almost nothing per deployment	Guaranteed by construction, but limited to the same design	Build and deployment tooling
Natural software diversity	Someone else, for their own reasons	A by-product of how open source is produced	Nothing in particular; the model reuses what exists
Decentralized software diversity	Many independent stakeholders, none of them in charge	A shared executable specification and public conformance tests	Monitoring of the deployed distribution, persuasion, funding, and penalties that scale with correlated failure

Table 5.1. Four models for obtaining software diversity: the three models presented in chapter 1 and the decentralized model this thesis identifies in Ethereum.

The interviews in chapter 3 show that the community replaces the commissioning authority with a specification, distributed funding, public monitoring, and protocol penalties. Client teams implement from the specification and a large conformance suite, coordinate upgrades in public all-core-devs meetings, and avoid copying another client's code: *"Client teams do exchange information among each other but take care not to copy each other's implementations. Copying implementations, even in a different programming language, would defeat the purpose of diversity"* (P5, Table 3.1). When one client approaches a supermajority, operators are persuaded to migrate rather than ordered to do so, using public client distribution data and documented migration procedures [**52, 67**]. The community understands the risk of client concentration becoming a single point of failure and acts on it.

chapter 4 shows that the 11 clients rely on open-source code developed outside the Ethereum community and for other purposes: eight embedded database engines and thousands of third-party libraries. Language choice has a great influence on software supply chain diversity. Six languages imply six compilers, six package ecosystems, and almost no overlap by package identity across all clients.

Overlap grows when clients share an ecosystem. The Go clients share 14% of modules, the Rust clients 27% of crates, and the two Java clients 59% of Maven artifacts [**71**]. However, even within a language, divergence is also possible: after forking from Go Ethereum, Erigon now shares with it only the Go modules that Prysm also needs, so their residual overlap is the language baseline rather than a leftover coupling.

Not every shared component is a failure of the model. Every client reaches the same BLS assembly through `blst` or a binding; the teams know it and funded its audit rather than implementing alternatives. The same holds for KZG and for `libsecp256k1`. This decision balances trade-offs: concentrate when independent correct implementation is hardest and more audits are needed; diversify when independent teams are likely to make uncorrelated mistakes. Cryptographic primitives fall on the first side; storage engines, networking stacks, and client architecture fall on the second. Where that choice to converge is not made explicitly and funded, it is a single point of failure.

Specification complexity sets a second bound. Every upgrade must be implemented 11 times, so protocol complexity falls hardest on the smallest teams, the ones that provide the diversity. A system's complexity therefore limits how many independent implementations it can sustain [**13**]. The specification is not a neutral document: how precisely it defines behavior determines both whether implementations can be equivalent and whether they can still differ.

5.2. Recommendations for Better Software Diversity Practices for Ethereum

Client teams decide what code is written and shared while operators decide how the diversity that already exists is deployed.

5.2.1. Recommendations for Ethereum client developers

Client teams should publish a software bill of materials (SBOM) with every release. The analysis in chapter 4 had to reconstruct dependency trees by hand. An SBOM would remove that burden and let each team automatically check whether its dependencies overlap with those of other clients. Convergence would then be a decision the team takes knowingly.

Teams should keep imports minimal. Pulling in a complete library to use a small part of it extends the shared surface with code the client never executes.

Every client should ship at least one alternative storage backend, as Go Ethereum, Lighthouse, Reth, and Teku already do, and should avoid defaulting to the engine that is most popular, LevelDB in particular.

5.2.2. Recommendations for Ethereum Client Operators

Operators can add a layer of diversity that no single client team can provide by using multi-node validator clients such as Vero or Vouch. These tools run multiple consensus clients simultaneously and compare their results, accepting only the answer on which the majority agrees. If a bug or attack affects one client, its incorrect result is outvoted by the others, protecting the operator's validator from penalties [**67**].

5.3. Decentralized Software Diversity in the Era of AI

Commissioned software diversity's two obstacles are a central authority and the cost of paying N teams for one function. Ethereum removes the first. AI-assisted development is now acting on the second. Peng et al. measured a 55.8% reduction in task-completion time for developers using GitHub Copilot [**59**], and the same tools apply to writing an independent implementation from a specification.

If producing an independent implementation becomes substantially cheaper, the remaining challenge shifts from cost to independence: an AI-generated implementation is only useful for diversity if it fails on different inputs than the others. Ron et al. developed Galapagos, a tool that uses large language models to generate, verify, and bundle equivalent program variants into N-version components [**65**]. The cross-language variants behaved identically while maintaining distinct compiled structures and runtime execution paths. Ron, Baudry, and Monperrus then showed that AI coding agents exhibit highly correlated failure modes, yet majority voting across diverse systems, models, and languages still reduces overall failure rates [**64**]. AI therefore cuts the cost of producing multiple implementations, and even without full independence, ensembles of generated versions can improve reliability.

5.4. Conclusion

This thesis asked how a system with no central authority adopts and maintains software diversity, and how far that diversity reaches into the code. We used a mixed methodology: semi-structured interviews with seven people who build, coordinate, operate, and monitor

Ethereum client diversity. We also conducted a systematic analysis of the software supply chain of all 11 execution and consensus clients.

Ethereum does not copy an existing software diversity model. It takes the implementation independence of commissioned N-version programming and the reuse of natural open-source diversity. Then, it replaces the central owner with an executable specification and conformance tests, distributed funding, public monitoring, and penalties that grow with correlated failure. The interviews show that the community understands this arrangement and maintains it: teams implement from the specification rather than from one another's code, coordinate upgrades in public, and intervene when one client concentrates too much of the network.

The code analysis shows that this strategy extends beyond language choice. The 11 clients reuse a vast open-source supply chain that was not written for blockchain: eight embedded database engines and thousands of third-party libraries across six language ecosystems. Cross-ecosystem overlap is limited. Within a language, overlap is higher—14% of Go modules, 27% of Rust crates, 59% of Java artifacts—yet, even there, divergence is possible, as Erigon and Go Ethereum demonstrate. Part of the remaining concentration is deliberate: the ecosystem knowingly shares `blst`, KZG, and `libsecp256k1`, and funds their audit.

These results describe a fourth model of software diversity, which we call decentralized software diversity: based on independent implementations of a shared specification written by hand, leverages the natural software diversity of open-source components, and is sustained without a central authority.

References

[1] *The Merge | ethereum.org — ethereum.org*, `https://ethereum.org/roadmap/merge/`, [Accessed 28-01-2026].

[2] *Mainnet postmortems | Prysm — prysm.offchainlabs.com*, `https://prysm.offchainlabs.com/docs/misc/mainnet-postmortems/`, December 2025, [Accessed 17-07-2026].

[3] *GitHub - ethereum/execution-specs: Specification for the Execution Layer. Tracking network upgrades. — github.com*, `https://github.com/ethereum/execution-specs`, 2026, Accessed: 2026-01-13.

[4] A. Avizienis, *The N-Version Approach to Fault-Tolerant Software*, IEEE Transactions on Software Engineering **SE-11** (1985), no. 12, 1491–1501.

[5] ———, *Toward Systematic Design of Fault-Tolerant Systems*, Computer **30** (1997), no. 4, 51–58.

[6] Benoit Baudry et Martin Monperrus, *The Multiple Facets of Software Diversity: Recent Developments in Year 2000 and Beyond*, ACM Computing Surveys (CSUR) **48** (2015), no. 1, 1–26.

[7] Rudolf Bayer et Edward McCreight, *Organization and Maintenance of Large Ordered Indexes*, Acta Informatica **1** (1972), no. 3, 173–189.

[8] T. Beik, *Ethereum Protocol Upgrades: Past, Present & Future*, `https://www.youtube.com/watch?v=HoclxIBR2EM`, 2023, Presented at The Ethereum Community Conference (EthCC). Accessed: 22 April 2026.

[9] Christopher Berner, *redb: A Simple, Portable, High-Performance, ACID, Embedded Key-Value Store in Rust*, `https://github.com/cberner/redb`, 2026, Accessed: 2026-01-13.

[10] Peter G. Bishop, David G. Esp, Mel Barnes, Peter Humphreys, Gustav Dahll, et Jaakko Lahti, *PODS — A Project on Diverse Software*, IEEE Transactions on Software Engineering **SE-12** (1986), no. 9, 929–940.

[11] William Bugden et Ayman Alahmar, *Rust: The Programming Language for Safety and Performance*, 2022.

[12] Vitalik Buterin, *How will Ethereum's Multi-Client Philosophy Interact with ZK-EVMs?*, `https://vitalik.eth.limo/general/2023/03/31/zkmulticlient.html`, March 2023, Accessed: 2026-08-18.

[13] ———, *Simplifying the L1*, `https://vitalik.eth.limo/general/2025/05/03/simplel1.html`, May 2025, Accessed: 2026-08-18.

[14] Vitalik Buterin et al., *Ethereum White Paper*, GitHub repository (2013), no. 22-23, 5–7.

[15] Vitalik Buterin et Virgil Griffith, *Casper the Friendly Finality Gadget*.

[16] Javier Cabrera-Arteaga, Nicholas Fitzgerald, Martin Monperrus, et Benoit Baudry, *Wasm-Mutate: Fast and Effective Binary Diversification for WebAssembly*, Computers Security **139** (2024), 103731.

[17] Gui-lin Cai, Bao-sheng Wang, Wei Hu, et Tian-zuo Wang, *Moving Target Defense: State of the Art and Characteristics*, Frontiers of Information Technology & Electronic Engineering **17** (2016), no. 11, 1122–1153.

[18] Ting Chen, Zihao Li, Yuxiao Zhu, Jiachi Chen, Xiapu Luo, John Chi-Shing Lui, Xiaodong Lin, et Xiaosong Zhang, *Understanding Ethereum via Graph Analysis*, ACM Trans. Internet Technol. **20** (2020), no. 2.

[19] Jin-Hee Cho, Dilli P Sharma, Hooman Alavizadeh, Seunghyun Yoon, Noam Ben-Asher, Terrence J Moore, Dong Seong Kim, Hyuk Lim, et Frederica F Nelson, *Toward proactive, adaptive defense: A survey on moving target defense*, IEEE Communications Surveys & Tutorials **22** (2020), no. 1, 709–745.

[20] Howard Chu, *MDB: A Memory-Mapped Database and Backend for OpenLDAP*, Proceedings of the 3rd International Conference on LDAP (LDAPCon), 2011.

[21] Frederick B. Cohen, *Operating System Rrotection through Program Evolution*, Comput. Secur. **12** (1993), no. 6, 565–584.

[22] Douglas Comer, *The ubiquitous B-tree*, ACM Computing Surveys **11** (1979), no. 2, 121–137.

[23] Bart Coppens, Bjorn De Sutter, et Stijn Volckaert, *Multi-variant execution environments*, The Continuing Arms Race: Code-Reuse Attacks and Defenses, 2018, pp. 211–258.

[24] John W. Creswell, *Chapter 18 - Mixed-Method Research: Introduction and Application*, Handbook of Educational Policy (Gregory J. Cizek, ed.), Academic Press, 1999, pp. 455–472.

[25] Siying Dong, Andrew Kryczka, Yanqin Jin, et Michael Stumm, *RocksDB: Evolution of Development Priorities in a Key-Value Store Serving Large-Scale Applications*, ACM Transactions on Storage **17** (2021), no. 4, 1–32.

[26] Alan AA Donovan et Brian W Kernighan, *The Go programming language*, Addison-Wesley Professional, 2015.

[27] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, et Daniela Damian, Selecting Empirical Methods for Software Engineering Research, 285–311, 01 2008, pp. 285–311.

[28] etcd-io, *bbolt: An Embedded Key/Value Database for Go*, `https://github.com/etcd-io/bbolt`, 2026, Accessed: 2026-01-13.

[29] Ethereum Foundation, *Announcing the Client Incentive Program*, `https://blog.ethereum.org/2021/12/13/client-incentive-program`, 12 2021, Accessed: 2026-08-20.

[30] ———, *Ethereum Foundation Report 2024*, `https://ethereum.foundation/report-2024.pdf`, 2024, Accessed: 2026-08-20.

[31] ———, *Ethereum Consensus Clients: Prysm, Lighthouse, Teku, Nimbus, Lodestar*, `https://ethereum.org/en/developers/docs/nodes-and-clients/#consensus-clients`, 2026, Accessed: 2026-01-13.

[32] ———, *Ethereum Execution Clients: Geth, Nethermind, Besu, Erigon*, `https://ethereum.org/en/developers/docs/nodes-and-clients/#execution-clients`, 2026, Accessed: 2026-01-13.

[33] ———, *Ethereum Proof-of-Stake: Consensus Specifications (Beacon Chain)*, `https://github.com/ethereum/consensus-specs`, 2026, Accessed: 2026-01-13.

[34] Bent Flyvbjerg, *Five Misunderstandings About Case-Study Research*, Qualitative Inquiry **12** (2006), no. 2, 219–245.

[35] S. Forrest, A. Somayaji, et D.H. Ackley, *Building diverse computer systems*, Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No.97TB100133), 1997, pp. 67–72.

[36] Ilir Gashi, Peter Popov, et Lorenzo Strigini, *Fault Tolerance via Diversity for Off-the-Shelf Products: A Study with SQL Database Servers*, IEEE Transactions on Dependable and Secure Computing **4** (2007), no. 4, 280–294.

[37] Go Ethereum Team, *Go Ethereum (Geth) Documentation*, `https://geth.ethereum.org/docs`, 2026, Accessed: 2026-01-13.

[38] Google, *LevelDB: A Fast Key-Value Storage Library*, `https://github.com/google/leveldb`, 2026, Accessed: 2026-01-13.

[39] Dominic Grandjean, Lioba Heimbach, et Roger Wattenhofer, *Ethereum Proof-of-Stake Consensus Layer: Participation and Decentralization*, Financial Cryptography and Data Security. FC 2024 International Workshops, Springer Nature Switzerland, 2025, pp. 253–280.

[40] Nicolas Harrand, Thomas Durieux, David Broman, et Benoit Baudry, *The behavioral diversity of java json libraries*, 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE), IEEE, 2021, pp. 412–422.

[41] James E. Just et Mark Cornwell, *Review and Analysis of Synthetic Diversity for Breaking Monocultures*, , WORM '04, Association for Computing Machinery, 2004, p. 23–32.

[42] John C. Knight, Diversity, 298–312, Springer-Verlag, 2011, p. 298–312.

[43] John C. Knight et Nancy G. Leveson, *An experimental evaluation of the assumption of independence in multiversion programming*, IEEE Transactions on Software Engineering **SE-12** (1986), no. 1, 96–109.

[44] Koen Koning, Herbert Bos, et Cristiano Giuffrida, *Secure and Efficient Multi-Variant Execution Using Hardware-Assisted Process Virtualization*, 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2016, pp. 431–442.

[45] Georgios Konstantopoulos, *Introducing Reth*, `https://www.paradigm.xyz/writing/reth?utm_source=chatgpt.com`, 2022, [Accessed 28-07-2026].

[46] Philip Koopman et John DeVale, *Comparing the Robustness of POSIX Operating Systems*, Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (FTCS-29), 1999, pp. 30–37.

[47] Moez Krichen, Mehdi Ammi, Aymen Mihoub, et Mansour Almutiq, *Blockchain for Modern Applications: A Survey*, Sensors (2022).

[48] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, et Olivier Barais, *SoK: Taxonomy of Attacks on Open-Source Software Supply Chains*, 2023 IEEE Symposium on Security and Privacy (SP), 2023, pp. 1509–1526.

[49] Jean-Claude Laprie, Karama Kanoun, et Yves Deswarte, *Diversity against Accidental and Deliberate Faults* , Computer Security, Dependability, and Assurance, IEEE Computer Society, July 1998, p. 171.

[50] Cheng Lei, Hong-Qi Zhang, Jing-Lei Tan, Yu-Chen Zhang, et Xiao-Hu Liu, *Moving Target Defense Techniques: A Aurvey*, Security and Communication Networks **2018** (2018), no. 1, 3759626.

[51] Chen Luo et Michael J. Carey, *LSM-based storage techniques: a survey*, The VLDB Journal **29** (2020), no. 1, 393–418.

[52] Jiahao Luo, *Unveiling Ethereum's P2P Network: The Role of Chain and Client Diversity*, arXiv preprint arXiv:2501.16236 (2025).

[53] Peter Mattis, *Introducing Pebble: A RocksDB-inspired key-value store written in Go — cockroachlabs.com*, `https://www.cockroachlabs.com/blog/pebble-rocksdb-kv-store/`, 2020, [Accessed 11-08-2026].

[54] Satoshi Nakamoto, *Bitcoin: A Peer-to-Peer Electronic Cash System*.

[55] Damilare Peter Oyinloye, Je Sen Teh, Norziana Jamil, et Moatsum Alawida, *Blockchain Consensus: An Overview of Alternative Protocols*, Symmetry (2021).

[56] Patrick O'Neil, Edward Cheng, Dieter Gawlick, et Elizabeth O'Neil, *The log-structured merge-tree (LSM-tree)*, Acta informatica **33** (1996), no. 4, 351–385.

[57] Bradley Peak, *What is Finality in Blockchain and Why It Matters — cointelegraph.com*, `https://cointelegraph.com/explained/what-is-finality-in-blockchain-and-why-does-it-matter`, 2025, [Accessed 28-01-2026].

[58] Sean Peisert, Bruce Schneier, Hamed Okhravi, Fabio Massacci, Terry Benzel, Carl Landwehr, Mohammad Mannan, Jelena Mirkovic, Atul Prakash, et James Bret Michael, *Perspectives on the SolarWinds Incident*, IEEE Security Privacy **19** (2021), no. 2, 7–13.

[59] Sida Peng, Eirini Kalliamvakou, Peter Cihon, et Mert Demirer, *The Impact of AI on Developer Productivity: Evidence from GitHub Copilot*, 2023.

[60] Potuz, *History of a Buggy Bug*, `https://www.potuz.net/posts/fulu-bug/`, December 2025, Accessed July 2026.

[61] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine Mcleavey, et Ilya Sutskever, *Robust Speech Recognition via Large-Scale Weak Supervision*, Proceedings of the 40th International Conference on Machine Learning, Proceedings of Machine Learning Research, vol. 202, PMLR, 23–29 Jul 2023, pp. 28492–28518.

[62] B. Randell, *System Structure for Software Fault Tolerance*, Proceedings of the International Conference on Reliable Software, 1975, p. 437–449.

[63] Pooja Ranjan, *Exploring Ethereum's client ecosystem*, Mar 2023.

[64] Javier Ron, Benoit Baudry, et Martin Monperrus, *N-Version Programming with Coding Agents*, 2026.

[65] Javier Ron, Diogo Gaspar, Javier Cabrera-Arteaga, Benoit Baudry, et Martin Monperrus, *Galápagos: Automated N-Version Programming with LLMs*, ACM Transactions on Software Engineering and Methodology (2025), arXiv:2408.09536.

[66] Javier Ron, Zheyuan He, et Martin Monperrus, *Proving and Rewarding Client Diversity to Strengthen Resilience of Blockchain Networks*, 2024.

[67] Javier Ron, César Soto-Valero, Long Zhang, Benoit Baudry, et Martin Monperrus, *Highly Available Blockchain Nodes With N-Version Design*, IEEE Transactions on Dependable and Secure Computing **21** (2024), no. 4, 4084–4097.

[68] Danny Ryan, *The State of Eth2, June 2020*, `https://blog.ethereum.org/2020/06/02/the-state-of-eth2-june-2020`, June 2020, Ethereum Foundation Blog. Accessed: 2026-08-18.

[69] F. Saglietti et W. Ehrenberger, *Software Diversity—Some Considerations About its Benefits and its Limitations*, IFAC Proceedings Volumes **19** (1986), no. 11, 27–34.

[70] Cesar Soto-Valero, Amine Benelallam, Nicolas Harrand, Olivier Barais, et Benoit Baudry, *The Emergence of Software Diversity in Maven Central*, 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), IEEE, May 2019, p. 333–343.

[71] Cesar Soto-Valero, Martin Monperrus, et Benoit Baudry, *The Multibillion Dollar Software Supply Chain of Ethereum* , Computer **55** (2022), no. 10, 26–34.

[72] SQLite Consortium, *SQLite: About and Documentation*, `https://www.sqlite.org/about.html`, 2026, Accessed: 2026-01-13.

[73] Karim Sultan, Umar Ruhi, et Rubina Lakhani, *Conceptualizing Blockchains: Characteristics Applications*, 2018.

[74] Sergen Uysal, *Ethereum Clients Explained: Execution, Consensus & Node Types | Quicknode Guides — quicknode.com*, `https://www.quicknode.com/guides/ethereum-development/getting-started/ethereum-clients-explained`, 2025, [Accessed 27-01-2026].

[75] Udo Voges, *Software Diversity*, Reliability Engineering System Safety **43** (1994), no. 2, 103–110, Special Issue on Software Safety.

[76] Stijn Volckaert, Bart Coppens, Bjorn De Sutter, Koen De Bosschere, Per Larsen, et Michael Franz, *Taming Parallelism in a Multi-Variant Execution Environment*, Proceedings of the Twelfth European Conference on Computer Systems, 2017, pp. 270–285.

[77] Taotao Wang, Chonghe Zhao, Qing Yang, Shengli Zhang, et Soung Chang Liew, *Ethna: Analyzing the underlying peer-to-peer network of ethereum blockchain*, IEEE Transactions on Network Science and Engineering **8** (2021), no. 3, 2131–2146.

[78] Gavin Wood et al., *Ethereum: A secure decentralised generalised transaction ledger*, Ethereum project yellow paper **151** (2014), no. 2014, 1–32.

[79] Leonid Yuriev, *libmdbx: One of the Fastest Embeddable Key-Value ACID Databases*, `https://gitflic.ru/project/erthink/libmdbx`, 2026, Accessed: 2026-01-13.

[80] Jianjun Zheng et Akbar Siami Namin, *A Survey on the Moving Target Defense Strategies: An Architectural Perspective*, Journal of Computer Science and Technology **34** (2019), no. 1, 207–233.