LEDGER: Blockchain-Oriented Software Engineering Principles

Luan Martins
COPPE, Universidade Federal do Rio de Janeiro
Rio de Janeiro, Brazil
luanmf@cos.ufrj.br

Diego Castro
COPPE, Universidade Federal do Rio de Janeiro
Rio de Janeiro, Brazil
diegocbcastro@cos.ufrj.br

Cláudia Maria Lima Werner
COPPE, Universidade Federal do Rio de Janeiro
Rio de Janeiro, Brazil
werner@cos.ufrj.br

Cláudio Miceli de Farias
COPPE, Universidade Federal do Rio de Janeiro
Rio de Janeiro, Brazil
cmicelifarias@cos.ufrj.br

Abstract

Blockchain systems diverge sharply from conventional software: deployed code cannot be patched, execution is metered by an economic cost paid by users, consensus correctness requires determinism across mutually untrusting nodes, and source, state, and pending transactions are publicly observable. These constraints resist the direct application of general-purpose principles such as SOLID and Clean Code. The blockchain software engineering literature has responded with pattern catalogs, vulnerability taxonomies, gas-optimization studies, and subdiscipline surveys, but no concise principle catalog of the kind practitioners memorize and teach. This paper proposes **LEDGER**, six principles for blockchain-oriented software engineering: **L**ayered Security, **E**xplicit Trust, **D**eterminism, **G**as Discipline, **E**nduring Code, and **R**eadable by Anyone. The six are derived from structural properties of the blockchain medium that recur across independent systematizations of the field, and validated by mapping the catalog onto six high-impact incidents, one canonical exemplar per principle, whose published root causes each re-express as a violation of the corresponding principle. The contribution is a teachable principle set the blockchain software engineering community can adopt, refine, and extend.

Keywords

blockchain software engineering, smart contracts, design principles, distributed ledger technology, SOLID

1 Introduction

Blockchain has emerged as a distinct software medium in which smart contracts execute on a distributed ledger maintained by consensus among many independent nodes. The medium now backs financial systems, supply-chain registries, identity systems, and governance protocols with millions of users and billions of dollars at stake. Software engineering for this medium has matured into a recognized subdiscipline, *blockchain-oriented software engineering* (BOSE), with a dedicated workshop series and a growing body of academic literature. As the subdiscipline matures, the question of which design principles its engineers should adopt has become correspondingly urgent.

The medium imposes structural constraints that general-purpose software engineering was not designed to address. A *smart contract* is program code that lives on the chain and executes when a transaction invokes it; once deployed, its compiled form cannot be modified by the developer, by the platform, or by any authority short of a chain-wide consensus event [16]. Execution is metered: every operation and every byte of state has a cost, called *gas* on Ethereum and corresponding terms on other platforms, paid in the platform's native currency by the user who issued the transaction [5, 8]. Consensus correctness requires every honest node executing the same code on the same input to arrive at the same result, which rules out floating-point arithmetic, system clocks, network calls, and any reliance on local environment; the runtime is also publicly observable, with source code, contract state, and pending transactions visible to participants, competitors, and adversaries alike. Together these constraints push back against the direct application of standard software engineering principles such as SOLID, design patterns, and Clean Code, built for environments that contradict each of these assumptions.

The literature on blockchain software engineering has produced extensive pattern catalogs [19], vulnerability taxonomies [21], defect catalogs validated by practitioner survey [4], and subdiscipline surveys [9], but no concise principle catalog of the kind practitioners memorize, quote, and teach. Workshop calls for such a synthesis have recurred since 2017 [7, 15], and the closest published attempts sit in practitioner grey literature without systematic methodology or coverage of the infrastructure level [17]. The gap is at the principle level specifically: a small, named set of design rules for blockchain that engineers can adopt, refine, and validate as the discipline matures.

We close the gap by deriving the principles directly from the medium's structural properties rather than by surveying the literature or proposing yet another pattern catalog. Independent systematizations of distributed ledger technology [10] and of blockchain engineering [9] converge on a small set of properties that distinguish the medium from general-purpose software, and Section 3.1 states the six we take as load-bearing. Each property becomes the seed of one principle. The resulting catalog is then validated by mapping it onto a set of well-documented incidents whose published root causes can be re-expressed in the catalog's vocabulary [2, 24].

This paper proposes **LEDGER**, six principles for blockchain-oriented software engineering: **L**ayered Security, **E**xplicit Trust, **D**eterminism, **G**as Discipline, **E**nduring Code, and **R**eadable by Anyone; each is stated as a one-sentence rule, grounded in the structural property it codifies, and paired with a brief note on its nearest counterpart in classical software engineering. The paper

contributes an explicit derivation of the catalog from six structural properties of the medium, with an honest account of what it leaves out; the catalog itself, with a rationale for each principle; and a retrospective evidence section mapping high-impact incidents to principle violations. The paper is divided as follows: Section 2 reviews related work and identifies the gap LEDGER fills; Section 3 derives the catalog and presents each principle; Section 4 applies the catalog to past incidents and reports the mapping; Section 5 discusses scope, limitations, and the validation agenda beyond this paper; Section 6 concludes.

2 Related Work

The literature relevant to LEDGER spans three bodies of work: the framing of blockchain as a distinct software engineering subdiscipline, the contract-level literature on patterns, defects, and quality metrics, and the infrastructure-level literature on protocol trade-offs. The BOSE framing traces to Porru et al. [15], who coined the term *blockchain-oriented software engineering* and launched the IWBOSE workshop series; the agenda has been carried forward by integrated process-model proposals [18] and consolidated by the ACM Computing Surveys review of Fahmideh et al. [9], which characterizes the discipline across foundations, processes, models, and roles. Together these works establish that blockchain engineering warrants its own treatment and identify recurring challenges (immutability, gas economics, security-criticality, the dual on-chain/off-chain architecture). They do not, however, prescribe a small principle set to address those challenges.

At the contract level, two parallel literatures have accumulated. The catalog literature has produced extensive collections of design patterns: Wöhrer and Zdun cataloged security patterns for Ethereum smart contracts [22]; the systematic literature review of Six et al. [19] enumerates more than 120 blockchain-related patterns; and the architectural textbook of Xu, Weber, and Staples [23] organizes patterns by interaction, data management, security, and structural concerns. The empirical literature on quality, in parallel, has produced defect taxonomies validated by practitioner survey [4], gas-inefficiency studies [5, 8], and vulnerability classifications enumerating ninety-four flaw types [21]. Both literatures document the engineering surface with empirical rigor but operate one level below the principle catalog LEDGER targets: patterns are concrete solutions to recurring problems, and defect or quality taxonomies describe what goes wrong rather than the design rules that would prevent it. Beyond academia, the practitioner community maintains widely used security guidance, including the OWASP Smart Contract Top 10 [14], Trail of Bits' *Building Secure Contracts* [20], the ConsenSys smart-contract best-practices compendium [6], and OpenZeppelin's audited reference implementations [13]. These resources enumerate vulnerabilities, review checklists, and hardened components; they operate at the same sub-principle level as the academic catalogs and complement, rather than replace, the small principle set LEDGER distills above them.

The infrastructure-level literature has formalized the engineering trade-offs of the protocol layer itself. The distributed ledger technology (DLT) trade-off framework of Kannengießer et al. [10] organizes DLT characteristics into six groups with explicit trade-off relationships, identifying the design tensions any platform must resolve; this systematization is one of the references against which our derivation of LEDGER is anchored. Permissionless consensus has been formalized in Lewis-Pye and Roughgarden's hierarchy of consensus models and impossibility results [11]. These works establish design constraints at the infrastructure level but, like the contract-level literature, characterize the engineering terrain rather than distilling it into named principles.

None of these bodies has produced a concise principle catalog at the abstraction level practitioners memorize, quote, and teach. The only attempt to date sits in practitioner grey literature without systematic methodology or coverage of the infrastructure level [17]. To our knowledge, LEDGER is among the first peer-reviewed proposals at this abstraction level: derived from the structural properties the literature above already identifies, and validated against the failure record the literature above already documents.

3 The LEDGER Principles

Section 3.1 derives the six principles from structural properties of the medium that recur across independent systematizations of the field, and names which candidate principles the catalog deliberately leaves out. Subsections 3.2 through 3.7 then present each principle in order of logical dependency (starting with Enduring Code, the load-bearing principle from which the others draw their urgency, followed by Determinism, Gas Discipline, Explicit Trust, Layered Security, and Readable by Anyone), with a one-sentence statement, a rationale grounded in the blockchain literature, and a short paragraph titled *Closest SOLID neighbor* for pedagogical reference. Figure 1 summarizes the derivation.

3.1 Deriving the catalog

The six principles are derived from six structural properties of the blockchain medium that recur across independent systematizations of the field. The DLT trade-off survey of Kannengießer et al. [10] organizes the medium into a small number of characteristic groups with explicit trade-off relationships. The ACM Computing Surveys engineering review of Fahmideh et al. [9] extracts a similar set of recurring engineering challenges. The smart-contract attack survey of Atzei et al. [2] characterizes the failure modes that follow when those properties are mishandled. Across these three independent lines, six properties consistently distinguish the medium from general-purpose software:

(P1) Permanence. Deployed code cannot be modified after deployment short of a chain-wide hard fork [16]. **(P2) Determinism.** Consensus correctness requires every honest node executing the same code on the same input to arrive at the same result. **(P3) Economic metering.** Each operation has a cost paid by the transaction sender in the platform's native currency. **(P4) Adversarial trust.** The caller, the called, and the off-chain world are potentially hostile; no implicit trust crosses a contract boundary. **(P5) Layered architecture.** Applications depend on libraries, contracts on a consensus layer, rollups on a settlement layer, bridges on multiple chains. **(P6) Public observability.** Source code, contract state, and pending transactions are visible to every participant.

We treat each property as the seed of one principle: Enduring Code codifies P1, Determinism codifies P2, Gas Discipline codifies P3, Explicit Trust codifies P4, Layered Security codifies P5, Readable by Anyone codifies P6 (Figure 1). The mapping is one-to-one. The cardinality is not a free parameter chosen to spell an acronym; it is the count of irreducible structural properties on which the cited systematizations converge. The principles are orthogonal by construction, one per property, but they routinely co-occur when violated, as Section 4 shows.

What the catalog leaves out. Three exclusions deserve to be named. *Privacy* does not generate a separate principle: in a publicly observable medium, privacy is pursued against the medium (through zero-knowledge proofs or off-chain computation), and we fold its discipline into Readable by Anyone, which acknowledges the tension. *Scalability and throughput* are quality attributes engineers optimize within the principles, not peers to them, sitting one level below LEDGER as performance sits below classical principle catalogs. *Consensus-aware design*, concerning how application logic interacts with finality, reorganizations, and validator incentives, is the most plausible candidate for a seventh principle.

3.2 Enduring Code

Statement. *Code deployed on-chain is permanent; design every line as if it must outlive any ability to fix it.*

Enduring Code is the load-bearing principle of LEDGER. Once a smart contract is deployed, its bytecode cannot be modified by the developer, by the platform, or by any authority short of a chain-wide hard fork. This permanence is not a limitation to work around but the defining property of the medium, and it is the source from which the other principles draw their urgency: determinism matters because incorrect execution cannot be retroactively corrected; gas discipline matters because inefficient code keeps costing users for as long as the chain runs; explicit trust matters because trust boundaries set at deployment cannot be renegotiated.

The empirical reality is more nuanced. *Proxy patterns*, in which a contract forwards calls to an implementation contract whose address can be updated, and upgradeability standards such as EIP-2535 and EIP-1822 provide mechanisms to evolve deployed systems. These patterns do not contradict the principle; they implement it. A proxy is an architectural acknowledgment that the underlying contract is immutable: evolvability must be designed in before deployment, never bolted on after. The empirical literature shows how rarely this is done well: of 44 million analyzed contracts, only 3% exhibit upgradeable characteristics, and 50% of upgradeable contracts are controlled by a single externally-owned account, a centralization risk that exists precisely because upgradeability was treated as a feature rather than as a design discipline [16].

Closest SOLID neighbor. The Open/Closed Principle states that software entities should be open for extension and closed for modification. Enduring Code reads as the structural variant: deployed contracts are literally closed for modification, not as a design choice but as a property of the medium. The "open for extension" half remains valid but acquires a new cost: extension mechanisms (proxies, modular architectures) must be engineered before deployment, not discovered during maintenance.

3.3 Determinism

Statement. *All nodes must produce identical results from identical inputs; non-determinism is a correctness bug, not a performance issue.*

Blockchain consensus depends on every honest node reaching the same conclusion about the new state after executing a transaction. If two nodes executing the same contract on the same input produce different results, consensus fails. This requirement removes from the smart-contract programmer's toolkit a wide range of operations considered routine elsewhere: floating-point arithmetic, system clocks, random number generators, network calls, file system access, and any reliance on locale, environment variables, or hardware-specific behavior. The set of available primitives is small, and the discipline of staying inside it is absolute.

Determinism is enforced structurally rather than by testing. The Ethereum Virtual Machine and its successors omit non-deterministic instructions; Solidity provides no floating-point type; block timestamps are available but must be treated as adversarial inputs, since block producers can manipulate them within a tolerance; randomness, when needed, is supplied by an oracle or derived from a verifiable randomness function, never generated locally; and violations of determinism do not produce silent corruption but chain forks, which the consensus engine surfaces immediately.

Closest SOLID neighbor. SOLID has no direct counterpart: the assumption it would address, that programs run in environments offering rich non-deterministic services, is too foundational to traditional software engineering to require articulation. In blockchain, a non-deterministic contract is not slow or unreliable but incorrect by construction.

3.4 Gas Discipline

Statement. *Every operation executed and every byte stored on-chain is a recurring cost paid by users; do as little on-chain as the problem allows.*

Computation and storage on a blockchain are metered economically. Each operation has a gas cost, paid by the transaction sender in the platform's native currency. The term *gas* originated in Ethereum; the discipline applies equally to Solana's compute units, Polkadot's weight, and the metering mechanisms of every other smart-contract platform. What unifies these systems is the economic substrate: on-chain resources are not just constrained but priced, and the price is paid by users for the lifetime of the deployed code.

Gas Discipline is best understood as a principle of restraint rather than optimization. The deepest savings come not from cheaper ways to perform a computation but from declining to perform it on-chain at all: events are cheaper than storage; off-chain computation with on-chain verification is cheaper than on-chain computation; storage layouts that pack related variables into single 256-bit slots are cheaper than naïve layouts. The empirical literature documents this directly: storage operations dominate gas costs by orders of magnitude, and minimizing them is the single highest-leverage optimization a contract author can apply [5, 8].

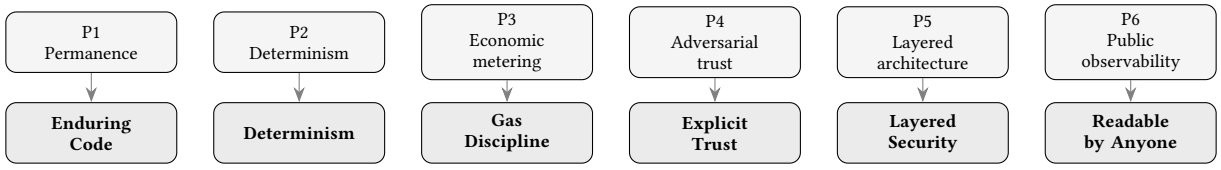

Figure 1: Deriving LEDGER. Each of the six structural properties of the blockchain medium (P1–P6, top row) seeds exactly one principle (bottom row). The mapping is one-to-one; Enduring Code is load-bearing, and the other five draw their urgency from the permanence it codifies.

Closest SOLID neighbor. The Single Responsibility Principle encourages small, focused functions. Gas Discipline reads as a parallel constraint on the same axis: small functions are still good practice, but per-call overhead is now an additional cost the engineer must weigh. Gas-optimal code is sometimes less SRP-compliant than clean-code aesthetics alone would recommend [12]. SOLID does not become invalid in the presence of gas costs; it becomes one term in a cost equation it never had to solve.

3.5 Explicit Trust

Statement. *Every component must declare what it trusts and what it verifies; no implicit trust crosses a contract boundary.*

Smart contracts execute in an environment where the caller, the called, the surrounding contracts, and the off-chain world are all potentially adversarial. Trust cannot be inferred from context. A contract that accepts data from an *oracle* (an off-chain data feed) must specify what makes the oracle trustworthy and what happens if the oracle lies. A contract that delegates execution through `delegatecall` (an EVM primitive that runs the callee's code in the caller's storage context) must reason about the storage layout of the target contract, because they share state. A contract that integrates with another contract through a standard interface (such as ERC-20 for fungible tokens or ERC-721 for non-fungibles) must verify that the counterparty honors the interface's invariants, not merely its signature.

Making trust explicit has three practical implications: external inputs must be validated cryptographically or economically, by checking signatures, requiring stake, or composing redundant oracles; access control must be declared per-function and audited per-deployment, since least privilege is a survival requirement rather than a recommendation; and composability is a trust-composition problem, since invoking another contract imports its security assumptions into one's own, a relationship whose formal treatment has only recently begun [3].

Closest SOLID neighbor. The Dependency Inversion Principle decouples high-level modules from low-level modules through abstractions. It assumes both modules run in a trusted process: the abstraction defines the interface, the implementation honors it. Explicit Trust reads as a generalization of the same idea into an adversarial setting: dependency direction is one axis to invert, and trust posture is a second axis the engineer must declare. An interface alone is not a contract; a verified, signed, or staked interface is.

3.6 Layered Security

Statement. *Layered systems must formally derive their security guarantees from the layer below; security does not compose by accident.*

Modern blockchain architectures are layered. Smart contracts depend on the consensus and execution guarantees of *Layer 1* (the base chain). *Layer 2* systems derive their security from Layer 1 through different mechanisms: *optimistic rollups* through fraud proofs and the assumption of at least one honest challenger, *ZK rollups* through validity proofs verified on Layer 1, and *validiums* weaken the inheritance by moving data availability off-chain. Bridges that connect two chains inherit the security of the weaker of the two. Applications depend on libraries (OpenZeppelin, in the Ethereum world) whose security must be inherited explicitly rather than rebuilt.

The principle states that each layer must formally inherit its guarantees from the layer beneath, with the inheritance mechanism made explicit. The same pattern applies downward to the application layer: a contract that imports OpenZeppelin's ERC-20 implementation inherits a specific, audited set of security properties for the imported code, but not for the contract's own logic. The recurring engineering failure is to treat the inheritance as automatic, on the assumption that secure components compose into secure systems.

Closest SOLID neighbor. The Liskov Substitution Principle and the Interface Segregation Principle assume that well-designed interfaces compose into well-behaved systems. The assumption holds in cooperative environments. In an adversarial setting, two secure components can compose into an insecure system through emergent interactions the individual components cannot anticipate. Layered Security makes the implicit assumption explicit: composition preserves correctness only when the inheritance is stated and proved, not when it is assumed.

3.7 Readable by Anyone

Statement. *Public verifiability is the system's fundamental security property; design every component so its behavior can be audited by anyone, at any time.*

Every transaction, every state change, and every line of deployed bytecode on a public blockchain is observable by every participant, including adversaries. This visibility is not a leak to be plugged but the source of the system's security: a blockchain is trustworthy because anyone can verify it. The principle takes the property seriously and asks the engineer to design *for* auditability, not merely to tolerate it.

In practice, Readable by Anyone manifests as three concrete practices: source code should be published and verified against deployed bytecode through services such as Etherscan verification or Sourcify, since unverified contracts forfeit the principal advantage of the medium; contract behavior should be designed to be analyzable by static-analysis tools, auditors, and motivated users, since opacity is a vulnerability rather than a defense; and the publicly observable execution environment must be accounted for in the design, since transaction ordering can be manipulated by block producers in pursuit of Maximal Extractable Value (MEV) and pending transactions can be observed and front-run.

Closest SOLID neighbor. SOLID has no direct counterpart because traditional software engineering treats source transparency as orthogonal to correctness: open-source and proprietary systems can each be well-engineered. In blockchain software engineering, transparency is itself a correctness property in the security sense. A contract whose behavior cannot be publicly verified cannot inherit the trust guarantees that make the medium meaningful in the first place.

4 Evidence from Past Incidents

A principle catalog earns its place by explaining failures the field already recognizes. This section applies LEDGER retrospectively to six high-impact blockchain incidents, selected as one canonical exemplar per principle, and asks a single question: *do the published root causes map onto LEDGER violations?* Subsection 4.1 states the case-selection criteria and reports the mapping in Table 1; Subsection 4.2 draws two observations on the catalog's vertical coverage and the principles' typical co-occurrence in practice.

4.1 Method and mapping

We selected six incidents, one per LEDGER principle, under three criteria: each has a published root cause in a peer-reviewed survey [2, 7, 24] or an authoritative protocol post-mortem [1]; each caused losses or systemic effects at a scale that mobilized professional and community attention; and together they span the failure surfaces the catalog claims to cover, from consensus-layer determinism through contract logic, gas economics, library composition, bridge security, and visibility-based deception. Each incident's published root cause is compressed to a single sentence in Table 1, alongside the violated principle. The mapping is a sanity check, not a quantitative validation: each canonical exemplar maps onto its principle without remainder.

4.2 Observations

Two observations follow. First, the one-to-one correspondence is a pedagogical choice: real incidents typically violate several principles at once, with Explicit Trust and Layered Security co-occurring in most contract-level failures documented in the survey literature [2, 24], and the table compresses each incident to the principle that most directly explains its loss. Second, the principles span the abstraction levels at which blockchain systems fail (consensus, contract semantics, contract economics, composition, bytecode visibility), a vertical no existing blockchain-engineering taxonomy covers in one principle set.

5 Discussion

The LEDGER catalog occupies a single abstraction level, the principle level, above patterns and practices and beneath the engineering-discipline surveys. It is neither a complete engineering methodology for blockchain systems nor a closed set of every principle the subdiscipline will eventually require. This section states who should use the catalog and how, with a worked example; lays out the empirical validation agenda and its challenges; and acknowledges the limitations of the proposal as it stands.

Who should use LEDGER, and when. The catalog serves instructors teaching blockchain-oriented design, researchers situating new studies, contract developers wanting a checklist shorter than a vulnerability taxonomy, auditors structuring review, and platform engineers reasoning about cross-layer risk. As a worked example, an engineer reviewing a lending protocol *before* deployment can walk the catalog: is every upgrade path designed in rather than bolted on (Enduring Code); does any branch depend on floating-point or wall-clock time (Determinism); can an unbounded loop over user-controlled state reach the block gas limit (Gas Discipline); is every oracle and external call explicit about what it trusts and verifies (Explicit Trust); is each inherited library guarantee stated rather than assumed (Layered Security); and is the deployed source verified and its ordering assumptions treated as front-running-aware (Readable by Anyone)? Each question ties a design decision to the principle whose violation Section 4 shows to be costly.

Validation agenda. The evidence of Section 4 is a sanity check rather than a quantitative validation, and the catalog requires scrutiny along three complementary axes. First, an *expert review* in the tradition of practitioner-anchored software engineering taxonomies [4]: a survey instrument administered to blockchain engineers will assess each principle's recognizability, perceived usefulness, and completeness, and the catalog will be refined against the results. Second, a *repository-mining study* on public Ethereum and non-EVM contracts will derive measurable signals for each principle (for example, the prevalence of non-upgradeable contracts handling critical state for Enduring Code, cataloged gas anti-patterns for Gas Discipline, and source-verification rates for Readable by Anyone) and quantify the catalog's discriminating power across deployed code at scale. Third, the incident analysis will be extended to a larger, classified corpus of post-mortems with inter-rater reliability and a quantitative breakdown of which principles co-occur with which failure classes. Each study carries its own difficulty: recruiting a representative sample of blockchain engineers, establishing chain-agnostic ground truth for the mining signals across heterogeneous platforms, and reaching acceptable inter-rater agreement on incident classification.

Limitations. Four limitations deserve explicit acknowledgment. First, the evidence is illustrative rather than a quantitative validation: the incident set is one canonical exemplar per principle, not an exhaustive sample, and principle co-occurrence is noted but not yet quantified. Second, the catalog of six is a starting point rather than a closed set, and Consensus-Aware Design (Section 3.1) is the most plausible candidate for a seventh principle. Third, the contract-level examples skew toward Ethereum and Solidity, reflecting the maturity of that ecosystem in the literature,

Table 1: Six incidents, one canonical exemplar per LEDGER principle, whose published root causes correspond to a violation of that principle. Rows follow the LEDGER ordering; impact values are approximate at the time of the incident.

Incident	Year	Impact	Published root cause (compressed)	LEDGER violation
Ronin Bridge [24]	2022	~$625M	Five of nine validator keys were compromised, satisfying the bridge's signature threshold for fraudulent Layer 1 withdrawals; the application layer's security was inherited from a 5-of-9 multisig without formal derivation.	**L**ayered Security.
The DAO [2]	2016	~$60M	*Reentrancy*: a callback to an untrusted external contract re-entered the withdrawal function before the caller's balance was decremented.	**E**xplicit Trust.
Bitcoin 2013 fork [1]	2013	6h chain split	A large block accepted by Bitcoin 0.8 (LevelDB) was rejected by pre-0.8 nodes (BerkeleyDB) because the default lock count on 0.7 was insufficient; two implementations diverged on block validity.	**D**eterminism.
GovernMental Ponzi [2]	2016	~1,100 ETH locked	The payout function looped over the participant list to clear state; once the list grew large enough, the loop's gas cost exceeded the block gas limit and the jackpot became permanently inaccessible.	**G**as Discipline.
Parity Multisig Freeze [7]	2017	~$280M	An unprivileged user invoked `initWallet` on a shared library, took ownership, and self-destructed it; all dependent wallets became inaccessible, and the loss is permanent because deployed code cannot be patched.	**E**nduring Code.
Squid Game Token [24]	2021	~$3.3M	The deployed contract contained an owner-only `sell` restriction absent from the project's marketing material and from any public source on Etherscan; the owner drained liquidity.	**R**eadable by Anyone.

so the chain-agnosticity claim must be tested explicitly against non-EVM platforms such as Solana and Cardano. Finally, the principles have not yet been evaluated with practitioners; the expert review above is the immediate next step, and the repository- and incident-mining studies address the first and third limitations in turn.

6 Conclusion

This paper proposed LEDGER, a catalog of six principles for blockchain-oriented software engineering (Layered Security, Explicit Trust, Determinism, Gas Discipline, Enduring Code, Readable by Anyone), derived from six structural properties of the medium that recur across independent systematizations of the field. The retrospective evidence of Section 4 shows that each principle corresponds to a canonical high-impact incident whose published root cause is a direct violation of that principle. Blockchain software engineering thus gains a concise, teachable principle catalog together with an initial body of supporting evidence. The validation agenda of Section 5, expert review first, will refine the catalog's boundaries, test the cardinality argument for a seventh principle (most plausibly Consensus-Aware Design), and check the chain-agnosticity claim against non-EVM systematizations of Solana, Cardano, and modular ecosystems.

Artifact Availability

A replication package, containing the structured incident-to-principle mapping of Section 4 with its per-incident source references and the manuscript sources, is archived on Zenodo under a Creative Commons license: https://doi.org/10.5281/zenodo.21499174 (version 1.0). It draws only on post-mortem data from the cited surveys and public protocol post-mortems; artifacts from the validation studies of Section 5 will follow the same conventions.

Acknowledgements

The authors used Claude (Anthropic) as a research assistant for literature review and manuscript preparation.

References

[1] Gavin Andresen. 2013. BIP 50: March 2013 Chain Fork Post-Mortem. Bitcoin Improvement Proposal. https://github.com/bitcoin/bips/blob/master/bip-0050.mediawiki Accessed: May 22, 2026.

[2] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *Proceedings of the 6th International Conference on Principles of Security and Trust (POST) (Lecture Notes in Computer Science, Vol. 10204)*. Springer, 164–186. doi:10.1007/978-3-662-54455-6_8

[3] Massimo Bartoletti, Riccardo Marchesin, and Roberto Zunino. 2024. DeFi Composability as MEV Non-interference. In *Proceedings of the 28th International Conference on Financial Cryptography and Data Security (FC) (Lecture Notes in Computer Science, Vol. 14745)*. Springer. doi:10.1007/978-3-031-78679-2_20

[4] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2022. Defining Smart Contract Defects on Ethereum. *IEEE Transactions on Software Engineering* 48, 1 (2022), 327–345. doi:10.1109/TSE.2020.2989002

[5] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-Optimized Smart Contracts Devour Your Money. In *Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 442–446. doi:10.1109/SANER.2017.7884650

[6] ConsenSys Diligence. 2023. Ethereum Smart Contract Best Practices. Online compendium. https://consensys.github.io/smart-contract-best-practices/ Accessed: July 21, 2026.

[7] Giuseppe Destefanis, Michele Marchesi, Marco Ortu, Roberto Tonelli, Andrea Bracciali, and Robert Hierons. 2018. Smart Contracts Vulnerabilities: A Call for Blockchain Software Engineering?. In *Proceedings of the 1st IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 19–25. doi:10.1109/IWBOSE.2018.8327567

[8] Andrea Di Sorbo, Sonia Laudanna, Anna Vacca, Corrado Aaron Visaggio, and Gerardo Canfora. 2022. Profiling Gas Consumption in Solidity Smart Contracts. *Journal of Systems and Software* 186 (2022), 111193. doi:10.1016/j.jss.2021.111193

[9] Mahdi Fahmideh, John Grundy, Aakash Ahmad, Jun Shen, Jun Yan, Davoud Mougouei, Peng Wang, Aditya Ghose, Anuradha Gunawardana, Uwe Aickelin, and Babak Abedin. 2023. Engineering Blockchain-Based Software Systems: Foundations, Survey, and Future Directions. *Comput. Surveys* 55, 6, Article 110 (2023), 44 pages. doi:10.1145/3530813

[10] Niclas Kannengießer, Sebastian Lins, Tobias Dehling, and Ali Sunyaev. 2020. Trade-Offs between Distributed Ledger Technology Characteristics. *Comput. Surveys* 53, 2, Article 42 (2020), 37 pages. doi:10.1145/3379463

[11] Andrew Lewis-Pye and Tim Roughgarden. 2023. Permissionless Consensus. arXiv:2304.14701 [cs.DC] https://arxiv.org/abs/2304.14701 Conference version in Financial Cryptography and Data Security 2023 titled "Byzantine Generals in the Permissionless Setting".

[12] Lodovica Marchesi, Michele Marchesi, Giuseppe Destefanis, Giulio Barabino, and Danilo Tigano. 2020. Design Patterns for Gas Optimization in Ethereum. In *Proceedings of the 2nd IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 9–15. doi:10.1109/IWBOSE50093.2020.9050163

[13] OpenZeppelin. 2024. OpenZeppelin Contracts: A Library for Secure Smart Contract Development. Software library, version 5. https://github.com/OpenZeppelin/openzeppelin-contracts Accessed: July 21, 2026.

[14] OWASP Foundation. 2025. OWASP Smart Contract Top 10. OWASP Smart Contract Security Project. https://owasp.org/www-project-smart-contract-top-10/ CC BY-NC-SA 4.0. Accessed: July 21, 2026.

[15] Simone Porru, Andrea Pinna, Michele Marchesi, and Roberto Tonelli. 2017. Blockchain-Oriented Software Engineering: Challenges and New Directions. In *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 169–171. doi:10.1109/ICSE-C.2017.142

[16] Ilham Qasse, Mohammad Hamdaqa, and Björn Þór Jónsson. 2024. Immutable in Principle, Upgradeable by Design: Exploratory Study of Smart Contract Upgradeability. arXiv:2407.01493 [cs.SE] https://arxiv.org/abs/2407.01493

[17] Daniel Shvetsov. 2023. SOLID Principles in Smart Contract Development. HackerNoon. https://hackernoon.com/solid-principles-in-smart-contract-development Accessed: July 22, 2026.

[18] Christian Sillaber, Bernhard Waltl, Horst Treiblmaier, Ulrich Gallersdörfer, and Michael Felderer. 2021. Laying the Foundation for Smart Contract Development: An Integrated Engineering Process Model. *Information Systems and e-Business Management* 19, 3 (2021), 863–882. doi:10.1007/s10257-020-00465-5

[19] Nicolas Six, Nicolas Herbaut, and Camille Salinesi. 2022. Blockchain Software Patterns for the Design of Decentralized Applications: A Systematic Literature Review. *Blockchain: Research and Applications* 3, 1 (2022), 100061. doi:10.1016/j.bcra.2022.100061

[20] Trail of Bits. 2024. Building Secure Contracts: Guidelines and Best Practices for Secure Smart Contract Development. Online handbook. https://secure-contracts.com/ Accessed: July 21, 2026.

[21] Fernando Richter Vidal, Naghmeh Ivaki, and Nuno Laranjeiro. 2024. OpenSCV: An Open Hierarchical Taxonomy for Smart Contract Vulnerabilities. *Empirical Software Engineering* 29, 4 (2024), 101. doi:10.1007/s10664-024-10446-8

[22] Maximilian Wöhrer and Uwe Zdun. 2018. Smart Contracts: Security Patterns in the Ethereum Ecosystem and Solidity. In *Proceedings of the 1st IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 2–8. doi:10.1109/IWBOSE.2018.8327565

[23] Xiwei Xu, Ingo Weber, and Mark Staples. 2019. *Architecture for Blockchain Applications*. Springer. doi:10.1007/978-3-030-03035-3

[24] Liyi Zhou, Xihan Xiong, Jens Ernstberger, Stefanos Chaliasos, Zhipeng Wang, Ye Wang, Kaihua Qin, Roger Wattenhofer, Dawn Song, and Arthur Gervais. 2023. SoK: Decentralized Finance (DeFi) Attacks. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2444–2461. doi:10.1109/SP46215.2023.10179435