COMPOSE

Compose Whitepaper

A Composition Layer for On-Chain Applications

Abstract

The smart-contract ecosystem already depends heavily on code reuse, but that reuse has not yet become shared on-chain infrastructure. Similar logic is repeatedly redeployed across projects, creating duplicated infrastructure that increases long-term maintenance, audit, and security burdens for the ecosystem.

Compose introduces Smart Contract Oriented Programming, or SCOP, to bring reuse into the deployed architecture itself. Using diamonds and stateless facets, Compose enables modular on-chain systems that are easier to build and trust across their lifecycle.

---

1. Introduction

Blockchain systems are reshaping how value, ownership, and coordination can operate on digital infrastructure. While their applications differ across industries, they share a common objective: to create decentralized systems that are programmable, trust-minimized and secure.

Modern on-chain applications, however, are rarely built from entirely isolated logic. They increasingly rely on shared standards, external libraries and reusable code patterns. An empirical study of more than 350,000 Solidity smart contracts found that a typical smart-contract project contains multiple contracts, with more than 80% originating from external sources [1, p. 1]. This finding shows how deeply Solidity development already depends on shared components and reusable patterns.

As smart-contract systems grow in size and functionality, code reuse becomes a development best practice as much as an ecosystem burden. Similar standards and contract logic are repeatedly redeployed across new applications. Each redeployment places duplicate infrastructure on-chain, consumes gas, and creates another instance that must be audited. In practice, the ecosystem often pays to publish logic that already exists elsewhere because reusable on-chain components are not yet standardized enough to be trusted and used broadly.

To address this gap, Compose introduces Smart Contract Oriented Programming, or SCOP: a development paradigm for building modular on-chain applications as composed systems rather than isolated deployments. SCOP treats reusable logic, application identity, shared state, and lifecycle management as explicit parts of the system design. In this model, Compose is the practical composition layer that makes SCOP usable through diamond-native architecture, where diamonds provide the application container, facets provide reusable deployed functionality, and modules support storage access, initialization, and internal extension helpers.

This whitepaper presents Compose as a composition layer for on-chain applications and explains how SCOP can make smart-contract systems more modular, reusable, auditable, and easier to reason about as the ecosystem continues to mature.

2. From Programs to Systems

The first generation of smart-contract development treated contracts primarily as individual programs deployed on-chain. Each contract has its own bytecode, address, storage, and public interface. This model is sufficient when applications are smaller and simpler to reason about in isolation.

Today, on-chain applications are different. Contracts routinely call other contracts, create new ones, and route logic across multiple deployed components. One large-scale study of Ethereum found that multi-contract transactions became increasingly common over time, with almost one-third of transactions involving at least two contracts in 2022 [2, p. 9]. This suggests that on-chain applications are more interconnected, and that system-level interaction is now a normal part of the Ethereum application layer.

This shift changes how smart-contract applications should be understood. The contract remains the core execution unit, but it is no longer always the complete system design. What matters is not only what each contract does, but how the parts fit together.

As a result, the main transition is from contract-level reasoning to system-level reasoning. Developers need to understand which components define the whole application, which on-chain primitives can be reused and how the system can evolve safely over time.

3. The Composition Problem

Developers commonly reuse standards, import libraries, copy templates, or fork repositories. These practices reduce development time, but most reuse still happens before deployment. Once an application is compiled and deployed, familiar logic often reappears on-chain as a new contract instance, with its own address, bytecode, maintenance surface, and audit requirements.

The security implications are also significant. A large-scale study of Ethereum code clones found that more than 96% of contracts in its dataset had duplicates, and that roughly 9.7% of similar contract pairs shared exactly the same vulnerabilities [3, p. 3]. This suggests that copy-based reuse can spread risk across the ecosystem. When vulnerable logic is copied

into many independent deployments, remediation becomes fragmented. Each affected project must be identified and handled separately.

This is not only a vulnerability problem. It is a knowledge gap. When logic is repeatedly redeployed, the ecosystem loses a clear relationship between the original code and the applications that depend on it.

Traditional software ecosystems provide a useful contrast. Developers in other ecosystems can rely on mature package managers to discover reusable code modules, track versions, manage dependencies, and integrate them into their new applications. Smart-contract development has source-code libraries, but on-chain logic is rarely treated as shared infrastructure.

This leaves the Solidity application ecosystem with an important gap. Reuse exists, but it is not yet organized around deployed systems. As a result, developers continue to rebuild similar features instead of building from a shared base of verified on-chain components.

4. Defining Smart Contract Oriented Programming

Many common smart-contract development practices borrow heavily from traditional software engineering. Developers use familiar principles such as modularity, abstraction, reuse, and separation of concerns to structure their contracts and reduce duplicated work.

These principles remain valuable, but the question is whether they have been adapted correctly to the assumptions of on-chain applications.

Traditional applications are usually built under a different execution model. A command-line tool, backend service, or Web2 application can be patched, replaced, or shut down when it becomes obsolete or unsafe. Its runtime environment is controlled by its operator, its internal infrastructure, and business logic are not normally exposed for arbitrary public execution.

On-chain applications operate under a different set of assumptions.

Smart contracts are persistent on-chain artifacts. Once deployed, their code, address, state, and interfaces become part of a public execution environment. Even when a system is designed to be upgradeable, the deployed artifacts themselves remain visible, callable, and part of the historical record.

Smart contracts are also shared by default. Their bytecode can be inspected, their public interfaces can be called by anyone, and their behavior is executed by a distributed network rather than by a private operator. This means they are constrained not only by the language used to write them, but also by the Ethereum Virtual Machine (EVM), gas costs, consensus rules, and the network on which they are deployed.

Because smart contracts can hold value and coordinate permissions, failures can have serious consequences. A bug, storage collision, or unsafe upgrade path can affect not only the application itself, but also users, integrators, and other contracts that depend on it.

Smart Contract Oriented Programming, or **SCOP**, addresses these constraints by adapting established software-engineering principles to the execution model of the EVM and the lifecycle of deployed on-chain systems.

Stateless logic components

In SCOP, application logic is separated into explicit, stateless components. These components do not own the application state directly. Instead, they operate over the state of the composed system. This makes reusable logic easier to decouple from project-specific behavior. A transfer facet, access-control facet, inspection facet, or other reusable logic component can be designed as shared infrastructure rather than being permanently embedded inside one project's custom contract.

Code written for understanding

SCOP treats readability as part of system design, not only as a coding style preference. Smart-contract systems are long-lived, public, and often depend on verified source code for auditability and trust. Code should therefore be written so that developers, auditors, and future maintainers can understand what each component does and how it fits into the larger system. Clear code improves confidence, reduces the likelihood of bugs, and makes long-term operation safer.

Native on-chain composition

Smart contracts already exist inside a shared execution environment. SCOP uses this property directly by treating deployed contracts as infrastructure that can be referenced, executed, and composed on-chain. Composition does not need to depend on an off-chain backend or trusted coordinator when the system's interfaces and execution paths are part of the on-chain architecture itself.

Composability & Reuse by design

SCOP treats reuse as an architectural property of deployed components, not only as a source-code practice before deployment. Components should be designed with clear interfaces, predictable boundaries, and assumptions that make them easier to verify and reuse across applications. When components are built this way, they can become shared on-chain infrastructure rather than project-specific copies.

Independent System Evolution

When components are separated by clear boundaries, unrelated parts of the system can evolve independently. New functionality can be added, existing functionality can be replaced, and unused functionality can be removed without forcing the entire application to be redeployed or redesigned. This makes change more local, auditable, and easier to reason about.

Upgradeable or Immutable by Design

SCOP does not require every application to remain upgradeable forever. Some systems need upgradeability during active development or governance-controlled operation. Others need immutability from the start, or after the system has matured. A SCOP-based architecture can support both models, allowing a system to remain flexible where needed and become immutable when stability and trust minimization become more important.

In this sense, SCOP defines a programming model shaped by deployed code, shared execution, and long-term trust, rather than assumptions imported directly from traditional software.

5. Technical Foundation & Trade-Offs

Compose makes SCOP practical by building on diamond-native architecture originally defined by ERC-2535 and further refined by ERC-8153. [4] [5]

In this smart contract paradigm, an application is not treated as one monolithic contract that contains all of its logic. Instead, it is organized as a composed system: a diamond proxy provides the stable contract address and owns the application state, while stateless facets provide focused units of functionality that can be added, replaced, removed, and reused depending on the needs of the application.

Although the system is internally composed of multiple facets, it still behaves externally like a single contract. Users interact with one address, one externally callable surface, and one selector space. This means duplicated selectors cannot coexist in the same diamond: each external function selector must resolve to a single implementation at any given time. When desired, the application can also become immutable by removing the upgrade mechanism while keeping other administrative features, such as ownership or access control, intact.

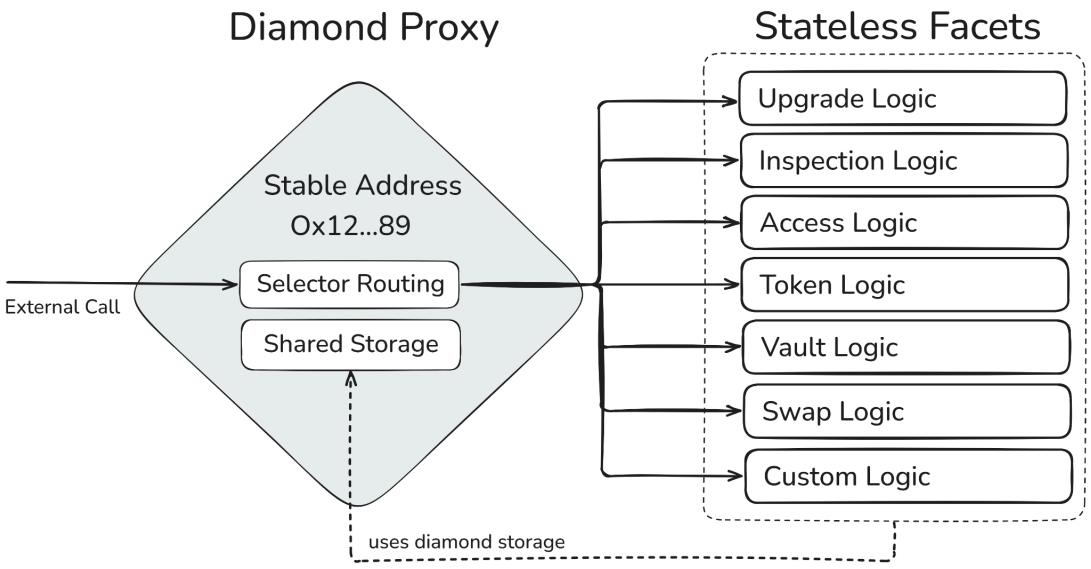

Figure 1: Diamond-native composition model: identity, logic, and state

This architecture separates three concerns that are often tightly coupled in traditional smart-contract deployments: identity, logic, and state.

The diamond proxy gives the application a stable on-chain identity and routes external calls by function selector. Facets provide reusable logic components. Storage domains define how shared state is organized and accessed across the composed system. Together, these parts allow a smart-contract application to grow without embedding every feature, standard, or extension into a single deployment.

Facets are the logic layer of the system. Each facet is a deployed contract that implements a focused set of external functions. In Compose, facets are intended to be small, readable, and self-contained. Instead of relying primarily on inheritance trees or project-specific copies of common code, developers can compose applications from pre-deployed facets, then add custom facets for the functionality unique to their system.

Selectors make this composition explicit. Each external function is identified by its selector, and the diamond maps selectors to the facets that currently own them. Adding, replacing, or removing functionality becomes a precise change to that mapping rather than a full redeployment of the application.

Because facets execute in the context of the diamond, they read and write the diamond's storage rather than their own. This makes storage discipline essential. Compose uses explicit storage locations, so components can share state intentionally or define independent storage domains when needed.

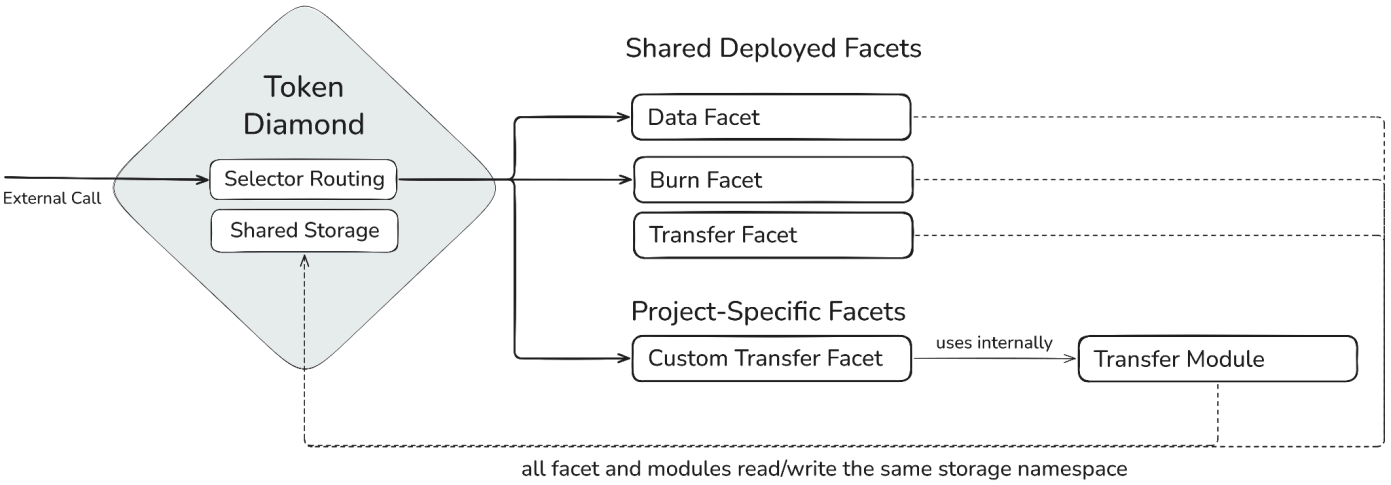

Figure 2: Facet and Module Interaction

Compose also distinguishes between deployed facets and internal modules. Facets provide complete external functionality that can be composed directly into a diamond. Modules act as a bridge between reusable facet infrastructure and project-specific logic: they provide internal helper code for deployment, initialization, storage access, and custom integrations. Unlike facets, modules are not installed into the diamond as externally callable components. Instead, they are used by custom facets. Because those facets execute in the diamond context, modules can help project-specific logic read and write the same explicit storage domains used by the composed system.

These benefits come with trade-offs. Diamond-based systems introduce an additional routing layer, and delegated execution has runtime overhead compared with logic implemented directly in a single contract. Diamond-based systems also require careful storage management, clear selector ownership, disciplined initialization, and well-governed upgrade authority.

Compose accepts these trade-offs by making the system structure explicit, so routing, selector ownership, storage access, and upgrade paths can be inspected rather than hidden inside a monolithic deployment.

6. Trusted Reuse & Auditability

For reusable on-chain logic to become shared infrastructure, it must be more than available logic components. It must be identifiable, verifiable, and understandable in the systems that use it. Compose treats trust as a property of explicit composition: developers and auditors should be able to see which components are used, what assumptions they make, how they are connected, and how the system has changed over time.

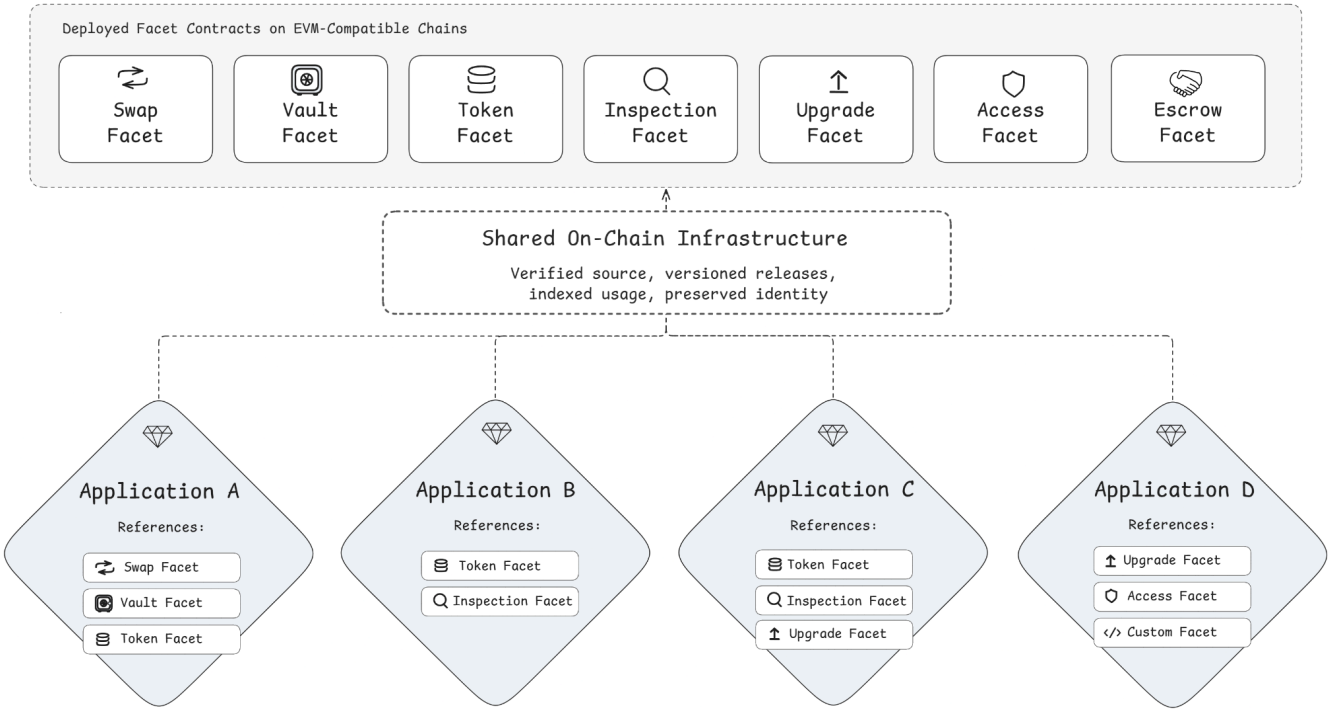

Figure 3: Trusted Reuse Through Shared Facet Infrastructure

A reusable component becomes more valuable when its identity is preserved across deployments and integrations. Its deployed address, verified source code, function selectors, version information, and known assumptions help establish what the component is and how it is expected to behave. This does not make the component automatically safe, but it gives developers and auditors a clearer basis for review.

Trust also depends on how components are assembled. A component may expose specific functions, rely on specific storage layout, require permissions, or depend on initialization steps. A safe component in one system can become unsafe in another if it is connected incorrectly or given the wrong authority. Clear boundaries make these assumptions easier to inspect before and after deployment.

Metadata and deployment history strengthen traceability by showing what a component is, where it is used, and how a system reached its current configuration. Neither replaces verification, but both provide important context for review.

This traceability also matters after deployment. If a reusable facet is later found to be vulnerable, preserving its identity can make it easier to identify which systems use it and which functions are affected. Events, indexers, and monitoring tools can help discover affected diamonds and support faster mitigation. This is much harder with pure source-code reuse, where similar logic may be redeployed many times as unrelated contract instances.

This visibility has a trade-off. If a pre-deployed facet is vulnerable, attackers could also be able to identify affected systems more quickly. Compose does not eliminate this risk. The same problem already exists with source-code reuse, where attackers can scan for contracts that contain a known vulnerable pattern. The difference is that explicit composition gives projects, auditors, and monitoring tools a clearer mechanism to detect affected systems, issue warnings, and coordinate response across the ecosystem.

This makes audits more structured without replacing them. A known component can still be misused, and a verified implementation can still introduce risk when composed with incompatible storage assumptions, unsafe permissions, incomplete initialization, or an unclear upgrade path. Composition improves auditability by making the system easier to reason about; it does not remove the need to evaluate the full system. The benefit is better audit scope. Instead of treating every deployed application as an opaque artifact, auditors can separate known components from new or modified logic, identify which functions each component controls, review storage and permission assumptions, and compare the current system against its previous state. This gives reviewers a clearer map of where new or system-specific risk is most likely to appear.

Trusted reuse therefore depends on three connected layers: component identity, composition transparency, and lifecycle visibility. Compose's role is to develop these practical layers for modular smart-contract systems, so reusable logic can move from informal code reuse toward shared on-chain infrastructure while supporting the verification work that trust requires.

7. Use Cases

In general, Compose is most useful when smart-contract systems need to be modular, reusable, and understandable over a long lifecycle. These needs appear in different forms depending on the project's size, maturity, governance model, and tolerance for change.

By applying SCOP through explicit components, Compose turns the same underlying architecture into different practical benefits for different contexts.

Large Modular Systems

Larger protocols often contain multiple domains of logic, such as permissions, accounting, transfers, governance, rewards, or integrations. Compose makes this separation natural by organizing each domain into focused components that are easier to understand and reason about. This makes the overall system easier to review, audit, and maintain.

This structure also makes collaboration easier. Teams can divide work around clear modules, allowing each developer or team to own a focused part of the system with less risk of unintended coupling. Well-understood facets can be implemented quickly to deliver features, while those that require deeper research can be isolated, documented, and developed on their own timeline without destabilizing the rest of the development cycle.

Transparent Upgradeable Systems

Some smart-contract systems need to evolve after deployment as requirements change, the system matures, or new integrations become necessary. Compose supports this lifecycle by allowing functionality to be upgraded in focused parts through explicit component-level changes. Because related functions are grouped into facets, upgrades become easier to inspect, and logic that is meant to move together can be changed together with less manual coordination and lower risk of human error.

This flexibility matters in real-world environments where requirements can change after deployment because of new integrations, governance decisions, security incidents, compliance needs, or larger business requirements. Compose makes this easier by allowing change to happen at the component level, while events and explicit upgrade records preserve part of the transparency and accountability normally associated with immutability.

Upgrade-to-Immutable & Immutable Systems

Compose is also useful for systems that are intended to become immutable. A project can begin with upgradeability during development or governance-controlled operation, then remove its upgrade mechanism once the system is complete. This makes immutability a final deployment posture without giving up flexibility during the earlier stages of development.

Standard-Based Applications

Compose is not limited to large protocols. Many applications only need a standard base, ownership, access control, and a few extensions, but they can still benefit from being assembled quickly from known components. By deploying a lightweight diamond proxy and reusing verified facets, teams can reduce time to market, lower duplicated deployment cost, and still benefit from the natural properties of the diamond model.

Across these use cases, Compose provides a different way to think about how on-chain applications can be implemented, deployed, and operated. By placing smart contracts at the center of the design, Compose gives teams and developers a clearer way to build, review, and operate modular systems that can evolve over time.

8. Building the Compose Ecosystem

Compose grew from practical work around diamond-based smart-contract development, the foundational work led by Nick Mudge, and the contributions of more than 30 individuals who helped shape the early codebase.

That work created the foundation for a broader view of modular on-chain applications: systems built from shared pre-deployed contracts.

The direction behind Compose took time to form because the problem extends beyond any single library, tool, or architectural pattern. It sits at the intersection of code reuse, upgradeability, auditability, deployed infrastructure, developer workflow, and long-term trust.

For Compose to become useful beyond its core library, it must make modular smart-contract systems easier to assemble, inspect, and operate across their lifecycle. Developers need a practical framework for composing systems, while auditors need a clearer view of how known components, custom logic, permissions, storage, and upgrades fit together.

Over time, Compose will expand into a broader ecosystem of open-source libraries, developer tooling, deployed component libraries, and trusted registries.

The goal is not only to provide reusable contracts, but to create the conditions for reusable on-chain infrastructure to be discovered, verified, monitored, and safely reused.

In this sense, Compose is not only a library for building diamonds. It is a step toward an composition ecosystem where modular on-chain applications can be assembled from known components and reviewed as structured systems.

9. Conclusion

On-chain applications are evolving from isolated contracts into long-lived systems made of many interacting parts. As this happens, source-code reuse alone is no longer enough. Reused logic must remain identifiable, inspectable, and trustworthy after deployment.

Compose introduces Smart Contract Oriented Programming as a practical model for that shift. By building on diamond-native architecture, it treats applications as composed systems made from explicit components, where logic, state, identity, and lifecycle management can be reasoned about directly.

Rather than hiding complexity, Compose gives modular on-chain systems a clearer structure. It provides a foundation for applications that can be assembled from known components, reviewed as structured systems, and operated with greater confidence over time.

References

[1] Sun, Kairan, et al. “Demystifying the Composition and Code Reuse in Solidity Smart Contracts.” Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 3 Dec. 2023. https://dl.acm.org/doi/10.1145/3611643.3616270.

[2] Ebrahimi, Amir M., et al. “A Large-Scale Exploratory Study on the Proxy Pattern in Ethereum.” Software Analysis and Intelligence Lab (SAIL), School of Computing, Queen’s University, 2025. https://arxiv.org/pdf/2501.00965.

[3] He, Ningyu, et al. “Characterizing Code Clones in the Ethereum Smart Contract Ecosystem.” Beijing University of Posts and Telecommunications, 2019. https://arxiv.org/pdf/1905.00272.

[4] Mudge, Nick. “ERC-2535: Diamonds, Multi-Facet Proxy.” Ethereum Improvement Proposals, Final, 2020. https://eips.ethereum.org/EIPS/eip-2535.

[5] Mudge, Nick. “ERC-8153: Facet-Based Diamonds.” Ethereum Improvement Proposals, Draft, 2026. https://eips.ethereum.org/EIPS/eip-8153.