ETSI GS PDL 033 V1.1.1 (2025-06)

Permissioned Distributed Ledger (PDL); Smart Contracts; System Architecture and Functional Specification

Disclaimer

The present document has been produced and approved by the Permissioned Distributed Ledger (PDL) ETSI Industry Specification Group (ISG) and represents the views of those members who participated in this ISG.
It does not necessarily represent the views of the entire ETSI membership.

Reference
DGS/PDL-0033_Smart_contract

Keywords
blockchain, PDL, policies, SLA, smart contract

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - APE 7112B
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° w061004871

Contents

Intellectual Property Rights

Essential patents

IPRs essential or potentially essential to normative deliverables may have been declared to ETSI. The declarations pertaining to these essential IPRs, if any, are publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI IPR online database.

Pursuant to the ETSI Directives including the ETSI IPR Policy, no investigation regarding the essentiality of IPRs, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Trademarks

The present document may include trademarks and/or tradenames which are asserted and/or registered by their owners. ETSI claims no ownership of these except for any which are indicated as being the property of ETSI, and conveys no right to use or reproduce any trademark and/or tradename. Mention of those trademarks in the present document does not constitute an endorsement by ETSI of products, services or organizations associated with those trademarks.

DECT™, **PLUGTESTS™**, **UMTS™** and the ETSI logo are trademarks of ETSI registered for the benefit of its Members. **3GPP™**, **LTE™** and **5G™** logo are trademarks of ETSI registered for the benefit of its Members and of the 3GPP Organizational Partners. **oneM2M™** logo is a trademark of ETSI registered for the benefit of its Members and of the oneM2M Partners. **GSM®** and the GSM logo are trademarks registered and owned by the GSM Association.

Foreword

This Group Specification (GS) has been produced by ETSI Industry Specification Group (ISG) Permissioned Distributed Ledger (PDL).

Modal verbs terminology

In the present document "**shall**", "**shall not**", "**should**", "**should not**", "**may**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the ETSI Drafting Rules (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

Executive summary

The present document provides a comprehensive exploration of smart contract technology within Permissioned Distributed Ledger (PDL) environments. It outlines the foundational principles, architectural considerations, and functional specifications necessary for implementing robust and efficient smart contracts. The present document emphasizes the importance of smart contracts in automating agreements and transactions, highlighting their key characteristics such as automation, transparency, immutability, efficiency, and programmability.

Key sections include an introduction to Object-Oriented Programming (OOP) concepts in smart contracts, which enhance modularity and reusability while maintaining security. The present document also addresses critical properties of smart contracts such as immutability, transparency, determinism, atomicity, autonomy, decentralization, state management, interoperability, and reusability. Each property is explored in detail with definitions, key aspects, implementation considerations, and best practices.

The present document further delves into the lifecycle phases of smart contracts, covering planning, development and testing, deployment and execution, maintenance and upgrade, and retirement or deprecation. It provides guidelines for designing smart contracts with considerations for security, scalability, interoperability, auditability, privacy, governance, and error handling. Emerging standards such as advanced ERC token standards, Non-Fungible Token (NFT) standards, and innovations in smart contract wallets are discussed to address evolving needs in blockchain ecosystems. Additionally, regulatory and environmental considerations are examined to ensure compliance with recent legislation on smart contract enforceability and to address energy consumption concerns through eco-friendly consensus mechanisms like proof-of-stake.

Overall, the present document serves as a comprehensive guide for developers and stakeholders involved in the design, implementation, and management of smart contracts in permissioned distributed ledger systems. It emphasizes the need for careful planning, rigorous testing, ongoing monitoring, and adherence to best practices to harness the full potential of smart contracts while mitigating associated risks.

Introduction

The present document by the ETSI Industry Specification Group (ISG) provides an in-depth exploration of smart contract technology within Permissioned Distributed Ledger (PDL) environments. It serves as a comprehensive guide for developers and stakeholders involved in the design, implementation, and management of smart contracts. The present document emphasizes the transformative potential of smart contracts in automating agreements and transactions, highlighting their key characteristics such as automation, transparency, immutability, efficiency, and programmability.

Smart contracts are defined as self-executing computer programs stored on a distributed ledger system, where the execution outcomes are recorded on the ledger. Unlike traditional contracts that rely on human interpretation and enforcement, smart contracts automatically execute predefined actions when specific conditions are met, eliminating the need for intermediaries. This automation reduces manual intervention, minimizes human error, and streamlines processes.

The present document outlines the architectural considerations and functional specifications necessary for implementing robust and efficient smart contracts. It covers key properties such as immutability, transparency, determinism, atomicity, autonomy, decentralization, state management, interoperability, and reusability. Each property is explored with definitions, key aspects, implementation considerations, and best practices.

Furthermore, the present document addresses emerging standards such as advanced ERC token standards and Non-Fungible Token (NFT) standards to meet evolving needs in blockchain ecosystems. It also examines regulatory and environmental considerations to ensure compliance with recent legislation on smart contract enforceability and address energy consumption concerns through eco-friendly consensus mechanisms like proof-of-stake. Overall, the present document provides valuable insights into the effective design, deployment, and management of smart contracts in permissioned distributed ledger systems. It emphasizes the importance of careful planning, rigorous testing, ongoing monitoring, and adherence to best practices to harness the full potential of smart contracts while mitigating associated risks.

1 Scope

The present document outlines the design, implementation, and management of smart contracts within Permissioned Distributed Ledger (PDL) environments. It provides a comprehensive framework for understanding the architectural principles, functional specifications, and emerging standards necessary for developing robust and efficient smart contract systems. The present document is intended to guide developers, stakeholders, and policymakers involved in smart contract technology, ensuring that these digital agreements operate effectively within PDL ecosystems.

In Scope

- **Smart Contract Architecture:** Detailed exploration of the architectural considerations for smart contracts, including Object-Oriented Programming (OOP) paradigms, modularity, and interoperability.
- **Functional Specifications:** Guidelines on the functional requirements for smart contracts, covering aspects such as automation, transparency, immutability, efficiency, and programmability.
- **Lifecycle Phases:** Examination of the lifecycle phases of smart contracts from planning and development to deployment, maintenance, and retirement.
- **Emerging Standards:** Discussion of advanced ERC token standards, Non-Fungible Token (NFT) standards, and innovations in smart contract wallets and account abstraction.
- **Regulatory Considerations:** Analysis of recent legislation on smart contract enforceability and regulatory guidance from financial authorities.
- **Environmental Impact:** Considerations related to the energy consumption of blockchain platforms and the transition to eco-friendly consensus mechanisms like proof-of-stake.

Out of Scope in this release

- **Specific Implementation Details:** The present document does not provide detailed code implementations or specific programming instructions for smart contracts.
- **Vendor-Specific Solutions:** It does not endorse or focus on solutions provided by specific vendors or proprietary technologies.
- **Comprehensive Legal Analysis:** While it addresses regulatory considerations, it does not provide exhaustive legal analysis or advice on compliance with all jurisdictional laws.
- **Exhaustive Security Protocols:** The present document covers security best practices but does not delve into every possible security protocol or threat mitigation strategy.

This scope ensures that the present document serves as a foundational guide for understanding and implementing smart contracts within PDL environments while recognizing the need for further exploration in specific technical or legal areas.

2 References

2.1 Normative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

Referenced documents which are not found to be publicly available in the expected location might be found in the ETSI docbox.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long-term validity.

The following referenced documents are necessary for the application of the present document.

[1] Agoric (2022): "A Deep Dive into Agoric: JavaScript Smart Contracts" clause 6, Agoric Blog.

[2] Al-Breiki H., Rehman M. H. U., Salah K., & Svetinovic D. (2020): "Trustworthy blockchain oracles: Review, comparison, and open research challenges", IEEE™ Access, 8, 85675-85685.

[3] Wei-Shan Lee, John A, Hsiu-Chun Hsu, Pao-Ann Hsiung (2022): "SPChain: A Smart and Private Blockchain-Enabled Framework for Combining GDPR-Compliant Digital Assets Management With AI Models", IEEE™ Access (vol 10), IEEE™ Journals & Magazine.

[4] Algorand Developer Portal (2022): "Algorand Smart Contracts Using PyTeal".

[5] Algorand Foundation (2020): "Algorand's Smart Contracts Architecture".

[6] Algorand Foundation (2021): "The Algorand Virtual Machine (AVM) and TEAL".

[7] Almakhour M., Sliman L., Samhat A. E. & Mellouk A. (2020): "Verification of smart contracts: A survey", Pervasive and Mobile Computing, 67, 101227.

[8] Void.

[9] Fouzia Alzhrani, Kawther Saeedi, and Liping Zhao: "Architectural Patterns for Blockchain Systems and Application Design", Appl. Sci. 2023, 13(20), 11533.

[10] Antonopoulos A. M. & Wood G. (2020): "Mastering Ethereum: Building Smart Contracts and DApps" (2nd ed.), O'Reilly Media.

[11] Seif T. Nassar, Abeer Hamdy, Khaled Nagaty (2024): "Hybrid zk-STARK and zk-SNARK Framework for Privacy-Preserving Smart Contract Data Feeds", 2024 International Conference on Computer and Applications (ICCA).

[12] Amritraj Singh, Reza M. Parizi, Qi Zhang, Kim-Kwang Raymond Choo, Ali Dehghantanha. (2019): " Blockchain smart contracts formalization: Approaches and challenges to address vulnerabilities", Computers and security Issue 88.

[13] Batubara F. R., Ubacht J. & Janssen M. (2020): "Unraveling transparency and accountability in blockchain". In Proceedings of the 21st Annual International Conference on Digital Government Research (pp. 204-213).

[14] Belchior R., Vasconcelos A., Guerreiro S. & Correia M. (2021): "A survey on blockchain interoperability: Past, present, and future trends", ACM Computing Surveys (CSUR), 54(8), pp. 1-41.

[15] Bernabe J. B., Canovas J. L., Hernandez-Ramos J. L., Moreno R. T. & Skarmeta A. (2021): "Privacy-preserving solutions for Blockchain: Review and challenges", IEEE™ Access, 9, 41917-41959.

[16] Bernardo B., Cauderlier R., Hu Z., Pesin B. & Tesson J. (2020): "Mi-Cho-Coq, a framework for certifying Tezos smart contracts", In International Symposium on Formal Methods (pp. 368-379). Springer, Cham.

[17] BogoToBogo: "Design Patterns - Observer Pattern", BogoToBogo, 2020.

[18] Bragagnolo S., Rocha H., Denker M. & Ducasse S. (2020): "SmartAnvil: Open-source tool suite for smart contract analysis". In Blockchain Technology: Platforms, Tools and Use Cases (pp. 271-287). Elsevier.

[19] Brunner C., Knirsch F. & Engel D. (2020): "SPROOF: A platform for issuing and verifying documents in a public blockchain". In Proceedings of the 1st Workshop on Blockchain-enabled Networked Sensor Systems (pp. 19-24).

[20] Buterin V. (2020): "An Incomplete Guide to Rollups". Ethereum Foundation Blog.

[21] N. B. Truong, K. Sun, G. M. Lee and Y. Guo: "GDPR-Compliant Personal Data Management: A Blockchain-Based Solution", in IEEE™ Transactions on Information Forensics and Security, vol. 15, pp. 1746-1761, 2020.

[22] Vitalik Buterin, Yoav Weiss, Dror Tirosh, Shahaf Nacson, Alex Forshtat, Kristof Gazso & Tjaden Hess (2021): "ERC-4337: Account Abstraction Using Alt Mempool", Ethereum Research Forum.

[23] Caldarelli G. & Ellul J. (2022): "Trusted Academic Transcripts on the Blockchain: A Systematic Literature Review", Applied Sciences, 12(3), 1022.

[24] Cardano Foundation (2021): "Alonzo: Smart Contracts in Cardano".

[25] Chakraborty P., Shahriyar R., Iqbal A., & Bosu A. (2021): "Understanding the software development practices of blockchain projects: A survey", In Proceedings of the 14th Innovations in Software Engineering Conference (pp. 1-11).

[26] Chakravarty M. M., Chapman J., MacKenzie K., Melkonian O., Peyton Jones M. & Wadler P. (2020): "Native custom tokens in the extended UTXO model", In International Symposium on Leveraging Applications of Formal Methods (pp. 89-111). Springer, Cham.

[27] Chakravarty M. M., Coretti S., Fitzi M., Gazi P., Kant P., Kiayias A., & Russell A. (2020): "Hydra: Fast isomorphic state channels", In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (pp. 1895-1912).

[28] Chen H., Pendleton M., Njilla L., & Xu S. (2020): "A Survey on Ethereum Systems Security: Vulnerabilities, Attacks, and Defenses", ACM Computing Surveys (CSUR), 53(3), pp. 1-43.

[29] Chen J., & Micali S. (2020): "Algorand", arXiv preprint arXiv:1607.01341v9.

[30] IEEE™ P3801/D3.1 (November 2021): "IEEE Draft Standard for Blockchain-based Electronic Contracts", ISBN:978-1-5044-8193-9, IEEE™ published.

[31] Chen T., Li X., Luo X. & Zhang X. (2020): "Under-optimized smart contracts devour your money", In 2020 IEEE™ 27th International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp. 380-384). IEEE™.

[32] Cimphony: "Smart Contract Oracles: Complete Guide 2024", Cimphony, 2024.

[33] Cosmos Network (2021): "CosmJS: The Swiss Army knife to power JavaScript based client solutions", Developer in github.

NOTE: Retrieved from http://tutorials.cosmos.network/tutorials/7-cosmjs/2-first-steps.html.

[34] CosmWasm Team (2023): "CosmWasm: Pushing the Boundaries of Smart Contract Innovation", Medium.

[35] CSIRO Research: "Factory Contract", Blockchain Patterns, n.d.

[36] DevX: "Model View Controller - Glossary", DevX, 2023.

[37] Di Angelo M. & Salzer G. (2020): "Characterizing types of smart contracts in the Ethereum landscape", In International Conference on Financial Cryptography and Data Security (pp. 389-404). Springer, Cham.

[38] DigitalOcean: "Observer Design Pattern in Java", 2020.

[39] ETSI GS PDL 011: "Permissioned Distributed Ledger (PDL); Specification of Requirements for Smart Contracts' architecture and security".

[40] Yongfeng Huang, Yiyang Bian, Renpu Li, J. Leon Zhao, Peizhong Shi (2021): "Smart contract security: A software lifecycle perspective", IEEE™ Access, 9, 78365-78384.

[41] DucManhPhan: "Observer pattern", 2020.

[42] Eberhardt J., & Tai S. (2020): "On or off the blockchain? Insights on off-chaining computation and data", In European Conference on Service-Oriented and Cloud Computing (pp. 3-15). Springer, Cham.

[43] Wang H., Cen Y., & Li X. (2020): "Blockchain Router: A Cross-Chain Communication Protocol", In 6th International Conference on Information Security and Privacy (ICISP 2020) (pp. 173-177).

[44] Ethereum published (2020): "ERC-721 Non-Fungible Token Standard", Ethereum Improvement Proposals.

[45] ETSI GS PDL 012: "Permissioned Distributed Ledger (PDL); Reference Architecture".

[46] Ethereum published (2024): "ERC-20 Non-Fungible Token Standard".

[47] Ferretti S. & D'Angelo G. (2020): "On the Ethereum blockchain structure: A complex networks theory perspective", Concurrency and Computation: Practice and Experience, 32(12), e5493.

[48] Gao J., Liu H., Liu C., Li Q., Guan Z., & Chen Z. (2020): "EASYFLOW: Keep Ethereum Away from Overflow", In 2020 IEEE™/ACM 42nd International Conference on Software Engineering (ICSE) (pp. 1345-1357).

[49] GeeksforGeeks: "Decorator Design Pattern".

[50] GeeksforGeeks: "Strategy Design Pattern", 2022.

[51] Goforth C. R. (2020): "The Lawyer's Cryptionary: A Resource for Talking to Clients about Crypto-Transactions", University of Illinois Law Review, 2020(1), pp. 99-156.

[52] Grishchenko I., Maffei M. & Schneidewind C. (2020): "Foundations and tools for the static analysis of Ethereum smart contracts", In International Conference on Computer Aided Verification (pp. 51-78). Springer, Cham.

[53] Diego Marmsoler, Achim D. Brucker (March 2025): "Isabelle/Solidity: A deep embedding of Solidity in Isabelle/HOL", Formal Aspects of Computing, Volume 37, Issue 2, Article No.: 15, Pages 1 – 56.

[54] Gudgeon L., Moreno-Sanchez P., Roos S., McCorry P. & Gervais A. (2020): "SoK: Layer-Two Blockchain Protocols", In International Conference on Financial Cryptography and Data Security (pp. 201-226). Springer, Cham.

[55] HCL Software: "Model-View-Controller design pattern", 2022.

[56] Hedera: "Smart Contract Design Patterns Explained", n.d.

[57] Herlihy M. (2018): "Atomic cross-chain swaps", In Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing.

[58] Hildenbrandt E., Saxena M., Zhu X., Daian P., Guth D., Rosu G. (2020): "KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine", IEEE™ Computer Society.

[59] Hofman D., Lemieux V. L., Joo A. & Batista D. A. (2019): "The margin between the edge of the world and infinite possibility": Blockchain, GDPR and information governance", Records Management Journal.

[60] Sayyed Usman Ahmed, Abutalha Danish, Nesar Ahmad &Tameem Ahmad. (2024): "Smart Contract Generation through NLP and Blockchain for Legal Documents", Procedia Computer Science.

[61] Messias J., Pahari V., Chandrasekaran B., Gummadi K. P. & Loiseau P.: "Understanding Blockchain Governance: Analyzing Decentralized Voting to Amend DeFi Smart Contracts", arXiv:2305.17655v2 [cs.CR] 04 January 2024.

[62] Oshani Seneviratne: "The Feasibility of a Smart Contract 'Kill Switch'", arXiv:2407.10302v1 [cs.CR] 14 July 2024.

[63] OpenZeppelin: "The State of Smart Contract Upgrades".

[64] H. Kunkcu, K. Koc, A. P. Gurgun, H. H. Dagou: "Operational Barriers against the Use of Smart Contracts in Construction Projects", Turkish Journal of Civil Engineering, 2023, pp. 81-106, Paper 747.

[65] Yusof Z. B., Wan Haniff W. A. A., Saripan H., Jayabalan S. J. K. & Ab Halim A. H. (2024): "Regulatory Framework on Smart Contracts: A Comparative Analysis", Information Management and Business Review, 16(2(I), pp. 221-230.

[66] Yongshun Xu, Heap-Yih Chong, Ming Chi: "A Review of Smart Contracts Applications in Various Industries: A Procurement Perspective".

[67] Amit Kothari: "What are smart contracts on the blockchain?".

[68] Fortune Business Insights: "Smart Contracts Market Size".

[69] Frontiers in Research Metrics and Analytics: "Decentralized governance and artificial intelligence policy with blockchain-based voting in federated learning".

[70] Taherdoost H. (2023): "Smart Contracts in Blockchain Technology: A Critical Review". Information, 14(2), 117.

[71] Hu Y., Liyanage M., Mansoor A., Thilakarathna K., Jourjon G. & Seneviratne A. (2019): "Blockchain-based Smart Contracts-Applications and Challenges", IEEE™ Transactions on Services Computing.

[72] Huang Y., Bian Y., Li R., Zhao J. L. & Shi P. (2021): "Smart contract security: A software lifecycle perspective", IEEE™ Access, 9, 11613-11621.

[73] Wenjin Yang, Meng Ao, Mingzhi Gao, Chunhai Li and Yongqing Chen (2024): "CMSS: A High-Performance Blockchain Storage System with Horizontal Scaling Support", In journal of Electronics 2024, 13(10), 1854.

[74] A. S. Yahaya, N. Javaid, M. U. Javed, A. Almogren and A. Radwan: "Blockchain-Based Secure Energy Trading With Mutual Verifiable Fairness in a Smart Community", in IEEE™ Transactions on Industrial Informatics, vol. 18, no. 11, pp. 7412-7422, 2022.

[75] IncusData: "Your Guide to Design Patterns - Observer Pattern", 2024.

[76] Infuy: "Design Patterns for Smart Contracts", n.d.

[77] IOHK (2024): "Unlocking more opportunities with PlutusV3", Cardano Documentation.

NOTE: Retrieved from https://docs.cardano.org/developer-resources/welcome.

[78] Yihao Guo, Minghui Xu, Xiuzhen Cheng, Dongxiao Yu,;Wangjie Qiu;Gang Qu, Weibing Wang and Mingming Song. (2024): "zkCross: A Novel Architecture for Cross-Chain Privacy-Preserving Auditing", In Proceedings of the 2024 33rd USENIX Security Symposium.

[79] Jin H., Wang Z., Wen M., Dai W., Zhu Y., Zou D.: "Aroc: An Automatic Repair Framework for On-Chain Smart Contracts", IEEE™ Trans. Softw. Eng. 2022, 48, 4611–4629.

[80] Kalra S., Goel S., Dhawan M. & Sharma S. (2018): "ZEUS: Analyzing safety of smart contracts", In Network and Distributed Systems Security (NDSS) Symposium 2018.

[81] Kamilaris A., Fonts A. & Prenafeta-Boldú F. X. (2019): "The rise of blockchain technology in agriculture and food supply chains", Trends in Food Science & Technology, 91, pp. 640-652.

[82] King S. & Nadal S. (2012): "PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake".

[83] Kosba A., Miller A., Shi E., Wen Z. & Papamanthou C. (2020): "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts", IEEE™ Transactions on Dependable and Secure Computing, 18(5), 2337-2355.

[84] Kosba A., Zhao Z., Kattis A., Wan C. & Miller A. (2020): "Zexe: Enabling Decentralized Private Computation", In 2020 IEEE™ Symposium on Security and Privacy (SP) (pp. 947-964). IEEE™.

[85] Martin Azpiroz (2024): "Deep Dive into Cosmos Inter Blockchain Communication Protocol - IBC".

[86] Ladleif J. & Weske M. (2021): "A unifying model of legal smart contracts", In International Conference on Advanced Information Systems Engineering (pp. 323-337). Springer, Cham.

[87] Célio Gil, Gouveia Rodrigues (2020): "Property-Based Testing of ERC-20 Smart Contracts", In Proceedings of the ACM Guide books.

[88] Monika di Angelo & Gernot Salzer (2020): "Characteristics of Wallet Contracts on Ethereum", IEEE™ Access, 2nd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS).

[89] Pyrros Chaidos & Aggelos Kiayias (2024): "Mithril: Stake-Based Threshold Multisignatures", published by springer: International Conference on Cryptology and Network Security SpringerLink.

[90] Li X., Jiang P., Chen T., Luo X. & Wen Q. (2020): "A survey on the security of blockchain systems", Future Generation Computer Systems, 107, pp. 841-853.

[91] Liu C., Liu H., Cao Z., Chen Z., Chen B. & Roscoe B. (2021): "ReGuard: finding reentrancy bugs in smart contracts", In 2021 IEEE™/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion) (pp. 65-69).

[92] Liu Y., Lu Q., Xu X., Zhu L. & Yao H. (2020): "Applying design patterns in smart contracts: A case study on a blockchain-based traceability system", IEEE™ Access, 8, 119707-119724.

[93] Liu Y., Lu Q., Xu X., Zhu L. & Yao H. (2018): "Applying Design Patterns in Smart Contracts", In 2018 International Conference on Blockchain (Blockchain), published by Springer.

[94] Lu Q., Xu X., Liu Y., Weber I., Zhu L. & Zhang W. (2021): "uBaaS: A unified blockchain as a service platform", Future Generation Computer Systems, 101, pp. 564-575.

[95] Macrinici D., Cartofeanu C. & Gao S. (2022): "Smart contract applications within blockchain technology: A systematic mapping study", Telematics and Informatics, 57, 101569.

[96] Mammadzada K., Iqbal M., Milani F., García-Bañuelos L. & Matulevičius R. (2020): "Blockchain oracles: A framework for blockchain-based applications", In Business Process Management: Blockchain and Central and Eastern Europe Forum (pp. 19-34). Springer, Cham.

[97] Metana: "Smart Contract Design Patterns in Solidity Explained!".

[98] "Designing the infrastructure persistence layer", .NET Microsoft Learn, 2020.

[99] "N-tier architecture style", Azure Architecture Center, Microsoft Learn, 2020.

[100] "Observer design pattern", Microsoft Learn, 2020.

[101] Mohanta B. K., Panda S. S. & Jena D. (2020): "An overview of smart contract and use cases in blockchain technology", In 2020 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT) (pp. 1-4). IEEE™.

[102] Keerthi Nelaturu, Anastasia Mavridoul, Andreas Veneris & Aron Laszka (2020): "Verified Development and Deployment of Multiple Interacting Smart Contracts with VeriSolid", In 2020 IEEE™ International Conference on Blockchain and Cryptocurrency (ICBC).

[103] Nijsse J. & Litchfield A. (2020): "A Taxonomy of Blockchain Consensus Methods", Cryptography, 4(4), 32.

[104] Nizamuddin N., Salah K., Azad M. A., Arshad J. & Rehman M. H. (2021): "Decentralized document version control using ethereum blockchain and IPFS", Computers & Electrical Engineering, 76, pp. 183-197.

[105] Jawad Khan: "The Factory Pattern and its Application in Solidity", Inclinedweb, 2022.

[106] Ojo A. & Adebayo S. O. (2017): "Blockchain as a next generation government information infrastructure: A review of initiatives in D5 countries", In Proceedings of the 18th Annual International Conference on Digital Government Research (pp. 503-504).

[107] ETSI GS PDL 022 (V1.1.1): "Permissioned Distributed Ledgers (PDL); PDL in Wholesale Supply Chain Management".

[108] Politou E., Casino F., Alepis E. & Patsakis C. (2020): "Blockchain Mutability: Challenges and Proposed Solutions", IEEE™ Transactions on Emerging Topics in Computing, 9(4), pp. 1972-1986.

[109] Polkadot (2021): "Ink! 3.0 Release Notes".

[110] Popchev I., Radeva I.; Doukovska L.: "Oracles Integration in Blockchain-Based Platform for Smart Crop Production Data Exchange", Electronics, vol. 12, no. 10, 2023, p. 2244.

[111] Praitheeshan P., Pan L., Yu J., Liu J. & Doss R. (2020): "Security analysis methods on Ethereum smart contract vulnerabilities: a survey", IEEE™ Access, 8, 24196-24212.

[112] Turki Ali Alghamdi, Rabiya Khalid & Nadeem Javaid (2024): "A Survey of Blockchain Based Systems: Scalability Issues and Solutions, Applications and Future Challenges", IEEE™ Access, 12.

[113] Radomski W., Cooke A. & Castonguay P. (2020): "ERC-1155 Multi Token Standard", Ethereum Improvement Proposals.

[114] Raiden Network (2020): "Raiden Network (RDN): A scaling solution on top of the Ethereum blockchain designed to enable fast, low-cost, and scalable payments".

[115] Ramos, D. & Pérez-Solà C. (2022): "Layer-2 blockchain scaling: a survey", ACM Computing Surveys, 55(1), pp. 1-37.

[116] Rapid Innovation (2023): "Implementing smart contracts: Challenges and solutions for businesses", Blockchain Innovations Journal, 8(1), pp. 34-48.

[117] Rapid Innovation (2023): "The future of smart contracts: Trends and predictions", Decentralized Systems Review, 10(2), pp. 78-95.

[118] Thi Thu Ha Doan & Peter Thiemann (2024): "A Formal Verification Framework for Tezos Smart Contracts Based on Symbolic Execution", In International Conference Lecture Notes in Computer Science, vol 15194. Springer, Singapore.

[119] Rodler M., Li W. Karame G. O. & Davi L. (2021): "Sereum: Protecting existing smart contracts against re-entrancy attacks", In Proceedings of the 2021 Network and Distributed System Security Symposium. Internet Society.

[120] Saad M., Spaulding J., Njilla L., Kamhoua C., Shetty S., Nyang D. & Mohaisen A. (2020): "Exploring the attack surface of blockchain: A systematic overview", IEEE™ Communications Surveys & Tutorials, 22(3), 1977-2008.

[121] Saleh F. (2020): "Blockchain Without Waste: Proof-of-Stake", Review of Financial Studies, 33(9), pp. 4324–4377.

[122] Samreen N. & Alam M. (2021): "A Survey of Security Vulnerabilities in Ethereum Smart Contracts", International Journal of Information Security and Privacy.

[123] Sayeed S., Marco-Gisbert H. & Caira T. (2020): "Smart contract: Attacks and protections", IEEE™ Access, 8, 24416-24427.

[124] Schalm Z. & Radomski W. (2021): "ERC-2981: NFT Royalty Standard", Ethereum Improvement Proposals.

[125] Jan Vanhoof & Tom Van Cutsem (2025): "A Comparative Study of Rust Smart Contract SDKs for Application-Specific Blockchains", In International Conference on Coordination Languages and Models. SpringerLink.

[126] SCSFG: "System Design - Smart Contract Security Field Guide", SCSFG, n.d.

[127] Sedlmeir J., Buhl H., Fridgen G. & Keller R. (2020): "The Energy Consumption of Blockchain Technology: Beyond Myth", Business & Information Systems Engineering, 62(6), pp. 599-608.

[128] Shala B., Trick U., Lehmann A., Ghita B. & Shiaeles S. (2020): "Novel trust consensus protocol and blockchain-based trust evaluation system for M2M application services", Internet of Things, 9, 100155.

[129] Insaf Achour, Hanen Idoudi, Samiha Ayed (2025): "Access Control Modeling and Validation for Ethereum Smart Contracts", Journal of Concurrency and computation, Volume 37, Issue 12-14, 25 June 2025, Concurrency and Computation: Practice and Experience - Wiley Online Library.

[130] Shivam Srivastava (2024): "Smart contracts: The Backbone of Decentralized Applications", Tatum Blog (January 15, 2024), Blockchain Technology Insights, 5(1), pp. 22-30.

[131] Siris V. A., Dimopoulos D., Fotiou N., Voulgaris S. & Polyzos G. C. (2020): "Interledger smart contracts for decentralized authorization to constrained things", IEEE™ Access, 8, 90555-90566.

[132] StackOverflow: "Broker architectural pattern", StackOverflow, 2020.

[133] SourceMaking: "Decorator Design Pattern", SourceMaking, 2020.

[134] Geeks for Geeks: "Pipe and Filter Architecture - System Design", 2024.

[135] SourceMaking: "Strategy Design Pattern", SourceMaking, 2020.

[136] Stackify: "What is N-Tier Architecture? How It Works, Examples, Tutorials & More", Stackify, 2020.

[137] Stoll C., Klaaßen L. & Gallersdörfer U. (2020): "The Carbon Footprint of Bitcoin", Joule, 3(7), pp. 1647-1661.

[138] Tezos Agora (2021): "Hangzhou — New Proposal for Tezos' Eighth Protocol Upgrade", Tezos Agora Forum.

[139] Tezos Foundation (2024): "The Evolution of Smart Contracts: What's Next?".

[140] Truong N. B., Sun K., Lee G. M. & Guo Y. (2020): "GDPR-Compliant Personal Data Management: A Blockchain-Based Solution", IEEE™ Transactions on Information Forensics and Security, 15, 1746-1761.

[141] Tsankov P., Dan A., Drachsler-Cohen D., Gervais A., Buenzli F. & Vechev M. (2018): "Securify: Practical security analysis of smart contracts", In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (pp. 67-82).

[142] Geeks for Geeks: "Broker Pattern".

[143] Tutorials Point: "Design Patterns - Decorator Pattern", 2023.

[144] Tutorials Point: "Design Patterns - Strategy Pattern", 2023.

[145] Udokwu C., Kormiltsyn A., Thangalimodzi K. & Norta A. (2021): "The State of the Art for Blockchain-Enabled Smart-Contract Applications in the Organization", In Informatics (Vol. 8, No. 2, p. 36). MDPI.

[146] Vogelsteller F. & Buterin V. (2020): "ERC-777: Token Standard", Ethereum Improvement Proposals.

[147] Qiuhong Zheng, Yi Li, Ping Chen & Xinghua Dong (2018): "An Innovative IPFS-Based Storage Model for Blockchain", IEEE™ Access, 2018 IEEE/WIC/ACM International Conference on Web Intelligence (WI).

[148] Wang S., Ouyang L., Yuan Y., Ni X., Han X. & Wang F. Y. (2020): "Blockchain-Enabled Smart Contracts: Architecture, Applications, and Future Trends", IEEE™ Transactions on Systems, Man, and Cybernetics: Systems, 51(1), 274-291.

[149] Sefa Akca, Ajitha Rajan & Chao Peng (2019): "SolAnalyser: A Framework for Analysing and Testing Smart Contracts", In 2019 IEEE™ 2019 26th Asia-Pacific Software Engineering Conference (APSEC).

[150] Werbach K. & Cornell N. (2020): "Contracts Ex Machina", Duke Law Journal, 67(2), pp. 313-382.

[151] Werner S. M., Perez D., Gudgeon L., Klages-Mundt A., Harz D. & Knottenbelt W. J. (2021): "SoK: Decentralized Finance (DeFi)", arXiv preprint arXiv:2101.08778.

[152] Wright A. & De Filippi P. (2020): "Decentralized Blockchain Technology and the Rise of Lex Cryptographia", SSRN Electronic Journal.

[153] Xiao Y., Zhang N., Lou W. & Hou Y. T. (2020): "A survey of distributed consensus protocols for blockchain networks", IEEE™ Communications Surveys & Tutorials, 22(2), 1432-1465.

[154] Xu C., Zhang C. & Xu J. (2021): "vChain: Enabling Verifiable Boolean Range Queries over Blockchain Databases", In Proceedings of the 2021 International Conference on Management of Data (pp. 1137-1149).

[155] Xu L., Xu C., Zhao P. & Zheng X. (2021): "Enabling Cross-Chain Transactions: A Decentralized Cryptocurrency Exchange Protocol", Frontiers in Blockchain, 4, 627442.

[156] Xu X., Weber I., Staples M., Zhu L., Bosch J., Bas, L., Pautasso C. & Rimba P. (2020): "A Taxonomy of Blockchain-Based Systems for Architecture Design", IEEE™ Transactions on Software Engineering, 47(7), 1464-1487.

[157] Xu C., Zhang C, J Xu, B Choi & Wang G. (2021): "Authenticated Keyword Search in Scalable Hybrid-Storage Blockchains", IEEE™ 2021 IEEE 37th International Conference on Data Engineering (ICDE).

[158] Yazdinejad A., Srivastava G., Parizi R. M., Dehghantanha A., Choo K. K. R. & Aledhari M. (2020): "Decentralized Authentication of Distributed Patients in Hospital Networks using Blockchain", IEEE™ Journal of Biomedical and Health Informatics, 24(8), 2146-2156.

[159] Yin W., Wen Q., Li W., Zhang H. & Jin Z. (2021): "An anti-quantum transaction authentication approach in blockchain", IEEE™ Access, 9, 16072-16080.

[160] Yue X., Wang H., Jin D., Li M. & Jiang W. (2021): "Healthcare Data Gateways: Found Healthcare Intelligence on Blockchain with Novel Privacy Risk Control", Journal of Medical Systems, 45(1), pp. 1-12.

[161] Zamyatin A., Al-Bassam M., Zindros D., Kokoris-Kogias E., Moreno-Sanchez P., Kiayias A. & Knottenbelt W. J. (2021): "SoK: Communication Across Distributed Ledgers", In International Conference on Financial Cryptography and Data Security (pp. 3-36). Springer, Berlin, Heidelberg.

[162] Zhang L. & Wen J. (2017): "An IoT electric business model based on the protocol of bitcoin", In 2017 2nd International Conference on Artificial Intelligence and Industrial Engineering (AIIE 2017) (pp. 13-17). Atlantis Press.

[163] Md. Abdur Rahman, Khalid Abualsaud, Stuart Barnes, Mamunur Rashid & Syed Maruf Abdullah (2020): "A Natural User Interface and Blockchain-Based In-Home Smart Health Monitoring System", IEEE™ 2020 IEEE International Conference on Informatics, IoT, and Enabling Technologies (ICIoT).

[164] Zhang P., White J., Schmidt D. C., Lenz G. & Rosenbloom S. T. (2020): "FHIRChain: Applying Blockchain to Securely and Scalability Share Clinical Data", Computational and Structural Biotechnology Journal, 16, pp. 267-278.

[165] Keerthi Nelaturu, Anastasia Mavridou, Emmanouela Stachtiari, Andreas Veneris & Aron Laszka (2023): "Correct-by-Design Interacting Smart Contracts and a Systematic Approach for Verifying ERC20 and ERC721 Contracts with VeriSolid", IEEE™ Access, 20(4).

[166] Nikolaos Kostopoulos, Konstantinos Sekaras, Nicolas Sarafidis, Ioannis Vlacho, Georga Kopoulos, Alexi Anania, Wendy Charles, Catarina Ferreira da Silva, Amit Joshi, Urko Larrañaga, Jim Mason, Inigo More, Daniel Szego & Ingrid Vasiliu-Feltes. (2022): "The current state of interoperability between blockchain networks", EU Blockchain Observatory and Forum.

[167] Hu Xiong, Ye Xia, Yaxin Zhao, Abubaker Wahaballa & Kuo-Hui Yeh (2025): "Heterogeneous Privacy-Preserving Blockchain-Enabled Federated Learning for Social Fintech", IEEE™ Access.

[168] Zheng Q., Li Y., Chen P. & Dong X. (2021): "An Innovative IPFS-Based Storage Model for Blockchain", In 2021 IEEE™ World Congress on Services (SERVICES) (pp. 49-53).

[169] Zheng Z., Xie S., Dai H. N., Chen W., Chen X., Weng J. & Imran M. (2021): "An overview on smart contracts: Challenges, advances and platforms", Future Generation Computer Systems, 105, pp. 475-491.

[170] Faiez Altamimi, Waqar Asif & Muttukrishnan Rajarajan (2020): " DADS: Decentralized (Mobile) Applications Deployment System Using Blockchain : Secured Decentralized Applications Store", IEEE™ 2020 International Conference on Computer, Information and Telecommunication Systems (CITS).

[171] Zhou W., Huang Y. & Zhang L. (2020): "Blockchain vulnerabilities and recent security challenges", Journal of Cybersecurity, 12(4), pp. 200-215.

[172] Zhu L., Wu Y., Gai K. & Choo K. K. R. (2021): "Controllable and trustworthy blockchain-based cloud data management", Future Generation Computer Systems, 114, pp. 283-295.

[173] Zhu S., Cai Z., Hu H., Li Y. & Li W. (2021): "zkCrowd: A hybrid blockchain-based crowdsourcing platform", IEEE™ Transactions on Industrial Informatics, 17(8), 5829-5839.

[174] Epherem Merete Sifra (2022): "Security Vulnerabilities and Countermeasures of Smart Contracts: A Survey", IEEE™ 2022 IEEE International Conference on Blockchain (Blockchain).

[175] Zhu Y., Qin Y., Zhou Z., Song X., Liu G. & Chu W. C. C. (2020): "Digital asset management with distributed permission over blockchain and attribute-based access control", In 2020 IEEE™ International Conference on Services Computing (SCC) (pp. 360-367). IEEE™.

[176] Zohar A. & Eyal I. (2020): "The Bitcoin Backbone Protocol with Chains of Variable Difficulty", Communications of the ACM, 63(3), pp. 95-102.

[177] Zou W., Lo D., Kochhar P. S., Le X. B. D., Xia X., Feng Y., Chen Z. & Xu B. (2021): "Smart Contract Development: Challenges and Opportunities", IEEE™ Transactions on Software Engineering, 47(10), 2084-2106.

2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long-term validity.

The following referenced documents may be useful in implementing an ETSI deliverable or add to the reader's understanding, but are not required for conformance to the present document.

[i.1] Buterin,V (2021): "Detailed technical explanation of Rollup expansion principles and challenges".

[i.2] Buterin V., et al. (2021): "Ethereum 2.0: A Sharding-Based Blockchain Protocol", Proceedings of the ACM Conference on Advances in Cryptographic Technology, pp. 78-95.

[i.3] Coleman J. & Horne L. (2021): "State Channels: An Overview and State of the Art", ACM Computing Surveys, 54(1), pp. 1-30.

[i.4] Feist J., Grieco G. & Groce A. (2020): "Slither: A Static Analysis Framework For Smart Contracts", In 2020 IEEE™/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion) (pp. 85-88).

[i.5] Feng Y., Torlak E. & Bodík R. (2020): "ETHPLOIT: From Fuzzing to Efficient Exploit Generation against Smart Contracts", In 2020 IEEE™/ACM 42nd International Conference on Software Engineering (ICSE) (pp. 1102-1114).

[i.6] Ferreira J. F., Cruz P., Durieux T. & Abreu R. (2020): "SmartBugs: A Framework to Analyze Solidity Smart Contracts", In 2020 35th IEEE™/ACM International Conference on Automated Software Engineering (ASE), 582-586.

[i.7] Ferreira J. F., Cruz P., Durieux T. & Abreu R. (2020): "SmartBugs: A Framework to Analyze Solidity Smart Contracts", In 2020 IEEE™ 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), 390-400.

[i.8] Gao Z. & Jiang L. (2021): "Automated Smart Contract Testing: Techniques and Challenges", ACM Transactions on Software Engineering and Methodology, 30(3), pp. 1-28.

[i.9] Chen Y. & Zhang F. (2020): "State Separation for Flexible Upgrades in Ethereum Smart Contracts", ACM Conference on Advances in Financial Technologies, pp. 183-195.

[i.10] He N., Zhang R., Wu L., Wang H., Luo X. & Guo Y. (2022): "EthPloit: From Fuzzing to Efficient Exploit Generation against Smart Contracts", In 2022 IEEE™ Symposium on Security and Privacy (SP), pp. 778-796.

[i.11] Jiang B., Liu Y. & Chan W. K. (2021): "ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection", In 2021 IEEE™/ACM 43rd International Conference on Software Engineering (ICSE), 606-617.

[i.12] John K., Yin H., Ektefa M. & Sakurai K. (2021): "Rollups on Trial: Optimistic vs. Zero-Knowledge", In 2021 IEEE™ International Conference on Blockchain (Blockchain) (pp. 336-343).

[i.13] Johnson K. & Brown M. (2023): "Modular Smart Contracts: A Compositional Approach to Contract Development", International Conference on Software Engineering (ICSE), pp. 1456-1467.

[i.14] Johnson L. & Smith K. (2022): "Rollups on Ethereum: A Comprehensive Survey", IEEE™ Transactions on Blockchain Technology, 3(2), 45-62.

[i.15] Lahiri S. K., Chen S., Wang Y. & Dillig I. (2020): "VERISOL: Verification of Solidity Programs Using SMT Solvers", 2020 Formal Methods in Computer Aided Design (FMCAD), pp. 60-68.

[i.16] Li Y., Liao C. & Zhang Y. (2020): "A Modular Design for Ethereum Smart Contracts Verification", In Formal Methods and Software Engineering (pp. 168-183). Springer.

[i.17] Nguyen T. D., Pham L. H., Sun J., Lin Y. & Minh Q. T. (2020): "sFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts", In 2020 IEEE™/ACM 42nd International Conference on Software Engineering (ICSE) (pp. 778-788).

[i.18] OpenZeppelin (2023): "OpenZeppelin Contracts Documentation".

[i.19] Patel M. & Gupta R. (2023): "Efficient Data Compression for Blockchain Storage Optimization", IEEE™ Access, 11, 12345-12360.

[i.20] ETSI GR PDL 018 (V1.2.1): "Permissioned Distributed Ledger (PDL); Redactable Distributed Ledgers".

[i.21] Rodler M., Li W., Karame G. O. & Davi L. (2020): "State Separation for Dynamic Smart Contract Upgrades", 2020 IEEE™ International Conference on Blockchain (Blockchain), 185-194.

[i.22] Sayeed S. & Marco-Gisbert H. (2021): "Proxy Patterns for Upgradeable Smart Contracts: A Survey", 2021 IEEE™ International Conference on Blockchain (Blockchain), 197-204.

[i.23] Singh A. & Parizi R. M. (2021): "Upgradeable Smart Contracts: A Survey", IEEE™ International Conference on Blockchain and Cryptocurrency (ICBC), 132-139.

[i.24] So S., Lee M., Park J., Lee H. & Oh H. (2020): "VeriSmart: A Highly Precise Safety Verifier for Ethereum Smart Contracts", In 2020 IEEE™ Symposium on Security and Privacy (SP) (pp. 1678-1694).

[i.25] Wang L. & Liu X. (2022): "Parametric Smart Contracts: A New Paradigm for Blockchain-Based Agreement", IEEE™ Transactions on Services Computing, 15(4), 2234-2247.

[i.26] Geeks for Geeks: "Pipe and filter", 2024.

[i.27] Wood G. (2020): "Polkadot: Vision for a Heterogeneous Multi-Chain Framework", White Paper.

[i.28] Zhang P., Xiao F. & Luo X. (2021): "SmartShield: Automatic Smart Contract Protection Made Easy", In 2021 IEEE™ 32nd International Symposium on Software Reliability Engineering (ISSRE) (pp. 25-35).

[i.29] Zhang Y. & Liu J. (2020): "Decentralized Storage: The Backbone of the Blockchain", Journal of Network and Computer Applications, 167, 102738.

[i.30] ETSI GR PDL 032 V1.1.1 (2025-04): "Permissioned Distributed Ledger (PDL); Artificial Intelligence for Permissioned Distributed Ledger".

[i.31] Marić O., Sprenger C. & Basin D. (2021): "What does it mean that Ouroboros is formally verified?", Cardano Stack Exchange.

[i.32] J. Wang, Y. Yu, J. Zhao and H. Yu: "The Blockchain Name System (BNS): Polymorphic Identification Technology for Blockchain Supervision", 2021 4th International Conference on Hot Information-Centric Networking (HotICN).

[i.33] Alper Saraç: "What is Repository Pattern?", LinkedIn, 2020.

3 Definition of terms, symbols and abbreviations

3.1 Terms

For the purposes of the present document, the following terms apply:

atomicity: all-or-nothing principle in smart contract execution, where a transaction either completes fully or has no effect at all

autonomy: ability of smart contracts to operate independently and execute predefined actions without external intervention

consensus mechanism: protocol used in blockchain networks to ensure agreement on the state of the ledger among distributed participants

decentralization: distribution of control and decision-making across a network of nodes, rather than relying on a central authority

determinism: property of smart contracts that ensures consistent execution across all nodes in the network, given the same input

gas: unit of measurement for the computational effort required to execute operations in certain blockchain networks, often associated with transaction fees

immutability: property of smart contracts that ensures their code cannot be altered once deployed on the blockchain, enhancing trust and security

interoperability: ability of smart contracts to communicate and interact with other contracts or systems across different blockchain networks

Layer-2: scaling solutions built on top of existing blockchain networks to improve transaction speed and reduce costs while leveraging the security of the underlying blockchain

Non-Fungible Token (NFT): unique digital asset represented on a blockchain, often used for digital art, collectibles, or other unique items

oracle: external service that provides real-world data to smart contracts, bridging the gap between on-chain and off-chain information

Permissioned Distributed Ledger (PDL): type of blockchain or distributed ledger technology where participation is restricted to authorized entities, as opposed to public blockchains

Proof-of-Stake (PoS): energy-efficient consensus mechanism where validators are chosen to create new blocks based on the amount of cryptocurrency they hold and are willing to "stake" as collateral

sharding: scaling solution designed to improve the capacity and transaction speed of blockchain networks

NOTE: Sharding in the context of smart contracts refers to a scaling solution designed to improve the capacity and transaction speed of blockchain networks, particularly Ethereum. It involves dividing the blockchain network into smaller, interconnected partitions called "shards," each capable of processing its own transactions and smart contracts in parallel.

smart contract: self-executing computer program stored on a distributed ledger system that automatically executes predefined actions when specific conditions are met, without the need for intermediaries

transparency: characteristic of smart contracts that allows all parties to verify the contract's code and its execution, promoting accountability and trust

3.2 Symbols

Void.

3.3 Abbreviations

For the purposes of the present document, the following abbreviations apply:

3GPP	3rd Generation Partnership Project
ABI	Application Binary Interface
API	Application Programming Interface
AVM	Algorand Virtual Machine
DAO	Decentralized Administered Organization
DApps	Decentralized Applications
DeFi	Decentralized Finance
DLT	Distributed Ledger Technology
DoS	Denial of Service
DPoS	Delegated Proof-of-Stake
ERC	Ethereum Request for Comments
ETSI	European Telecommunications Standards Institute
EVM	Earned Value Management
GDPR	General Data Privacy Regulation
GS	Group Specification
HTLC	Hashed Timelock Contract
IBC	Institute of Barristers' Clerks
IDE	Integrated Development Environment
IoT	Internet of Things
IPFS	InterPlanetary File System
ISG	Industry Specification Group
MVC	Model-View-Controller
NatSpec	Natural Language Specification
NFT	Non-Fungible Token
OOP	Object-Oriented Programming
PAB	Plutus Application Backend

PBFT	Practical Bysantine Fault Tolerance
PDL	Permissioned Distributed Ledger
PoA	Proof-of-Authority
PoS	Proof-of-Stake
QoS	Quality of Service
RBAC	Role-Based Access Control
SLA	Service Level Agreement
SMT	Satisfiability Modulo Theories
SNARK	Succinct Non-Interactive Argument of Knowledge
STARK	Scalable Transparent Argument of Knowledge
TEAL	Transaction Execution Approval Language
TEE	Trusted Execution Environment
UTXO	Unspent Transaction Output

4 Introduction to Smart Contracts

4.1 Introduction

Smart Contracts represent a paradigm shift in how agreements and transactions are conceptualized and executed in the digital age. At their core, Smart Contracts are self-executing computer programs stored on a distributed ledger system, wherein the outcome of any execution of the program is recorded on the distributed ledger. Unlike traditional contracts that rely on human interpretation and enforcement, Smart Contracts automatically execute predefined actions when specific conditions are met, without the need for intermediaries.

The primary purpose of a Smart Contract is to encode and automate contractual clauses, business logic, or any set of rules that govern interactions between parties. When deployed on a Permissioned Distributed Ledger (PDL), these contracts become immutable and transparent, ensuring that all participants can trust in their execution without relying on a central authority.

Key characteristics of Smart Contracts include:

1) **Automation:** Smart Contracts reduce the need for manual intervention, minimizing human error and streamlining processes.

2) **Transparency:** All parties can verify the contract's code and its execution, promoting trust and accountability.

3) **Immutability:** Once deployed, the contract's code cannot be altered, ensuring consistency and reducing the risk of tampering.

4) **Efficiency:** By removing intermediaries and automating processes, transactions can be faster and more cost-effective.

5) **Programmability:** Smart Contracts can handle complex logic, interact with other contracts, manage digital assets, and even control real-world processes through IoT devices.

While Smart Contracts offer significant advantages, it is crucial to understand that they are not a panacea. Their effectiveness depends on careful design, thorough testing, and appropriate use cases. The immutable nature of Smart Contracts means that errors in code can have significant consequences, underscoring the importance of robust development and auditing practices.

Smart Contracts find applications across various industries, including finance, supply chain management, healthcare, and government services. They enable new business models and facilitate complex multi-party agreements in ways that were not previously possible.

The present document explores the properties, architectures, and the challenges and opportunities Smart Contracts present in reshaping how entities conduct business and manage agreements in the digital era. The following clauses will provide a comprehensive overview of Smart Contract technology, its implementation in Permissioned Distributed Ledgers, and the considerations necessary for their effective design, deployment, and management.

4.2 Object-Oriented Paradigm

4.2.1 Introduction to OOP in Smart Contracts

Smart Contracts align closely with the principles of Object-Oriented Programming (OOP), providing a powerful framework for modelling complex systems and interactions. This paradigm is particularly suitable for representing contractual relationships and business logic in a distributed ledger environment in general and in Permissioned Distributed Ledger (PDL) in particular. Understanding and leveraging the object-oriented paradigm in the context of Smart Contracts is crucial for effective contract design, especially as contract complexity increases and as they interact with other contracts and external systems in the broader ecosystem of decentralized applications [148], [156].

[R1] Smart contracts **SHALL** implement key OOP concepts to enhance modularity, reusability, and maintainability.

[R2] The implementation of OOP concepts **SHALL NOT** compromise the security or efficiency of the smart contract.

[D1] Developers **SHOULD** follow established design patterns that leverage OOP concepts in smart contracts.

[D2] The use of OOP concepts **SHOULD** not introduce unnecessary complexity or gas costs.

[D3] Contracts **SHOULD** be designed with clear separation of concerns, using OOP principles to organize code logically.

[D4] Developers **SHOULD** consider the limitations of OOP in blockchain environments (e.g. all data is ultimately public) when applying these concepts.

[D5] Regular code reviews and audits **SHOULD** be conducted to ensure proper implementation of OOP concepts without introducing vulnerabilities.

4.2.2 Key OOP Concepts in Smart Contracts

4.2.2.1 Encapsulation

Smart Contracts encapsulate both data (state) and behaviour (functions) into a single unit, mirroring real-world contracts where terms and conditions are bundled with actions and obligations. This encapsulation helps in organizing code and maintaining data integrity [92].

[D6] Smart contracts **SHOULD** encapsulate state variables and functions that operate on them within a single contract.

[D7] Developers **SHOULD** use access modifiers (public, private, internal) to control the visibility of contract elements.

4.2.2.2 State and Behaviour

4.2.2.2.1 State

Internal storage, typically in the form of key-value pairs, is analogous to object fields in OOP. This method is often used to represent the current condition, referred to as state, of the smart contract. The state may be represented using other methods too, such as state-machines. Each instantiated smart contract maintains its own state. This means that even if multiple instances of the same contract type are created, each instance will have its own independent set of state variables and data.

[R3] The state of the contract **SHALL** be defined at all times.

[D8] Contracts **SHOULD** clearly separate state variables from functions that modify them.

[D9] State changes **SHOULD** be made through well-defined functions to maintain encapsulation.

[O1] A smart contract **MAY** be in one of the following states, from the list below:

1) **Pending:** The contract has been created but not yet deployed to the blockchain.
2) **Active:** The contract is deployed and operational, ready to execute its functions.
3) **Paused:** The contract's core functions are temporarily suspended, often used for maintenance or security reasons.
4) **Expired:** The contract has reached its predetermined end date or condition.
5) **Completed:** All required actions or conditions of the contract have been fulfilled.
6) **Terminated:** The contract has been deliberately ended before its natural completion, possibly due to agreed-upon conditions or external factors.
7) **Locked:** Certain functions or funds within the contract are temporarily inaccessible.
8) **Unlocked:** Previously locked functions or funds become accessible again.
9) **In Dispute:** There is a disagreement about the contract's execution, and it may be under review or arbitration.
10) **Upgraded:** The contract has been updated to a new version, often with improved functionality or security.
11) **Deprecated:** The contract is no longer recommended for use but may still be functional.
12) **Frozen:** All activities of the contract are halted, usually due to a critical issue or regulatory requirement.
13) **In Execution:** The contract is actively processing a transaction or function call.
14) **Awaiting Input:** The contract is waiting for specific data or action from participants to proceed.
15) **Failed:** The contract encountered an error and could not complete its intended function.

These states can vary widely depending on the specific implementation and purpose of the smart contract.

[O2] Developers **MAY** define custom states tailored to their particular use case.

4.2.2.2.2 Behaviour

The behaviour of a smart contract is defined through functions that specify the set of actions allowed for the given Smart Contract, with appropriate scope modifiers, similar to object methods in OOP.

The set of actions allowed for a given smart contract depends on its specific design and purpose.

[O3] A general list of common actions that **MAY** be implemented in various smart contracts follows:

1) **Deploy:** Initialize and publish the contract on the blockchain.
2) **Execute:** Trigger a specific function within the contract.
3) **Read:** Access and view contract data without modifying it.
4) **Write:** Modify contract data or state.
5) **Transfer:** Send tokens or assets from one address to another.
6) **Approve:** Grant permission for another address to interact with the contract on behalf of the approver.
7) **Mint:** Create new tokens or assets within the contract.
8) **Burn:** Destroy or remove tokens or assets from circulation.
9) **Stake:** Lock up tokens for a specific period, often for rewards or voting rights.

10) **Unstake:** Withdraw previously staked tokens.

11) **Claim:** Collect rewards, dividends, or other benefits from the contract.

12) **Vote:** Participate in governance decisions related to the contract.

13) **Propose:** Submit new proposals for contract changes or decisions.

14) **Upgrade:** Update the contract to a new version with improved functionality.

15) **Pause:** Temporarily suspend specific contract functions.

16) **Unpause:** Resume previously paused functions.

17) **Withdraw:** Remove funds or assets from the contract.

18) **Deposit:** Add funds or assets to the contract.

19) **Set Parameters:** Adjust configurable aspects of the contract.

20) **Terminate:** End the contract's operation permanently.

The actual set of actions for any given smart contract would be determined by its specific implementation and use case. Some contracts might have only a few of these actions, while others could have additional, more specialized functions.

[O4] Developers **MAY** define custom functions tailored to their particular use case.

4.2.2.3 Instantiation

Instantiation refers to the process of creating a specific instance or occurrence of a smart contract from its definition or template. Similar to objects in OOP, Smart Contracts are instantiated from a contract definition (akin to a class). When a smart contract is instantiated, it receives a unique identifier. This identifier distinguishes this particular instance of the contract from other instances of the same contract type. This instantiation process allows for the creation of multiple, independent instances of a smart contract, each with its own identity and state, while all instances share the same underlying code structure defined in the contract [177].

[R4] Smart contracts **SHALL** have a clear instantiation process, typically through a constructor function.

[R5] Once instantiated, each Smart Contract **SHALL** hold a unique identifier and maintain its own state.

4.2.2.4 Inheritance and Composition

Inheritance is a mechanism that allows a smart contract to inherit properties and methods from another contract. This enables the creation of more complex contracts built upon simpler ones. Many Smart Contract platforms support inheritance, allowing contracts to extend or inherit from other contracts. This facilitates code reuse, and the creation of more complex contracts built upon simpler ones.

Composition involves building complex objects or structures by combining simpler ones. It allows for the creation of contracts that contain or use other contracts as part of their functionality. A contract can include instances of other contracts and delegate certain tasks to them. While these are powerful and useful concepts, their implementation is not mandatory for all smart contract platforms but is recommended when possible and appropriate.

[D10] Developers **SHOULD** use inheritance to create hierarchies of contracts with shared functionality.

[D11] Composition **SHOULD** be used when a contract needs to use functionality from another contract without inheriting all its properties.

[D12] Multiple inheritance **SHOULD** be used cautiously to avoid complexity and potential conflicts.

[O5] Smart contract platforms **MAY** support Inheritance and Composition.

4.2.2.5 Polymorphism

Polymorphism in smart contracts refers to the ability to use a uniform interface to interact with different contract types. This allows different smart contracts to be treated in a similar way if they implement the same interface, even if their internal implementations differ. Through interfaces and abstract contracts, Smart Contracts can implement polymorphism, enabling uniform treatment of different contracts adhering to the same interface. This concept, borrowed from object-oriented programming, enhances the modularity and reusability of smart contract code [i.32].

[D13] Contracts **SHOULD** use interfaces to define common behaviour that can be implemented by multiple contracts.

[D14] Developers **SHOULD** leverage polymorphism to create more flexible and extensible contract systems.

[O6] Smart contracts **MAY** support Polymorphism.

4.2.2.6 Visibility and Access Control

Visibility and Access Control in smart contracts refers to mechanisms used to control the accessibility and modifiability of functions and state variables within the contract. Smart Contracts use access modifiers to control the visibility and accessibility of functions and state variables, similar to public, private, and protected keywords in OOP languages [175].

Common visibility modifiers in smart contracts typically include:

1) **Public:** Accessible from within and outside the contract.
2) **Private:** Only accessible within the contract.
3) **Internal:** Accessible within the contract and by derived contracts.
4) **External:** Only accessible from outside the contract.

These modifiers allow developers to fine-tune the level of access to different parts of the contract, enhancing security and controlling how the contract interacts with other contracts or external entities. This capability is crucial for creating secure and well-structured smart contracts.

[R6] Smart contracts **SHALL** implement appropriate visibility modifiers for all functions and state variables.

[D15] Access control mechanisms **SHOULD** be implemented to restrict function calls based on roles or conditions.

[D16] Developers **SHOULD** use Visibility and Access Control to control how smart contracts interact with other contracts or external entities.

4.2.2.7 Events

Events in smart contracts are mechanisms for logging and notifying external entities about specific occurrences or state changes within the contract. Events provide a way for smart contracts to communicate with the outside world, triggering notifications that can be picked up by external systems or user interfaces. They thus serve as a crucial bridge between the on-chain logic and off-chain systems or users. Many smart contract languages support the concept of events, similar to the observer pattern in OOP, allowing external entities to react to changes in the contract's state [39].

[R7] Contracts **SHALL** emit events for significant state changes or important actions.

[D17] Smart contract languages **SHOULD** support events.

[D18] Events **SHOULD** be used to provide an audit trail of contract activities.

[D19] Event parameters **SHOULD** be carefully chosen to balance between providing necessary information and maintaining efficiency and memory used.

4.2.3 Benefits of OOP in Smart Contracts

Some of the benefits of using OOP when developing smart contracts are listed below. It is thus recommended, though not mandatory, that smart contracts are developed accordingly [94], [86], [72].

- **Modularity:** Contracts can be designed as self-contained modules, enhancing readability and maintainability.
- **Reusability:** Common patterns and functionalities can be abstracted into base contracts and reused across multiple implementations.
- **Flexibility:** The ability to compose contracts from smaller, specialized contracts allows for flexible and adaptable designs.

[R8] Smart contracts with business/operational roles **SHALL** incorporate both foundational and functional attributes as needed.

[R9] Smart contracts **SHALL** be designed to leverage OOP principles to enhance modularity, reusability, and maintainability.

[R10] The implementation of OOP concepts in smart contracts **SHALL NOT** compromise the security, efficiency, or gas optimization of the contract.

[D20] Developers **SHOULD** use OOP principles to create modular smart contracts that can be easily understood, maintained, and upgraded.

[D21] Smart contracts **SHOULD** be designed with clear separation of concerns, using OOP concepts to organize code logically and enhance readability.

[D22] Inheritance **SHOULD** be used to create hierarchies of contracts with shared functionality, promoting code reuse and reducing redundancy.

[D23] Developers **SHOULD** leverage interfaces and abstract contracts to define common behaviour that can be implemented by multiple contracts, enhancing flexibility and interoperability.

[D24] Encapsulation **SHOULD** be employed to group related data and functions together, improving data integrity and reducing the risk of unintended interactions.

[D25] Smart contracts **SHOULD** use composition to combine simpler contracts into more complex systems, allowing for greater flexibility in contract design.

[D26] Polymorphism **SHOULD** be utilized to create more adaptable and extensible contract systems, particularly when dealing with upgradeable contracts or systems with multiple implementations.

[D27] Developers **SHOULD** implement proper access control mechanisms using OOP principles to restrict function calls and state modifications to authorized parties only.

[D28] Event emission **SHOULD** be used consistently to provide a clear audit trail of contract activities and state changes, leveraging OOP principles for structured event handling.

[D29] Smart contracts **SHOULD** be designed with modularity in mind, allowing for easier testing, auditing, and potential upgrades of specific components.

[D30] Developers **SHOULD** consider creating libraries or base contracts for commonly used functionalities, promoting code reuse across multiple contracts or projects.

[D31] The use of design patterns that leverage OOP concepts (such as factory patterns, proxy patterns, or state machines) **SHOULD** be considered to solve common smart contract design challenges.

[D32] Developers **SHOULD** balance the benefits of OOP with the unique constraints of blockchain environments, such as gas costs and the public nature of all contract data.

[D33] Regular code reviews and audits **SHOULD** be conducted to ensure that OOP principles are applied effectively and do not introduce unexpected vulnerabilities or inefficiencies.

[D34] Smart contracts **SHOULD** be designed with flexibility to adapt to changing business needs, using OOP principles to facilitate easier updates and extensions.

[D35] Documentation **SHOULD** clearly explain the OOP structure of the smart contract system, including class hierarchies, interfaces, and design patterns used.

4.2.4 Considerations for OOP in Distributed Environments

4.2.4.1 Defining the major considerations

While Smart Contracts follow OOP principles, they operate in a distributed environment with unique constraints [71].

[R11] Developers **SHALL** consider the unique characteristics of distributed blockchain environments when applying OOP principles to smart contract design.

[R12] The implementation of OOP concepts **SHALL NOT** compromise the security, efficiency, or decentralized nature of the smart contract system.

[D36] Developers **SHOULD** consider the following factors and develop smart contract accordingly:

- Gas costs (in blockchain systems where such concept exists). (See clause 4.2.4.2 herewith).
- Immutability after deployment. (See clause 4.3.1 herewith).
- Public nature of blockchain data. (See clause 4.2.4.3 herewith).
- Consensus mechanisms of the underlying distributed ledger. (See clause 4.2.4.4 herewith).

[D37] Developers **SHOULD** balance the benefits of OOP design with the constraints and characteristics of distributed blockchain environments.

[D38] Contracts **SHOULD** be designed with modularity and upgradeability in mind, using patterns like proxy contracts to allow for future improvements.

[D39] Developers **SHOULD** thoroughly test smart contracts in environments that simulate the distributed nature of blockchains, including potential network latencies and state inconsistencies.

[D40] The use of formal verification techniques **SHOULD** be considered to prove the correctness of critical contract functions in a distributed setting.

[D41] Developers **SHOULD** stay informed about the latest developments in blockchain technology and adjust their OOP approaches accordingly.

[D42] Regular security audits **SHOULD** be conducted to ensure that the application of OOP principles does not introduce vulnerabilities in the distributed environment.

[D43] Contracts **SHOULD** implement robust error handling and fallback mechanisms to maintain system integrity in the face of unexpected conditions in the distributed network.

4.2.4.2 Gas costs

In blockchain environments like Ethereum, every operation has an associated gas cost. This includes not just data storage, but also computation. OOP features can sometimes lead to more complex computations, which in turn can increase gas costs. While OOP principles can bring many benefits to smart contract development in terms of code organization, reusability, and maintainability, their implementation in a distributed, gas-cost environment requires careful consideration. Developers need to balance the advantages of OOP with the unique constraints and costs associated with blockchain environments, often leading to a hybrid approach that adapts OOP principles to the specific needs of smart contract development [31].

[R13] Smart contracts **SHALL** be designed with gas optimization in mind, balancing OOP principles with efficiency.

[D44] Developers **SHOULD** carefully consider the gas costs of inheritance hierarchies and avoid deep inheritance chains that could lead to expensive contract deployments or function calls.

[D45] Contracts **SHOULD** use libraries for common functions to reduce deployment costs and promote code reuse.

[D46] Developers **SHOULD** optimize data structures and algorithms to minimize gas consumption, even if it means deviating from traditional OOP patterns.

[D47] The following aspects **SHOULD** be considered:

1) **Inheritance**

   Pros: Inheritance can reduce code duplication, potentially leading to smaller contract sizes and lower deployment costs.

   Cons: Deep inheritance hierarchies can increase the complexity of function calls, potentially leading to higher gas costs for each transaction.

2) **Polymorphism**

   Pros: Can lead to more flexible and modular code, potentially reducing the need for multiple similar contracts.

   Cons: Dynamic dispatch (resolving which function to call at runtime) can increase gas costs compared to direct

3) **Encapsulation**

   Pros: Can help in organizing code and data, potentially leading to more efficient storage layouts.

   Cons: Strict encapsulation might require additional function calls to access data, increasing gas costs.

4) **Composition**

   Pros: Can lead to more modular and reusable code, potentially reducing overall contract size.

   Cons: Interactions between multiple contracts (if composition is implemented across contracts) can increase gas costs due to additional contract calls.

5) **State Variables**: The use of numerous state variables, common in OOP, can lead to higher storage costs in a blockchain environment.

6) **Function Calls**: In OOP, it is common to have many small, specialized functions. In a gas-cost environment, this can lead to higher costs due to the overhead of multiple function calls.

7) **Optimization Challenges**: Some OOP patterns that are efficient in traditional environments may not be gas-efficient in a blockchain context, requiring developers to balance OOP principles with gas optimization.

8) **Contract Size**: Complex OOP structures can lead to larger contract sizes, which have higher deployment costs and may hit contract size limits on some platforms.

9) **Readability vs. Efficiency**: While OOP can improve code readability and maintainability, these benefits should be balanced against the need for gas efficiency in smart contracts.

10) **Testing and Verification**: OOP structures can make formal verification of smart contracts more complex, which is crucial for ensuring contract security.

4.2.4.3 Public nature of blockchain data

While OOP principles can still be valuable in smart contract development, the public nature of blockchain data fundamentally changes how these principles are applied. Developers should constantly balance the benefits of OOP design with the unique transparency and security requirements of blockchain environments. This often leads to new patterns and practices that adapt OOP concepts to the realities of public, immutable, and transparent distributed ledgers [108].

[R14] Developers **SHALL** be aware that all contract data is publicly visible, even if marked as "private" in the contract code.

[D48] Sensitive information **SHOULD NOT** be stored directly on the blockchain, even in private variables.

[D49] Contracts **SHOULD** use cryptographic techniques (e.g. hashing, encryption) when dealing with sensitive data that has to be referenced on-chain.

[D50] Developers **SHOULD** design contracts with the understanding that encapsulation does not provide data hiding in the traditional OOP sense on a blockchain.

[R15] Developers **SHALL** be extra cautious about what data is stored, even in "private" variables.

[R16] Interfaces **SHALL** be designed with consideration for security and potential misuse.

[D51] The following aspects **SHOULD** be considered:

1) **Data Visibility**: In traditional OOP, private data members are truly private. However, in blockchain environments, all data is publicly visible, even if it is marked as "private" in the contract code. This fundamental difference means that OOP encapsulation does not provide data hiding in the traditional sense on a blockchain.

2) **Encapsulation Redefined**: While data cannot be hidden, encapsulation in smart contracts is more about access control than information hiding. It defines who can modify the data, not who can see it.

3) **Sensitive Information**: OOP often encourages grouping related data, but this might lead to storing sensitive information on-chain if not carefully managed.

4) **State Transparency**: The entire state of an object (smart contract) is publicly visible and auditable. This transparency can be beneficial for auditing and verification but challenges traditional OOP notions of information hiding.

5) **Privacy Patterns**: New patterns emerge to handle privacy concerns, such as storing only hashes on-chain and keeping the actual data off-chain. These patterns often do not align with traditional OOP practices and require a different approach to data management.

6) **Public Interfaces**: In OOP, interfaces are usually seen as contracts between objects. In blockchain, they become contracts between the smart contract and the entire network. This public nature means interface design requires extra consideration for security and potential misuse.

7) **Inheritance and Public Scrutiny**: Inherited functions and state variables are all publicly visible, which means vulnerabilities in parent contracts are exposed to public scrutiny. This visibility can aid in security audits but also exposes potential attack vectors.

8) **Composition and Contract Interactions**: When using composition (contracts interacting with other contracts), all these interactions are public. This transparency can be beneficial for auditing but requires careful design to prevent front-running or other exploitation.

9) **Versioning and Upgrades**: In OOP, objects can be easily updated. In blockchain, contract upgrades are complex and all versions remain publicly visible. This persistence of old versions challenges traditional OOP upgrade patterns.

10) **Data Structuring**: OOP encourages grouping related data, but in a public blockchain, this might inadvertently reveal relationships that were meant to be obfuscated.

11) **Event Logging**: While events in OOP are typically internal, blockchain events are public logs that anyone can subscribe to and analyse. This changes how events are used and what information they should contain.

12) **Testing and Verification**: The public nature of the code and data means that testing and formal verification become even more critical. Every aspect of the contract, including inherited code and composed contracts, is open to public scrutiny and potential exploitation.

13) **Documentation Practices**: With all code being public, inline comments and documentation practices become part of the public interface of the contract. This requires a shift in how developers approach code documentation.

4.2.4.4 Consensus mechanisms of the underlying distributed ledger

The application of Object-Oriented Programming (OOP) principles in a distributed blockchain environment interacts in complex ways with the consensus mechanisms of the underlying distributed ledger. While OOP principles can provide valuable structure and organization to smart contract code, their implementation has to be carefully adapted to the realities of distributed consensus. Developers need to consider not just the logical structure of their code, but how that structure interacts with the consensus mechanism to ensure reliable, deterministic execution across a distributed network. This often leads to a hybrid approach that respects OOP principles while adapting them to the unique constraints of distributed ledger environments [156].

[R17] Smart contracts **SHALL** be designed to operate deterministically to ensure consistent execution across all nodes in the network.

[D52] Developers **SHOULD** avoid using non-deterministic operations or external calls that could lead to inconsistent contract states across nodes.

[D53] Contracts **SHOULD** be designed to handle potential state reversions due to blockchain reorganizations or consensus failures.

[D54] Developers **SHOULD** consider the impact of the consensus mechanism on contract execution time and design contracts accordingly.

[D55] The following points **SHOULD** be considered:

1) **State Changes and Consensus**: In OOP, object state changes are typically immediate and localized. In a distributed ledger, every state change (object modification) has to go through consensus. This means that what appears to be a simple operation in OOP terms (like changing an object's property) becomes a network-wide agreement process.

2) **Transactional Nature of Operations**: OOP methods that modify state has to be designed with the understanding that they are essentially database transactions that need to be agreed upon by the network. This can lead to race conditions and concurrency issues not typically considered in traditional OOP.

3) **Determinism Requirements**: Consensus mechanisms require that contract execution be deterministic - given the same input, all nodes have to arrive at the same state. This constrains the use of certain OOP features like random number generation or time-dependent operations, which should be carefully implemented to maintain determinism.

4) **Execution Order**: In OOP, method execution order is typically controlled by the programmer. In a distributed ledger, the order of transaction execution can be influenced by network factors and miner/validator decisions. This can affect how OOP principles, like inheritance and polymorphism behave in practice.

5) **Atomicity of Operations**: While OOP often assumes that method calls are atomic, in a distributed ledger, operations can be interrupted by consensus failures or network issues. This requires careful design of contract methods to handle partial execution scenarios.

6) **State Rollbacks**: Consensus mechanisms may require state rollbacks if a transaction is ultimately rejected. OOP designs need to account for the possibility that any state change might be reverted, which is not a common consideration in traditional OOP.

7) **Gas Costs and Consensus**: Different consensus mechanisms may have different cost models (like gas in Ethereum). OOP designs need to be optimized not just for logical efficiency, but for efficiency within the specific consensus cost model.

8) **Finality Considerations**: Different consensus mechanisms offer different levels of finality (how quickly a transaction becomes irreversible). OOP designs need to account for the possibility of state changes being considered "probable" rather than "definite" for some period.

9) **Sharding and OOP**: Some consensus mechanisms use sharding for scalability. This can complicate OOP designs, especially when objects (contracts) need to interact across shards. Sharding is defined in the note below.

10) **Smart Contract Upgrades**: Consensus mechanisms often make upgrading contracts complex. This challenges traditional OOP notions of class evolution and versioning.

11) **Event Propagation**: In OOP, events are often used for immediate responses. In a distributed ledger, event propagation is subject to consensus timing, which can introduce delays.

12) **Inheritance in a Distributed Context**: Inherited functions in smart contracts have to be compatible with the consensus mechanism, potentially limiting the flexibility of inheritance hierarchies.

13) **Polymorphism and Network Agreement**: Dynamic dispatch in polymorphism has to be implemented in a way that all nodes can agree on which method is being called, potentially limiting its use.

14) **Encapsulation and Network Validation**: While encapsulation in OOP is about information hiding, in a distributed ledger it is more about access control that the entire network can validate and agree upon.

15) **Composition and Cross-Contract Calls**: In a distributed environment, composition often involves cross-contract calls, which are subject to consensus rules and can have different execution guarantees than intra-contract method calls.

4.3 Properties of Smart Contracts

4.3.1 Immutability

4.3.1.1 Definition

Immutability in the context of smart contracts refers to the property that once a smart contract is deployed on a Permissioned Distributed Ledger (PDL), its code cannot be altered or changed.

Immutability is a fundamental property of smart contracts that provides trust and security but also requires careful planning and design to manage its limitations and challenges [108].

This means:

1) **Code Permanence:** The executable code of the smart contract, once accepted through consensus and added to the ledger, remains fixed and unchangeable.

2) **Persistent State:** While the internal state (data) of the contract can change according to its predefined rules, the logic governing these changes (i.e. the code itself) remains constant.

3) **Irreversible Deployment:** After deployment, the contract's bytecode becomes a permanent part of the ledger's history.

[R18] Smart contracts **SHALL** adhere to the principle of immutability once deployed on the blockchain.

[R19] Developers **SHALL** understand and respect the immutable nature of smart contract code after deployment.

[R20] The contract's bytecode **SHALL** become a permanent part of the ledger's history upon deployment.

[D56] Smart contracts **SHOULD** include clear version identifiers to manage different iterations of contract logic.

[D57] Developers **SHOULD** maintain comprehensive documentation of the contract's design, including any upgrade mechanisms or parameterization.

[D58] Contracts **SHOULD** implement a time-delay mechanism for critical operations or upgrades to allow for community review and potential reversion.

[D59] Developers **SHOULD** consider using formal verification techniques to prove the correctness of immutable contract logic.

[D60] Smart contracts **SHOULD** include mechanisms for data migration or state transfer when upgrades are necessary.

[D61] Developers **SHOULD** educate users and stakeholders about the immutable nature of the contract and any upgrade mechanisms in place.

[D62] Contracts **SHOULD** implement event emission for any operations that modify upgradeable components or parameters.

[D63] Developers **SHOULD** consider implementing a simulation or "dry run" functionality for complex operations to allow users to verify outcomes before executing irreversible transactions.

4.3.1.2 Implications

The immutable nature of Smart Contracts has several important implications [59]:

1) **Trust and Transparency:** Participants can trust that the rules of the contract will not change unexpectedly, as they can verify the immutable code at any time.

2) **Auditability:** The unchanging nature of the code allows for thorough auditing and long-term analysis of contract behaviour.

3) **Predictability:** Users can rely on consistent contract behaviour over time, which is crucial for long-term planning and interaction.

4) **Security:** Immutability prevents malicious alterations of the contract code after deployment, enhancing overall security.

5) **Challenges in Upgradeability:** It complicates the process of fixing bugs or updating functionality, as the original contract cannot be directly modified.

[R21] Smart contracts **SHALL** be designed with the understanding that their core logic cannot be altered after deployment.

[D64] Developers **SHOULD** leverage immutability to build trust and predictability into their smart contract systems.

[D65] Contracts **SHOULD** emit events for significant state changes to provide an immutable audit trail.

4.3.1.3 Challenges

While immutability is a key feature, it also presents challenges [177]:

1) **Error Permanence:** Bugs or oversights in the contract become permanent, potentially leading to vulnerabilities or unintended behaviours.

2) **Adaptability Issues:** Changing business requirements or regulations can be difficult to accommodate in immutable contracts.

3) **Gas and Storage Considerations:** Immutable contracts contribute to the ever-growing size of the ledger, which has implications for storage and processing requirements.

[R22] Developers **SHALL** plan for potential bugs or oversights, as they cannot be directly fixed in deployed contracts.

[D66] Smart contracts **SHOULD** include mechanisms to pause or disable critical functions in case of discovered vulnerabilities.

[D67] Developers **SHOULD** implement thorough testing and auditing processes before deployment to minimize the risk of errors in immutable code.

4.3.1.4 Solutions

To address these challenges, developers have created patterns that allow for some degree of upgradeability while maintaining the integrity of the original contract [112], [160]:

1) **Proxy Patterns:** Using upgradeable proxy contracts that point to the latest version of the logic contract. This method allows installing new versions of smart contracts using a proxy without changing the original proxy code.

2) **Data Separation:** Storing mutable data separately from immutable code.

3) **Parameterization:** Designing contracts with configurable parameters that can be adjusted without changing the core code.

4) **Modular Design:** Creating systems of interconnected contracts that can be individually replaced or upgraded.

5) **Thorough Testing and Auditing:** Implementing rigorous testing procedures and external audits before deployment to minimize the risk of errors.

[D68] Smart contracts **SHOULD** implement upgradeability patterns (e.g. proxy patterns) when future modifications might be necessary.

[D69] Developers **SHOULD** use parameterization to allow for adjustments in contract behaviour without changing the core code.

[D70] Contracts **SHOULD** be designed with modularity in mind, allowing for the replacement of specific components without affecting the entire system.

[D71] Developers **SHOULD** implement governance mechanisms for managing upgrades or parameter changes in a decentralized manner.

Examples of such solutions are detailed in clause A.1.1.1 herewith.

Additional methods exist, and a detailed discussion of the same is beyond the scope of the present document.

4.3.2 Transparency

4.3.2.1 Definition

Transparency in smart contracts refers to the property that allows all authorized participants to inspect and verify the contract's code, current state, and execution history. This property is inherent to the distributed nature of the underlying ledger technology.

Transparency is a powerful feature of smart contracts that enhances trust and accountability. However, it should be carefully balanced with privacy considerations to create robust and appropriate solutions for different use cases.

[R23] Smart contracts **SHALL** be designed to provide transparency in their code, execution, and state changes.

[R24] Transparency mechanisms **SHALL NOT** compromise the security or privacy of sensitive information.

[R25] Smart contract code and all associated libraries **SHALL** be available for auditing purposes.

[D72] Developers **SHOULD** use zero-knowledge proofs where possible to verify computations without revealing underlying data.

[D73] Smart contracts **SHOULD** implement role-based access control to limit visibility of sensitive data to authorized parties only.

[D74] Contracts **SHOULD** use off-chain storage solutions for sensitive data, with only hashes or references stored on-chain.

[D75] Developers **SHOULD** consider implementing encrypted data fields for storing sensitive information on-chain.

[D76] Smart contracts **SHOULD** emit events for significant state changes, but these events SHOULD NOT contain sensitive data.

[D77] When dealing with financial transactions, contracts **SHOULD** use techniques like confidential transactions to hide transaction amounts while proving their validity.

[D78] Developers **SHOULD** consider using secure multi-party computation techniques for collaborative operations on private data.

[D79] Smart contracts **SHOULD** implement time-locked encryption for data that needs to be kept private for a certain period.

[D80] Where applicable, contracts **SHOULD** use homomorphic encryption to perform computations on encrypted data.

[D81] Developers **SHOULD** regularly audit their smart contracts to ensure that privacy measures do not inadvertently create security vulnerabilities.

[D82] Contracts **SHOULD** provide a clear and accessible interface for users to understand what information is public and what is protected.

[D83] Developers **SHOULD** implement a tiered transparency model, where different levels of information are accessible to different stakeholders based on their roles and permissions.

4.3.2.2 Key Aspects

Achieving transparency requires several factors to be considered [169]:

1) **Code Visibility:** The code of smart contracts is typically visible to all authorized participants, allowing for auditing and verification of the contract's behaviour.

2) **Transaction Traceability:** All interactions with a smart contract are recorded on the PDL, creating a transparent and auditable trail of contract execution.

3) **State Visibility:** The current state of the contract, including its variables and data, is accessible to authorized parties.

[R26] Smart contract developers **SHALL** implement Transaction Traceability and State Visibility.

[D84] Smart contract developers **SHOULD** implement Code Visibility, allowing for public verification of contract logic.

[D85] Contracts **SHOULD** emit events for all significant state changes and important function calls to enhance traceability.

4.3.2.3 Benefits

Transparency offers several advantages in smart contract systems including [148]:

1) **Trust Enhancement:** Participants can verify the contract's logic and execution, fostering trust in the system.

2) **Auditability:** Enables thorough auditing of contract behaviour and transaction history.

3) **Bug Detection:** Allows for community-driven identification of potential bugs or vulnerabilities.

4) **Accountability:** Creates a clear record of all interactions, promoting accountability among participants.

[D86] Developers **SHOULD** leverage transparency to build trust among users and stakeholders.

[D87] Contracts **SHOULD** provide clear mechanisms for users to verify the current state and historical transactions.

4.3.2.4 Challenges

While transparency is generally beneficial, it also presents certain challenges [156]:

1) **Privacy Concerns:** In some cases, complete transparency may not be desirable, especially when dealing with sensitive business logic or personal data.

2) **Competitive Advantage:** Visible code may reveal proprietary algorithms or business strategies.

3) **Front-Running:** In some scenarios, transparent pending transactions could be exploited by malicious actors.

[D88] Smart contract developers **SHOULD** write code that addresses privacy concerns, does not reveal proprietary algorithms and can not be exploited by malicious users.

[D89] Developers **SHOULD** implement mechanisms to prevent front-running attacks in transparent systems.

4.3.2.5 Balancing Transparency and Privacy

To address these challenges, PDL systems often provide mechanisms to balance transparency with privacy needs [83], [15]:

1) **Zero-Knowledge Proofs:** Allow verification of computations without revealing the underlying data.
2) **Private Data Collections:** Enable storing sensitive data off-chain while maintaining a hash of the data on-chain.
3) **Access Control:** Implement granular access controls to limit visibility to authorized parties only (which, in a sense, is the essence of PDL compared to permissionless systems).
4) **Encryption:** Use encryption techniques to protect sensitive data while maintaining overall system transparency.

[R27] Smart contracts **SHALL** implement mechanisms to protect sensitive data while maintaining overall system transparency.

[R28] Any privacy-preserving technique used in smart contracts **SHALL NOT** compromise the integrity or auditability of the blockchain.

[R29] Smart contracts handling personal data **SHALL** comply with relevant data protection regulations (e.g. GDPR).

4.3.2.6 Considerations for Implementation

When designing smart contract systems, developers should consider [93], [13]:

1) **Regulatory Compliance:** Ensure that transparency features align with relevant data protection regulations.
2) **User Education:** Inform users about what information will be visible and to whom.
3) **Transparency Levels:** Design systems with appropriate levels of transparency for different types of data and operations.
4) **Audit Trails:** Implement comprehensive logging to maintain transparency of system operations.

[R30] Smart contracts **SHALL** implement logging mechanisms to record all significant state changes and important operations.

[R31] Access control mechanisms **SHALL** be implemented to ensure that only authorized parties can view or modify sensitive data.

[R32] Smart contracts **SHALL** include functions to allow authorized parties to verify the contract's current state.

[R33] Any external data sources or oracles used by the smart contract **SHALL** be clearly documented and their roles in the contract's operation **SHALL** be transparent.

[D90] Developers **SHOULD** use standardized patterns and libraries (e.g. OpenZeppelin) to implement common functionalities, enhancing readability and reducing the risk of errors.

[D91] Smart contracts **SHOULD** implement a tiered transparency model, where different levels of information are accessible to different stakeholders.

[D92] Contracts **SHOULD** include mechanisms for gradual disclosure of information, such as commit-reveal schemes, where appropriate.

[D93] Developers **SHOULD** consider implementing upgradeable contract patterns to allow for future improvements while maintaining transparency of changes.

[D94] Smart contracts **SHOULD** emit events for all significant actions, providing a clear audit trail.

[D95] Where possible, contracts **SHOULD** use zero-knowledge proofs to prove the correctness of computations without revealing the underlying data.

[D96] Developers **SHOULD** implement clear error messages that provide enough information for debugging without exposing sensitive data.

[D97] Smart contracts **SHOULD** include functions that allow users to query their own data or permissions without accessing others' information.

[D98] Where applicable, contracts **SHOULD** implement time-locked transparency, where certain information becomes public after a predetermined period.

[D99] Developers **SHOULD** consider implementing a transparency dashboard or interface that provides an easy-to-understand overview of the contract's current state and history.

[D100] Smart contracts **SHOULD** include mechanisms for authorized parties to flag or challenge potentially incorrect or fraudulent data.

[D101] Developers **SHOULD** implement rate limiting on data queries to prevent potential denial-of-service attacks or data scraping.

[D102] Where privacy is a concern, contracts **SHOULD** use techniques like ring signatures or stealth addresses to obscure transaction participants while maintaining verifiability.

[D103] Developers **SHOULD** carefully balance the level of detail in publicly emitted events to provide necessary transparency without compromising privacy or security.

[D104] Smart contracts **SHOULD** implement versioning mechanisms to track and communicate changes in contract logic or data structures over time.

[D105] Developers **SHOULD** consider implementing a simulation mode that allows users to preview the effects of their actions before committing them to the blockchain.

4.3.3 Determinism

4.3.3.1 Definition

Determinism in smart contracts refers to the property that ensures identical inputs and state will always produce the same output, regardless of when or where the contract is executed. Determinism is a fundamental property of smart contracts that ensures reliability and consistency in distributed systems. It plays a vital role in maintaining the integrity of the blockchain and the trust of its users. Developers should prioritize deterministic behaviour in their smart contract design and implementation to ensure robust and reliable decentralized applications [156], [169], [120], [173].

4.3.3.2 Key Aspects of Determinism

The key aspects of determinism in smart contracts are:

1) **Consistent Execution:** Given the same input and state, a smart contract will always produce the same output across all nodes.

2) **Predictability:** Users and developers can reliably predict the outcome of contract execution based on inputs.

3) **Network Consensus:** Determinism enables nodes to agree on the state of the blockchain without central coordination.

4.3.3.3 Importance of Determinism

Determinism is important for the following reasons:

1) **Consensus:** Determinism is crucial for achieving consensus in distributed systems. All nodes arrive at the same state after executing a transaction.

2) **Reliability:** Users can trust that contract outcomes are consistent and not subject to external variables.

3) **Auditability:** Deterministic behaviour allows for easier verification and auditing of contract executions.

4) **Security:** Predictable execution helps prevent certain types of attacks and vulnerabilities.

4.3.3.4 Challenges associated with Determinism

Determinism poses several challenges when developing smart contracts:

1) **External Data:** Integrating external data (e.g. through oracles) while maintaining determinism can be challenging.

2) **Time-Dependent Operations:** Operations that depend on time or randomness require careful implementation to remain deterministic.

3) **Floating-Point Arithmetic:** Non-deterministic behaviour in floating-point operations across different hardware/OS can lead to inconsistencies.

4) **Randomness:** Generating random numbers in a deterministic environment.

4.3.3.5 Determinism Implementation Considerations

When developing smart contracts the following details related to determinism should be considered:

1) **Avoidance of Non-Deterministic Operations:** Steering clear of functions that may produce different results (e.g. random number generation, system time).

2) **Standardized Data Formats:** Using standardized data formats and encodings to ensure consistent interpretation across all nodes.

3) **Versioning:** Clearly defining and managing contract versions to ensure all nodes are executing the same code.

4) **Careful Oracle Design:** When using external data sources, mechanisms to ensure consistent data across all nodes should be implemented.

5) **Fixed-Point Arithmetic:** Fixed-point math should be preferred over floating-point to avoid precision and rounding issues.

6) **Testing:** Rigorous testing across different environments is essential to verify deterministic behaviour.

7) **Gas Considerations:** Gas costs can affect determinism if not properly managed.

4.3.3.6 Balancing Determinism and Functionality

While determinism is crucial, it can limit certain functionalities. Smart contract designers should carefully balance the need for determinism with the desired features of their application. In some cases, hybrid approaches using off-chain computation with on-chain verification can provide a solution.

4.3.3.7 Requirements and Recommendations

[R34] Smart contracts **SHALL** produce the same output for a given input and state, regardless of when or where they are executed.

[R35] The execution environment for smart contracts **SHALL** ensure consistent behaviour across all nodes in the network.

[R36] Smart contracts **SHALL NOT** rely on non-deterministic functions or external data sources that could lead to inconsistent results.

[R37] Any use of time-dependent operations in smart contracts **SHALL** be based on block numbers or timestamps provided by the blockchain, not system time.

[D106] Developers **SHOULD** avoid using floating-point arithmetic in smart contracts due to potential inconsistencies across different hardware.

[D107] Smart contracts **SHOULD** use deterministic random number generation techniques when randomness is required.

[D108] Developers **SHOULD** implement comprehensive unit tests to verify deterministic behaviour under various conditions.

[D109] When interacting with external systems, smart contracts **SHOULD** use a commit-reveal scheme or similar mechanism to ensure determinism.

[D110] Smart contracts **SHOULD** include explicit error handling for all possible execution paths to ensure consistent behaviour even in error scenarios.

[D111] Developers **SHOULD** use formal verification techniques where possible to mathematically prove the deterministic behaviour of critical contract functions.

[D112] When implementing complex algorithms, contracts **SHOULD** break them down into smaller, deterministic steps that can be easily verified.

[D113] Smart contracts **SHOULD** use standardized data formats and encodings to ensure consistent interpretation of data across all nodes.

[D114] Developers **SHOULD** carefully manage contract versioning to ensure all nodes are executing the same code.

[D115] When using oracles or external data sources, contracts **SHOULD** implement a consensus mechanism among multiple sources to enhance determinism.

[O7] In order to ensure deterministic behaviour developers **MAY** consider the following:

1) Instead of using the current timestamp directly, use block numbers as a proxy for time.
2) For randomness, use verifiable random functions or commit-reveal schemes.
3) When integrating external data, use a decentralized oracle network with a consensus mechanism.

4.3.4 Atomicity

4.3.4.1 Definition

Atomicity in smart contracts refers to the property that ensures a transaction or operation is treated as a single, indivisible unit. This means that the execution of a smart contract function either completes entirely or not at all, with no intermediate states.

Atomicity is a critical property of smart contracts that contributes to their reliability and predictability. It simplifies reasoning about contract behaviour and helps prevent many classes of errors and vulnerabilities. Developers should strive to maintain atomicity in their smart contract designs to ensure robust and trustworthy decentralized applications [156], [177], [173], [128].

4.3.4.2 Key Aspects of Atomicity

Atomicity in smart contracts ensures that a transaction is treated as a single, indivisible unit of work. This means that all operations within a transaction have to either complete successfully or have no effect at all. Understanding the key aspects of atomicity is crucial for developing robust and reliable smart contracts that can maintain data consistency even in the face of failures or interruptions.

1) **All-or-Nothing Execution:**
   - In smart contracts, a transaction either completes fully or is entirely reverted.

- No partial changes are left on the blockchain.

EXAMPLE: A token swap in a decentralized exchange either completes with both parties receiving their respective tokens, or the entire transaction is reverted.

2) **State Consistency:**

- The contract's state remains consistent before and after the transaction.
- There is no possibility of an in-between or inconsistent state.
- Critical for maintaining the integrity of complex financial operations or multi-step processes within a single transaction.

3) **Error Handling:**

- If any part of the transaction fails, the entire operation is rolled back to its initial state.
- Utilizes blockchain's built-in mechanisms like Ethereum's revert function to ensure atomicity (see clause 4.3.4.4 below).
- Prevents partial execution that could lead to fund loss or inconsistent contract states.

4) **Transactional Boundaries:**

- Clearly defines the scope of atomic operations within smart contract functions.
- Ensures that related state changes are grouped together and executed as a single unit.

4.3.4.3 Importance of Atomicity

Understanding the importance of atomicity is crucial for developers and stakeholders to build robust, reliable, and secure smart contract systems that can maintain data consistency even in the face of failures or interruptions.

1) **Data Integrity:**

- Prevents inconsistent or partial updates to the contract's state.
- Crucial for financial applications where partial transactions could lead to loss of funds or incorrect balances.
- Maintains the reliability of contract data, which is essential for building trust in decentralized systems.

2) **Predictability:**

- Users can rely on transactions being fully completed or fully reverted.
- Simplifies error handling and recovery processes in DApp development.
- Enhances user experience by providing clear transaction outcomes.

3) **Security:**

- Atomic operations help prevent vulnerabilities that could arise from partially completed transactions.
- Mitigates risks associated with race conditions and timing attacks.
- Crucial for preventing exploitation in high-value transactions or complex DeFi protocols.

4) **Consistency in Multi-Step Operations:**

- Ensures that complex operations involving multiple state changes maintain overall consistency.
- Critical for DeFi applications like atomic swaps, lending protocols, or governance systems.

4.3.4.4 Atomicity Implementation Mechanisms

This clause explores the various mechanisms and techniques used to achieve atomicity in smart contract execution.

1) **Transaction Rollback:**

- If any part of a transaction fails, all changes are automatically reverted.
- Utilizes blockchain's native mechanisms.

EXAMPLE 1: Ethereum's EVM rollback feature.

- Ensures no trace of the failed transaction remains in the contract's state.

2) **Exception Handling:**

- Proper use of try-catch mechanisms and error handling to manage failures.

EXAMPLE 2: In Solidity, using require(), assert(), and revert() statements to enforce conditions and handle errors.

- Custom error messages can be used to provide more context about the failure.

3) **Gas Limits:**

- In Ethereum and similar systems, gas limits ensure that a transaction either completes within the allocated resources or is entirely reverted.
- Prevents partial execution due to out-of-gas errors, maintaining atomicity.
- Developers have to carefully estimate gas costs to ensure complex operations can complete within block gas limits.

4) **Check-Effects-Interactions Pattern:**

- Implementing a specific order of operations: checks, then effects on the contract's state, and finally external interactions.
- Helps maintain atomicity by reducing the risk of reentrancy and other related vulnerabilities.

4.3.4.5 Challenges of Atomicity

Implementing atomicity in smart contracts presents several unique challenges that developers should carefully navigate. These challenges arise from the distributed nature of blockchain systems, the limitations of smart contract platforms, and the complexities of ensuring all-or-nothing transaction execution.

1) **Complex Operations:**

- Ensuring atomicity in complex, multi-step operations can be challenging and requires careful design.

EXAMPLE: In a complex DeFi protocol, ensuring that all steps of a leveraged position creation are atomic.

[O8] Solutions **MAY** involve breaking down complex operations into smaller, manageable atomic units.

2) **External Calls:**

- Interactions with external contracts or services may introduce risks to atomicity if not properly managed.
- The calling contract may not have control over the execution of external contracts.

[O9] Techniques like the pull payment pattern **CAN** be used to maintain atomicity in cross-contract interactions.

3) **Performance Considerations:**

- Atomic operations may sometimes impact performance, especially in complex scenarios.

- Large atomic operations might consume significant gas, potentially hitting block gas limits.
- Balancing atomicity with gas efficiency is crucial for contract optimization.

4) **Cross-Chain Atomicity:**

- Achieving atomicity across different blockchain networks presents significant challenges.
- Requires advanced techniques like Hash Time Locked Contracts (HTLCs) or relay networks.

4.3.4.6 Atomicity Best Practices

Implementing atomicity in smart contracts is crucial for ensuring the integrity and consistency of transactions, but it requires careful consideration and adherence to best practices. This clause outlines key strategies and techniques for effectively implementing atomicity in smart contract design. By following these best practices, developers can create more robust and reliable smart contracts that maintain data consistency and transaction integrity, even in complex blockchain environments.

1) **Single-Function Atomic Operations:**

[D116] Developers **SHOULD:**

- design functions to perform atomic operations whenever possible.
- Encapsulate related state changes within a single function to leverage built-in transaction atomicity.

EXAMPLE: A function that updates user balances and emits events should be contained in one atomic operation.

2) **Checks-Effects-Interactions Pattern:**

- Crucial for functions that interact with other contracts or transfer tokens.

[D117] Developers **SHOULD:**

- Implement this pattern to minimize the risk of reentrancy attacks while maintaining atomicity.
- Perform all checks at the beginning of the function, followed by state changes, and external calls last.

3) **Proper State Management:**

[D118] Developers **SHOULD:**

- Ensure that all state changes within a function are consistent and atomic.
- Use temporary variables or structures to manage complex state changes before committing them.
- Consider using a state machine pattern for complex, multi-step processes to ensure atomicity at each stage.

4) **Thorough Testing:**

[D119] Developers **SHOULD:**

- Implement comprehensive testing, including failure scenarios, to verify atomic behaviour.
- Use tools like Hardhat or Truffle for writing and running extensive test suites.
- Simulate various error conditions to ensure proper rollback and state consistency.

5) **Gas Estimation and Management:**

[D120] Developers **SHOULD:**

- Carefully estimate gas costs for atomic operations to prevent out-of-gas errors.

- Consider breaking very large operations into smaller, manageable atomic transactions if necessary.
- Use gas-efficient coding practices to maximize the complexity of atomic operations within gas limits.

6) **Event Emission:**

[D121] Developers **SHOULD:**

- Develop smart contracts that emit events at the end of atomic operations to provide transparency and facilitate off-chain tracking.
- Ensure that events are only emitted after all state changes have been successfully made.

4.3.4.7 Atomicity in Multi-Contract Interactions

When dealing with interactions across multiple contracts, maintaining atomicity becomes more complex. Several techniques are listed below:

1) **Two-Phase Commit Protocols:**

- Useful for coordinating atomic operations that span multiple contracts.

[D122] Developers **SHOULD** Implement a preparation phase and a commit phase across contracts.

EXAMPLE: In a complex DeFi operation involving multiple protocols, use a coordinator contract to manage the two-phase commit.

2) **Intermediate States with Final Confirmation:**

- Provides a way to handle atomicity across asynchronous operations or multiple transactions.

[D123] Developers **SHOULD** use intermediate states to track the progress of multi-contract operations.

[D124] Developers S**HOULD** implement a final confirmation step to ensure all parts of the operation were successful before considering it complete.

3) **Proxy Contracts and Upgradability:**

- Allows for upgrading interaction logic while maintaining a consistent interface for atomicity.

[D125] Developers **SHOULD** utilize proxy patterns to encapsulate complex multi-contract interactions.

4) **Batched Transactions:**

- Increases gas efficiency and ensures atomicity across related operations.

[D126] Developers **SHOULD** use contract methods that can handle multiple operations in a single transaction.

5) **Failure Handling and Rollback Mechanisms:**

[D127] Developers **SHOULD** implement robust failure handling for multi-contract interactions.

[D128] Developers **SHOULD** design rollback mechanisms that can revert changes across multiple contracts if any part of the operation fails.

4.3.4.8 Requirements and Recommendations

[R38] Smart contract transactions **SHALL** be executed entirely or not at all, with no partial execution states.

[R39] The contract state **SHALL** remain consistent before and after each transaction, with no possibility of an intermediate state.

[R40] If any part of a transaction fails, the entire operation **SHALL** be rolled back to its initial state.

[R41] Smart contracts **SHALL** implement proper error handling to ensure atomicity in case of exceptions or failures.

[D129] Developers **SHOULD** use the checks-effects-interactions pattern to maintain atomicity and prevent reentrancy attacks.

[D130] Smart contracts **SHOULD** implement a transactional model for complex operations involving multiple state changes.

[D131] When interacting with external contracts, developers **SHOULD** use a two-phase commit protocol to ensure atomicity across contract boundaries.

[D132] Contracts **SHOULD** include explicit checks at the beginning of functions to validate preconditions before making any state changes.

[D133] Developers **SHOULD** use modifiers to encapsulate common check-and-revert patterns, enhancing code reusability and atomicity.

[D134] For operations involving multiple participants, contracts **SHOULD** implement an all-or-nothing execution model, such as atomic swaps.

[D135] Smart contracts **SHOULD** emit events at the end of successful atomic operations to provide clear audit trails.

[D136] Developers **SHOULD** use formal verification techniques to prove the atomicity of critical contract functions.

[D137] When implementing upgradeable contracts, the upgrade process **SHOULD** be atomic to prevent inconsistent contract states.

[D138] For operations that cannot be made atomic within a single transaction, contracts **SHOULD** implement a state machine pattern with clearly defined and verifiable state transitions.

[D139] Developers **SHOULD** thoroughly test contract behaviour under various failure scenarios to ensure proper rollback and state consistency.

[D140] When dealing with large datasets, contracts **SHOULD** use techniques like Merkle trees to maintain atomicity while minimizing on-chain storage.

4.3.5 Autonomy

4.3.5.1 Definition

Autonomy in smart contracts refers to their ability to self-execute and self-enforce without the need for intermediaries or external control once deployed. This property enables smart contracts to operate independently based on their predefined rules and conditions. Autonomy is a fundamental characteristic that sets smart contracts apart from traditional agreements and software. It enables trustless, efficient, and consistent execution of complex agreements and processes. However, it also requires careful design and consideration to balance the benefits of autonomous operation with the need for safety, upgradability, and handling of unforeseen circumstances. When implemented effectively, autonomous smart contracts can significantly enhance the efficiency and reliability of various systems and processes across multiple industries [148], [169], [95].

4.3.5.2 Key Aspects of Autonomy

This clause explores the key aspects of autonomy in smart contracts, highlighting how this feature contributes to their efficiency, reliability, and trustworthiness in decentralized systems. Understanding these aspects is crucial for developers and stakeholders to fully leverage the potential of autonomous smart contracts while being aware of their limitations and implications.

1) **Self-Execution:**

   - Smart contracts automatically execute when predefined conditions are met.
   - No manual intervention is required, reducing human error and delays.

EXAMPLE: A decentralized SLA contract automatically pays out penalty when specific criteria are met.

2) Independence:

- Once deployed, smart contracts operate independently of their creators or any central authority.
- The contract's behaviour is determined solely by its code and the blockchain's state.
- This independence ensures impartiality and resistance to tampering.

3) **Rule-Based Behaviour:**

- Contract actions are determined by coded rules, not by discretionary decisions of external parties.
- All logic is transparent and verifiable in the contract's code.
- Ensures predictable outcomes based on input conditions.

4) **Immutability:**

- Once deployed, the core logic of a smart contract typically cannot be altered.
- Enhances trust as users can be certain of the contract's behaviour.
- Challenges arise when upgrades or modifications are necessary.

5) **Deterministic Execution:**

- Given the same input and blockchain state, a smart contract will always produce the same output.
- Critical for maintaining consensus across a decentralized network.

4.3.5.3 Importance of Autonomy

This clause explores the critical importance of autonomy in smart contract systems, highlighting how this feature contributes to their efficiency, reliability, and trustworthiness.

1) **Trustlessness:**

- Autonomy reduces the need to trust intermediaries or central authorities.
- The contract's behaviour is determined by its code, which is visible and verifiable.
- Enhances transparency and reduces the risk of manipulation or fraud.

2) **Efficiency:**

- Automated, autonomous execution can significantly reduce transaction times and costs.
- Eliminates the need for manual processing and intermediaries.
- Enables 24/7 operation without human intervention.

3) **Consistency:**

- Autonomous operation ensures consistent application of rules and conditions across all interactions.
- Reduces errors and discrepancies that can occur with manual processes.
- Particularly valuable in complex, multi-party agreements.

4) **Decentralization:**

- Autonomy is a key enabler of decentralized systems and applications.
- Allows the creation of trustless, peer-to-peer networks and marketplaces.

5) **Programmable Money:**

- Enables the creation of sophisticated financial instruments and protocols.
- Facilitates complex, condition-based transactions that were previously impractical.

4.3.5.4 Autonomy Implementation Considerations

Implementing autonomy in smart contracts requires careful consideration of various technical and design factors to ensure the contract can operate independently and securely. This clause explores the key considerations developers should address when implementing autonomous functionality in smart contracts.

1) **Trigger Mechanisms:**

[R42] Developers **SHALL** design contracts with clear trigger conditions that initiate autonomous actions.

[D141] Developers **SHOULD** consider various types of triggers: time-based, event-based, or state-based.

EXAMPLE: An automated market maker contract that rebalances based on price changes.

2) **State Management:**

[R43] Developers **SHALL** ensure the contract can autonomously manage and update its state based on interactions and conditions.

[D142] Developers **SHOULD** implement robust data structures to track relevant information.

[D143] Developers **SHOULD** use storage efficiently to minimize gas costs while maintaining necessary state.

3) **External Data Handling:**

[D144] Developers **SHOULD** consider how the contract will autonomously interact with external data sources (oracles) when necessary.

[D145] Developers **SHOULD** implement mechanisms to verify and validate external data to maintain trust.

[D146] Developers **SHOULD** design fallback mechanisms in case of oracle failures or data inconsistencies.

4) **Gas Considerations:**

[D147] Developers **SHOULD** optimize contract logic to minimize gas costs for autonomous operations.

[D148] Developers **SHOULD** consider implementing gas price strategies for time-sensitive autonomous actions.

5) **Interaction with Other Contracts:**

[D149] Developers **SHOULD** design patterns for autonomous interactions between multiple contracts.

[D150] Developers **SHOULD** implement safeguards to handle potential failures in inter-contract communications.

4.3.5.5 Challenges Associated with Autonomy

While autonomy is a fundamental feature of smart contracts, it also presents several challenges that developers and stakeholders have to carefully consider. This clause explores the key challenges associated with implementing and maintaining autonomous smart contracts in distributed ledger environments. Understanding these challenges is crucial for creating robust, secure, and effective smart contract systems that can operate independently while mitigating potential risks and unintended consequences.

1) **Complexity:**

- Autonomous systems can become complex, making them difficult to design, test, and maintain.
- Increased complexity can lead to higher risks of bugs or vulnerabilities.
- Requires sophisticated testing and simulation environments.

2) **Unforeseen Scenarios:**

- It is challenging to anticipate and code for all possible scenarios, which can lead to unexpected behaviours.
- Edge cases may emerge that were not considered during development.
- Market manipulations or extreme conditions can expose vulnerabilities.

3) **Upgradability:**

- Fully autonomous contracts may be difficult to upgrade or modify once deployed, potentially limiting flexibility.
- Balancing immutability with the need for upgrades is a significant challenge.
- Requires careful consideration of upgrade patterns and governance mechanisms.

4) **Regulatory Compliance:**

- Autonomous execution may conflict with regulatory requirements in certain jurisdictions.
- Challenges in implementing KYC/AML procedures in fully autonomous systems.

5) **User Understanding and Trust:**

- Users may find it difficult to trust or understand fully autonomous systems.
- Lack of human oversight can be concerning for high-value transactions.

4.3.5.6 Balancing Autonomy and Control

While autonomy is a key feature of smart contracts, there may be scenarios where some level of external control is desirable.

[O10] Developers **MAY** follow these guidelines:

1) **Emergency Stops:**

- Implementing mechanisms to pause or stop a contract in case of critical issues.
- Often referred to as "circuit breakers" or "pause functions".
- Should be used judiciously to maintain trust in the system's autonomy.

2) **Governance Mechanisms:**

- Allowing for community-driven decisions to modify certain parameters or behaviours of the contract.
- Can be implemented through voting systems or Decentralized Autonomous Organizations (DAOs).
- Balances autonomy with adaptability and community control.

3) **Upgradeable Patterns:**

- Using proxy patterns or other techniques to allow for upgrades while maintaining a degree of autonomy.
- Enables bug fixes and feature improvements without compromising core autonomous behaviours.
- Requires careful design to prevent centralization or unauthorized changes.

4) **Parameterization:**

- Allowing certain contract parameters to be adjusted without changing core logic.
- Enables fine-tuning of contract behaviour without compromising autonomy.

5) **Modular Design:**

- Breaking down complex systems into smaller, autonomous modules.

- Allows for upgrading or replacing individual components while maintaining overall system autonomy.

4.3.5.7 Autonomy in Different Contexts

Autonomy can be realized in multiple contexts, each with its unique characteristics and influence on the behaviour of smart contracts:

1) **Financial Contracts:**

   - Autonomous execution of trades, loans, or insurance payouts based on predefined conditions.

EXAMPLES: Automated market makers, yield farming protocols, decentralized insurance.

CHALLENGES: Handling market volatility, ensuring fairness in autonomous trades.

2) **Supply Chain:**

   - Automatic tracking and verification of goods movement and ownership transfers.
   - IoT integration for real-time, autonomous updates of supply chain status.

CHALLENGES: Integrating with physical world events, handling disputes autonomously.

3) **Governance:**

   - Autonomous execution of voting results or organizational rules in Decentralized Autonomous Organizations (DAOs).
   - Implementing complex governance structures with multiple stakeholders.

CHALLENGES: Balancing automation with the need for human judgment in complex decisions.

4) **Identity and Access Management:**

   - Autonomous verification of credentials and granting of access rights.
   - Self-sovereign identity systems with autonomous data sharing and verification.

CHALLENGES: Ensuring privacy and compliance with data protection regulations.

5) **Gaming and Virtual Worlds:**

   - Autonomous management of in-game economies and asset ownership.
   - Implementing complex game mechanics through autonomous smart contracts.

CHALLENGES: Balancing fairness, preventing exploits in autonomous systems.

4.3.5.8 Best Practices for Implementing Autonomy in Smart Contracts

[D151] The following best practices **SHOULD** be followed to ensure the integrity of autonomous smart contracts:

1) **Thorough Testing:**

   - Rigorously test autonomous behaviours under various scenarios to ensure reliability.
   - Utilize formal verification techniques where possible to mathematically prove contract behaviours.
   - Implement comprehensive unit tests, integration tests, and simulation of edge cases.

2) **Clear Documentation:**

   - Provide comprehensive documentation of the autonomous behaviours for users and auditors.
   - Clearly explain trigger conditions, state changes, and potential outcomes of autonomous actions.

- Use natspec comments in code to enhance readability and auto-generate documentation.

3) **Fail-Safe Mechanisms:**

- Implement safeguards to handle unexpected situations without compromising the system's integrity.
- Design graceful failure modes that prioritize fund safety and system stability.
- Consider implementing rate limiting or circuit breakers for extreme scenarios.

4) **Transparent Monitoring:**

- Implement event emission for all significant autonomous actions.
- Provide dashboards or interfaces for users to monitor contract state and autonomous behaviours.
- Consider off-chain monitoring solutions to alert stakeholders of unusual activities.

5) **Gradual Rollout:**

- Start with limited autonomy and gradually increase as the system proves reliable.
- Use feature flags or tiered autonomy levels to control the extent of autonomous behaviour.

6) **Regular Audits:**

- Conduct thorough third-party audits of autonomous contract systems.
- Implement a bug bounty program to incentivize discovery of vulnerabilities.

7) **Gas Efficiency:**

- Optimize autonomous functions for gas efficiency to ensure reliable execution.
- Consider implementing dynamic gas price strategies for time-sensitive autonomous actions.

8) **Upgradability Planning:**

- Design with future upgradability in mind, even if immediate upgrades are not planned.
- Implement clear upgrade processes that maintain trust and decentralization.

4.3.5.9 Requirements and Recommendations

[R44] Smart contracts **SHALL** execute automatically when predefined conditions are met, without requiring manual intervention.

[R45] Once deployed, smart contracts **SHALL** operate independently of their creators or any central authority.

[R46] Smart contracts **SHALL** be self-enforcing, automatically executing the coded rules and conditions.

[R47] The execution of smart contract functions **SHALL** be determined solely by the contract's code and the current blockchain state.

[D152] Developers **SHOULD** implement clear and comprehensive trigger conditions for autonomous contract execution.

[D153] Smart contracts **SHOULD** include mechanisms for self-management of their state and resources.

[D154] Contracts **SHOULD** implement safeguards and circuit breakers to handle unexpected scenarios autonomously.

[D155] Developers **SHOULD** design contracts with clear separation between autonomous operations and functions requiring human intervention.

[D156] Smart contracts **SHOULD** emit events for significant autonomous actions to provide transparency and auditability.

[D157] Contracts **SHOULD** include mechanisms for autonomous interaction with other contracts and external systems (e.g. oracles) when necessary.

[D158] Developers **SHOULD** implement tiered autonomy, where certain high-impact actions require multi-signature approval or time locks.

[D159] Smart contracts **SHOULD** include self-diagnostic capabilities to monitor their own state and performance.

[D160] Contracts **SHOULD** be designed with modular architecture to allow for autonomous upgrades of specific components.

[D161] Developers **SHOULD** implement mechanisms for contracts to autonomously manage and optimize their resource usage (e.g. gas costs in Ethereum).

[D162] Smart contracts **SHOULD** include fallback mechanisms to ensure continued operation in case of failures in external dependencies.

[D163] Contracts designed for long-term operation **SHOULD** include mechanisms for autonomous adaptation to changing network conditions or regulations.

4.3.6 Decentralization

4.3.6.1 Definition

Decentralization in the context of smart contracts refers to the distribution of contract execution, validation, and storage across multiple nodes in a network, rather than relying on a single central authority. This property is fundamental to the trustless and resilient nature of blockchain-based smart contract systems. Decentralization is a core principle of blockchain-based smart contract systems, offering unique advantages in terms of trust, resilience, and censorship resistance. However, it also introduces complexities and trade-offs that should be carefully considered in system design and implementation. The appropriate level and form of decentralization can vary depending on the specific requirements and constraints of each use case. As the field evolves, new approaches to balancing the benefits of decentralization with practical considerations continue to emerge, shaping the future of smart contract systems [169], [156], [177], [173].

4.3.6.2 Key Aspects of Decentralization

1) **Distributed Execution:** Smart contracts are executed independently by multiple nodes in the network, ensuring that no single point of failure can disrupt the entire system. This distributed execution is facilitated by the blockchain's decentralized architecture, where each node has a copy of the contract code and state.

2) **Consensus Mechanisms:** Agreement on the state and outcomes of smart contract executions is achieved through consensus among network participants. This consensus mechanism ensures that all nodes agree on the validity of transactions and the state of the contract, maintaining the integrity of the system.

3) **Redundancy:** Contract code and state are replicated across multiple nodes, enhancing resilience and availability. This redundancy ensures that even if some nodes fail or behave maliciously, the system can continue to function without interruption.

4.3.6.3 Importance of Decentralization

1) **Trust Minimization:** Decentralization reduces the need to trust any single entity, as the integrity of the system is maintained collectively by the network participants. This trust minimization is crucial in ensuring the security and reliability of smart contracts.

2) **Resilience:** The system can continue to function even if some nodes fail or behave maliciously. This resilience is a direct result of the decentralized architecture, where multiple nodes can execute and validate contracts independently.

3) **Censorship Resistance:** It becomes difficult for any single entity to censor or manipulate contract executions. This censorship resistance is a key benefit of decentralization, ensuring that transactions are executed fairly and transparently.

4) **Transparency:** Decentralized systems often provide greater transparency, as multiple parties can verify contract executions. This transparency fosters trust among participants and ensures the accuracy and integrity of the data stored on the blockchain.

4.3.6.4 Decentralization Implementation Considerations

1) **Consensus Algorithms:** Choice of consensus mechanism (e.g. Proof of Work, Proof-of-Stake, or Permissioned consensus) affects the degree and nature of decentralization. The choice of consensus algorithm is critical in determining the level of decentralization and the security of the system.

2) **Network Architecture:** The structure of the network (public, private, or consortium) influences the level of decentralization. Public blockchains are fully decentralized, while private and consortium blockchains may have a more centralized architecture.

3) **Data Distribution:** Strategies for distributing and synchronizing contract data across the network are crucial in maintaining the integrity and availability of the system. This includes ensuring that all nodes have a consistent view of the contract state and data.

4.3.6.5 Challenges associated with Decentralization

1) **Performance Trade-offs:** Decentralized execution can be slower and less efficient compared to centralized systems. This is due to the need for consensus among multiple nodes, which can introduce latency and reduce throughput.

2) **Scalability:** Maintaining decentralization while scaling the system can be challenging. As the number of nodes increases, the complexity of achieving consensus and maintaining data consistency also increases.

3) **Governance:** Decentralized decision-making for system upgrades or changes can be complex. This includes ensuring that all stakeholders are involved in the decision-making process and that changes are implemented in a transparent and secure manner.

4) **Regulatory Compliance:** Decentralized systems may face challenges in complying with regulatory requirements designed for centralized systems. This includes ensuring that the system meets requirements for data protection, anti-money laundering, and know-your-customer regulations.

4.3.6.6 Degrees of Decentralization

Decentralization in smart contracts is not a binary concept but rather exists on a spectrum. This clause explores the various degrees of decentralization that can be implemented in smart contract systems. By examining the nuances of different levels of decentralization, developers can better tailor smart contract architectures to specific use cases and regulatory environments.

1) **Fully Decentralized:** Public blockchain networks where anyone can participate in contract execution and validation. These networks are fully decentralized, with no central authority controlling the system.

2) **Partially Decentralized:** Consortium or permissioned networks where a select group of participants manage the system. These networks are partially decentralized, with a central authority controlling access to the system

3) **Hybrid Models:** Systems that combine elements of centralized and decentralized architectures for different aspects of operation. These hybrid models can offer a balance between security, performance, and regulatory requirements.

4.3.6.7 Decentralization in Different Contexts

Decentralization is a fundamental principle of blockchain technology, but its implementation and implications can vary significantly across different contexts and use cases. This clause explores the different contexts of decentralization.

1) **Execution Decentralization:** Distribution of contract execution across multiple nodes. This ensures that no single point of failure can disrupt the entire system.

2) **Storage Decentralization:** Distribution of contract code and state across the network. This ensures that all nodes have a consistent view of the contract state and data.

3) **Governance Decentralization:** Distribution of decision-making power for system changes and upgrades. This ensures that all stakeholders are involved in the decision-making process and that changes are implemented in a transparent and secure manner.

4.3.6.8 Best Practices of Decentralization

Decentralization is a core principle of blockchain technology, but implementing it effectively in smart contracts requires careful consideration and adherence to best practices. This clause outlines key strategies and approaches for maximizing the benefits of decentralization in smart contract design and implementation. By following these best practices, developers can create more robust, transparent, and truly decentralized systems that align with the fundamental ethos of blockchain technology while addressing practical challenges and limitations.

1) **Appropriate Level of Decentralization:**

[D164] Developers **SHOULD** Choose a level of decentralization that balances security, performance, and regulatory requirements for the specific use case. This includes considering the trade-offs between decentralization and performance.

2) **Transparency:**

[D165] Developers **SHOULD** Provide clear information about the degree of decentralization in the system. This includes ensuring that all stakeholders understand the level of decentralization and the benefits and risks associated with it.

3) **Incentive Alignment:**

[D166] Developers **SHOULD** Design systems where participants are incentivized to maintain the decentralized nature of the network. This includes ensuring that participants are rewarded for contributing to the system and that malicious behaviour is discouraged.

4) **Regular Audits:**

[D167] Developers **SHOULD** Conduct audits to ensure the system maintains its intended level of decentralization over time. This includes regularly reviewing the system's architecture and ensuring that it remains secure and resilient.

4.3.6.9 Requirements and Recommendations

[R48] Smart contracts **SHALL** be designed to operate in a decentralized manner, without reliance on a single central authority.

[R49] The execution and validation of smart contracts **SHALL** be distributed across multiple nodes in the network.

[R50] Smart contracts **SHALL** use consensus mechanisms that align with the decentralized nature of the underlying blockchain.

[R51] The design of smart contracts **SHALL NOT** introduce centralization points that could compromise the overall decentralized architecture.

[D168] Developers **SHOULD** implement decentralized governance mechanisms for critical contract decisions or upgrades.

[D169] Smart contracts **SHOULD** use decentralized oracle solutions to fetch external data, avoiding reliance on a single data source.

[D170] Contracts **SHOULD** implement mechanisms for distributed key management and multi-signature approvals for high-value transactions.

[D171] Developers **SHOULD** consider using decentralized storage solutions (e.g. IPFS) for large data sets associated with smart contracts.

[D172] Smart contracts **SHOULD** be designed to operate effectively across different nodes with potentially varying computational capabilities.

[D173] Developers **SHOULD** implement mechanisms to incentivize node participation and maintain a healthy level of decentralization.

[D174] Contracts **SHOULD** include fallback mechanisms to ensure continued operation in case of node failures or network partitions.

[D175] Developers **SHOULD** consider implementing sharding or layer-2 solutions to enhance scalability while maintaining decentralization.

[D176] Smart contracts **SHOULD** use decentralized identity solutions where user authentication is required.

[D177] Developers **SHOULD** implement transparent and decentralized upgrade mechanisms when contract upgradability is necessary.

[D178] Contracts **SHOULD** use decentralized randomness sources when random number generation is required.

[D179] Developers **SHOULD** consider the trade-offs between decentralization and performance when designing smart contract systems.

[D180] Smart contracts **SHOULD** implement decentralized dispute resolution mechanisms where applicable.

[D181] Developers **SHOULD** use formal verification techniques to ensure that decentralized execution produces consistent results across all nodes.

[D182] Contracts **SHOULD** be designed with modularity to allow for easier upgrades and maintenance in a decentralized environment.

4.3.7 State Management

4.3.7.1 Definition

State management in smart contracts refers to the process of storing, accessing, and modifying the contract's data over time. The state represents the current condition of the contract, including all its variables and data structures, which can change as a result of transactions or interactions. State management is a critical aspect of smart contract design and implementation. It directly impacts the contract's functionality, performance, and cost-effectiveness. Effective state management requires careful consideration of data structures, access patterns, and the specific requirements of the contract's use case. As smart contract platforms evolve, new techniques and best practices for state management continue to emerge, offering improved solutions for scalability, efficiency, and security [169], [156], [177], [173].

4.3.7.2 Key Aspects of State Management

State management is a critical aspect of smart contract development, as it ensures that the contract's data is accurately stored, updated, and retrieved. Effective state management is essential for maintaining the integrity and reliability of the contract.

1) **State Variables:** Data stored within the contract that persists between function calls and transactions. State variables are used to store critical information, such as user balances, contract settings, and other relevant data.

2) **State Transitions:** Changes to the contract's state as a result of function executions or external interactions. State transitions are triggered by transactions, events, or other contract interactions, and they update the contract's state accordingly.

3) **State Consistency:** Ensuring that the contract's state remains valid and consistent across all nodes in the network. State consistency is crucial for maintaining the integrity of the contract and preventing errors or inconsistencies.

4.3.7.3 Importance of State Management

State management is essential for ensuring the accuracy, reliability, and integrity of smart contracts. Proper state management is critical for maintaining the contract's data and ensuring that it functions as intended.

1) **Data Integrity:** Proper state management ensures the accuracy and reliability of the contract's data. This includes ensuring that data is correctly stored, updated, and retrieved.

2) **Transaction Processing:** The current state determines how new transactions are processed and what outcomes they produce. Accurate state management is essential for ensuring that transactions are processed correctly.

3) **Business Logic:** The state often represents critical business information and rules encoded in the contract. Effective state management is essential for ensuring that the contract's business logic is executed correctly.

4) **Auditability:** A well-managed state allows for easier tracking and auditing of the contract's history and current condition. This includes ensuring that all state changes are properly recorded and tracked.

4.3.7.4 State Management Implementation Considerations

Implementing effective state management requires careful consideration of several factors, including storage optimization, access control, state synchronization, and versioning.

1) **Storage Optimization:** Efficiently managing state to minimize storage costs and improve performance. This includes using efficient data structures and minimizing the amount of data stored on-chain.

2) **Access Control:** Implementing proper controls on who can read or modify different aspects of the state. This includes using access control mechanisms, such as permissions and roles.

3) **State Synchronization:** Ensuring all nodes in the network maintain a consistent view of the contract's state. This includes using synchronization mechanisms, such as consensus algorithms.

4) **Versioning:** Managing state changes across different versions of the contract, especially during upgrades. This includes ensuring that state changes are properly tracked and updated.

4.3.7.5 Challenges associated with State Management

State management can be challenging, especially in complex contracts or distributed environments. Several challenges are associated with state management, including scalability, complexity, concurrency, and state bloat.

1) **Scalability:** As the state grows, it can impact the performance and cost of contract operations. This includes ensuring that the contract's state is efficiently managed and scaled.

2) **Complexity:** Managing complex state structures and relationships can be challenging, especially in large contracts. This includes ensuring that the contract's state is properly organized and managed.

3) **Concurrency:** Handling multiple simultaneous state changes in a distributed environment. This includes ensuring that state changes are properly synchronized and updated.

4) **State Bloat:** Over time, accumulated state data can lead to bloated contracts and increased costs. This includes ensuring that the contract's state is properly managed and cleaned up.

4.3.7.6 State Management Patterns

Several patterns can be used to manage state in smart contracts, including CRUD operations, mapping structures, hierarchical state, and event logging.

1) **CRUD Operations:** Implementing Create, Read, Update, and Delete operations for state management. This includes using CRUD operations to manage the contract's state.

2) **Mapping Structures:** Using key-value mappings for efficient state storage and retrieval. This includes using mapping structures to store and retrieve state data.

3) **Hierarchical State:** Organizing state in hierarchical structures for complex data relationships. This includes using hierarchical structures to manage complex state relationships.

4) **Event Logging:** Using events to track state changes and provide an audit trail. This includes using events to track and record state changes.

4.3.7.7 Advanced State Management Techniques

Several advanced techniques can be used to manage state in smart contracts, including state channels, commit-reveal schemes, and Merkle trees.

1) **State Channels:** Off-chain state management with periodic on-chain settlement for improved scalability. This includes using state channels to manage state off-chain.

2) **Commit-Reveal Schemes:** Two-step processes for managing state changes in scenarios requiring privacy or preventing front-running. This includes using commit-reveal schemes to manage state changes.

3) **Merkle Trees:** Efficient data structures for managing and verifying large sets of data. This includes using Merkle trees to manage and verify state data.

4.3.7.8 State Management Best Practices

Several best practices can be used to manage state in smart contracts, including minimizing state, state validation, gas optimization, clear state documentation, and state cleanup.

1) **Minimize State:**

[D183] Developers **SHOULD** store only essential data on-chain to reduce costs and improve performance. This includes minimizing the amount of data stored on-chain.

2) **State Validation:**

[D184] Developers **SHOULD** implement robust checks to ensure state changes maintain the contract's invariants. This includes validating state changes to ensure they are correct.

3) **Gas Optimization:**

[D185] Developers **SHOULD** design state operations with gas costs in mind, especially for frequently used functions. This includes optimizing state operations to minimize gas costs.

4) **Clear State Documentation:**

[D186] Developers **SHOULD** provide comprehensive documentation of the contract's state structure and management processes. This includes documenting the contract's state structure and management processes.

5) **State Cleanup:**

[D187] Developers **SHOULD** implement mechanisms to clean up or archive old or unused state data when possible. This includes cleaning up or archiving old or unused state data to reduce costs and improve performance.

6) **State Cleanup:** Implement mechanisms to clean up or archive old or unused state data when possible.

4.3.7.9 Requirements and Recommendations

[R52] Smart contracts **SHALL** maintain a consistent and valid state across all nodes in the network.

[R53] State variables in smart contracts **SHALL** be clearly defined and properly encapsulated.

[R54] Smart contracts **SHALL** implement secure mechanisms for state transitions and updates.

[R55] The contract state **SHALL** be verifiable by authorized parties at any given time.

[D188] Developers **SHOULD** use appropriate data structures (e.g. mappings, arrays) to efficiently manage contract state.

[D189] Smart contracts **SHOULD** implement access control mechanisms to restrict state modifications to authorized parties only.

[D190] Contracts **SHOULD** emit events for all significant state changes to provide an audit trail.

[D191] Developers **SHOULD** implement state validation checks to ensure the integrity of the contract's state after each transaction.

[D192] Smart contracts **SHOULD** use gas-efficient patterns for state management to optimize transaction costs.

[D193] Contracts **SHOULD** implement mechanisms to handle potential state inconsistencies due to network partitions or consensus failures.

[D194] Developers **SHOULD** consider using commit-reveal schemes for managing state updates that involve time-sensitive or confidential information.

[D195] Smart contracts **SHOULD** implement state migration strategies for upgradeable contracts to ensure seamless transitions.

[D196] Contracts **SHOULD** use appropriate data types and structures to minimize storage costs while maintaining necessary precision.

[D197] Developers **SHOULD** implement mechanisms to archive or prune historical state data to manage contract size over time.

[D198] Smart contracts **SHOULD** use modifiers to enforce state-dependent conditions on function execution.

[D199] Developers **SHOULD** consider implementing state channels or layer-2 solutions for high-frequency state updates to improve scalability.

[D200] Contracts **SHOULD** include functions to allow users to query their own state without exposing sensitive information about other users.

[D201] Developers **SHOULD** implement proper error handling and revert mechanisms to maintain state consistency in case of failed transactions.

[D202] Smart contracts **SHOULD** use the checks-effects-interactions pattern to prevent reentrancy attacks when updating state.

[D203] Developers **SHOULD** consider using formal verification techniques to prove the correctness of critical state transition logic.

[D204] Contracts **SHOULD** implement mechanisms to handle potential state rollbacks due to blockchain reorganizations.

[D205] Developers **SHOULD** use constant and immutable variables where appropriate to optimize gas costs and improve code readability.

[D206] Smart contracts **SHOULD** implement a clear separation between mutable and immutable state to enhance security and auditability.

[D207] Developers **SHOULD** consider implementing state checkpoints for complex contracts to allow easier debugging and state verification.

4.3.8 Interoperability

4.3.8.1 Definition

The rapid growth and diversification of blockchain networks and smart contract platforms have created a critical need for interoperability in decentralized systems. Currently, many smart contracts and blockchain applications operate in isolated environments, unable to communicate or share data effectively across different platforms or with off-chain systems. This lack of interoperability limits the potential of blockchain technology and hinders the development of more complex, powerful, and user-friendly decentralized applications.

Key challenges include:

1) **Standardization:** The absence of universally accepted protocols for cross-chain communication and data exchange.

2) **Security:** Ensuring the integrity and confidentiality of data as it moves between different blockchain environments and off-chain systems.

3) **Scalability:** Managing the increased complexity and potential performance bottlenecks introduced by cross-chain interactions.

4) **Platform Diversity:** Designing smart contracts that can operate seamlessly across multiple blockchain platforms with varying architectures and consensus mechanisms.

5) **Data Verification:** Implementing reliable methods for verifying and processing data from external sources, such as oracles or other blockchains.

6) **Complexity Management:** Balancing the need for advanced interoperability features (e.g. atomic swaps, cross-chain asset transfers) with the requirement for simplicity and ease of use.

7) **Regulatory Compliance:** Ensuring that interoperable systems adhere to diverse regulatory requirements across different jurisdictions.

Addressing these challenges is crucial for the continued evolution of blockchain technology and the broader adoption of decentralized applications. Successful solutions will need to balance technical innovation with practical considerations of security, usability, and regulatory compliance, ultimately enabling the creation of a more interconnected and efficient blockchain ecosystem.

Interoperability in smart contracts refers to the ability of different contracts, platforms, or blockchain networks to communicate, share data, and work together seamlessly. By prioritizing interoperability in smart contract architecture, developers can create more versatile and future-proof applications that can leverage the strengths of different blockchain networks and seamlessly integrate with existing systems, thereby driving broader adoption and functionality in the decentralized ecosystem [14], [131], [161], [94].

4.3.8.2 Key Aspects of Interoperability

Interoperability in smart contracts refers to the capability of different contracts and blockchain systems to communicate and work together seamlessly. This is crucial for building a cohesive ecosystem of decentralized applications.

1) **Cross-Contract Communication:** The ability for smart contracts to call functions and exchange data with other contracts is fundamental for creating complex, interconnected decentralized applications. This communication enables contracts to leverage functionalities from one another, enhancing their capabilities.

2) **Cross-Chain Interactions:** Mechanisms for contracts on different blockchain networks to interact and share information are essential for expanding the reach and functionality of decentralized applications across multiple platforms.

3) **Standardization:** Common interfaces and protocols facilitate interaction between different contracts and systems, ensuring that they can communicate effectively without compatibility issues. Standards like ERC-20 [46] for tokens on Ethereum are examples of such standardization efforts.

4) **Data Compatibility:** Ensuring data formats and structures are compatible across different platforms and contracts is vital for seamless data exchange and integration, preventing errors or misinterpretations during interactions.

4.3.8.3 Importance of Interoperability

Interoperability is crucial for the growth and functionality of blockchain ecosystems, enabling diverse applications to work together harmoniously.

1) **Ecosystem Development:** Interoperability fosters a more connected and robust ecosystem of decentralized applications by allowing them to interact and build upon each other's functionalities, leading to innovation and growth.

2) **Functionality Extension:** Allows contracts to leverage functionalities and data from other contracts or systems, enhancing their own capabilities without needing to implement everything from scratch.

3) **Asset Portability:** Enables the transfer and use of digital assets across different platforms or networks, increasing their utility and value by making them accessible in various contexts.

4) **Scalability Solutions:** Facilitates layer-2 solutions and sidechains that can interact with main networks, providing scalability improvements while maintaining connectivity with the primary blockchain.

4.3.8.4 Interoperability Implementation Mechanisms

Implementing interoperability requires specific mechanisms that enable seamless communication and interaction between different blockchain entities.

1) **Standardized Interfaces:** Implementing common interfaces (e.g. ERC standards in Ethereum) ensures consistent interaction patterns between contracts, simplifying integration and reducing errors.

2) **Oracles:** Using oracle services facilitates communication between smart contracts and external systems, allowing them to access real-world data or interact with off-chain services securely.

3) **Cross-Chain Bridges:** Mechanisms for transferring assets and information between different blockchain networks are crucial for enabling cross-chain interactions, expanding the functionality of decentralized applications across platforms.

4) **Atomic Swaps:** Enabling trustless exchanges of assets between different blockchain networks without intermediaries, ensuring secure and efficient cross-chain transactions.

4.3.8.5 Challenges associated with Interoperability

While interoperability offers significant benefits, it also introduces challenges that need to be addressed to ensure secure and efficient interactions.

1) **Security Risks:** Interoperability can introduce new attack vectors, especially in cross-chain interactions where vulnerabilities in one system might be exploited through another.

2) **Complexity:** Managing interactions between multiple contracts or systems increases complexity, requiring careful design and coordination to ensure smooth operation.

3) **Performance Overhead:** Cross-chain or external interactions may introduce latency and additional costs due to the need for extra verification steps or communication protocols.

4) **Governance Issues:** Coordinating upgrades and changes across interoperable systems can be challenging, as it requires alignment among multiple stakeholders with potentially differing priorities.

4.3.8.6 Interoperability Levels

Different levels of interoperability address various aspects of interaction between blockchain systems, from basic data exchange to full ecosystem integration.

1) **Syntactic Interoperability:** Ensuring data formats and communication protocols are compatible so that systems can exchange information without errors or misinterpretations.

2) **Semantic Interoperability:** Ensuring that the meaning of exchanged information is understood across different systems, allowing them to interpret data correctly in their specific contexts.

3) **Cross-Platform Interoperability:** Allowing interaction between contracts on different blockchain platforms, enabling broader functionality and asset transfer across ecosystems.

4) **Ecosystem Interoperability:** Enabling seamless interaction between various DApps, wallets, and services within a blockchain ecosystem, fostering a cohesive user experience.

4.3.8.7 Emerging Interoperability Solutions

Several innovative solutions are emerging to enhance interoperability across blockchain networks, addressing existing limitations and expanding possibilities.

1) **Polkadot Parachains:** A network of interconnected blockchain "parachains" that can easily interact with each other, providing a scalable solution for cross-chain communication within the Polkadot ecosystem. Polkadot is discussed in greater depth in clause 4.5.3 herewith.

2) **Cosmos Inter-Blockchain Communication (IBC):** A protocol for secure communication between independent blockchains, facilitating interoperability while maintaining sovereignty over their own operations.

3) **Ethereum Layer-2 Solutions:** Mechanisms like Optimistic Rollups and ZK-Rollups that interact with the main Ethereum network to improve scalability while maintaining security through periodic settlements on the main chain. Layer-2 solutions are discussed in greater depth in clause 4.4.9.2 herewith.

Examples of such solutions are listed in clause A.1.1.2 herewith.

4.3.8.8 Best Practices when Implementing Interoperability

Implementing interoperability effectively requires adherence to best practices that ensure compatibility, security, and efficiency in cross-system interactions.

1) **Use Established Standards:**

[D208] Developers **SHOULD** implement widely-accepted standards and interfaces to ensure compatibility across different systems, reducing integration complexity and potential errors.

2) **Modular Design:**

[D209] Developers **SHOULD** create modular contracts that can easily interact with other components, allowing flexibility in integrating new functionalities or adapting to changes in the ecosystem.

3) **Thorough Testing:**

[D210] Developers **SHOULD** Rigorously test interoperability features, especially for cross-chain interactions where unexpected issues may arise due to differing protocols or implementations.

4) **Clear Documentation:**

[D211] Developers **SHOULD** provide comprehensive documentation on how to interact with your contract from other contracts or systems, facilitating easier integration by developers from other projects.

5) **Security Considerations:**

[D212] Developers **SHOULD** implement robust security measures, especially when dealing with external calls or cross-chain transactions, to protect against potential vulnerabilities introduced by interoperability features.

4.3.8.9 Requirements and Recommendations

[R56] Smart contracts **SHALL** implement standardized interfaces to ensure interoperability within the PDL ecosystem.

[R57] Cross-chain communication protocols **SHALL** be implemented securely to facilitate interoperability between different blockchain networks.

[R58] Smart contracts **SHALL** use standardized data formats for input and output to ensure compatibility with other systems.

[R59] Interoperability mechanisms **SHALL NOT** compromise the security or integrity of the smart contract or its underlying blockchain.

[D213] Smart contracts **SHOULD** implement modular designs to facilitate easier integration with other systems.

[D214] Contracts **SHOULD** use oracles or bridge protocols for secure cross-chain data transfer and verification.

[D215] Developers **SHOULD** implement versioning mechanisms to manage compatibility across different contract versions.

[D216] Smart contracts **SHOULD** use standardized event emission formats to enable consistent off-chain monitoring and integration.

[D217] Contracts **SHOULD** implement fallback mechanisms to handle interactions with non-compliant or outdated systems.

[D218] Developers **SHOULD** consider implementing atomic swap protocols for cross-chain asset transfers.

[D219] Smart contracts **SHOULD** use standardized identity and authentication protocols to ensure interoperability of user credentials across systems.

[D220] Contracts **SHOULD** implement clear error handling and status codes to facilitate troubleshooting in cross-system interactions.

[D221] Developers **SHOULD** consider implementing proxy patterns to allow for contract upgrades without breaking existing integrations.

[D222] Smart contracts **SHOULD** use standardized metadata formats to provide self-descriptive interfaces for automated discovery and integration.

[D223] Contracts **SHOULD** implement mechanisms to handle potential differences in consensus finality across interacting blockchain systems.

[D224] Developers **SHOULD** consider implementing state channels or layer-2 solutions that are compatible across multiple blockchain platforms.

[D225] Smart contracts **SHOULD** use standardized cryptographic primitives to ensure compatibility of security mechanisms across systems.

[D226] Smart Contracts **SHOULD** implement mechanisms to handle potential differences in transaction speed and cost across interacting systems.

[D227] Developers **SHOULD** consider implementing cross-chain governance mechanisms for contracts that operate across multiple networks.

[D228] Smart contracts **SHOULD** use standardized time representations to ensure consistent temporal logic across different systems.

[D229] Smart Contracts **SHOULD** implement mechanisms to handle potential differences in data privacy regulations across interacting jurisdictions.

[D230] Developers **SHOULD** consider implementing interoperable storage solutions for managing large datasets across multiple systems.

[D231] Smart contracts **SHOULD** provide clear documentation of their interoperability features and requirements to facilitate integration efforts.

4.3.9 Threats and Security

4.3.9.1 Security Aspects of Smart Contracts

Security in smart contracts involves implementing measures to protect contracts, their assets, and users from vulnerabilities, attacks, and unintended behaviours, ensuring they operate safely and reliably in potentially adversarial environments. Given their immutable and financial nature, security vulnerabilities in smart contracts can have severe consequences, necessitating a multi-layered approach combining rigorous development practices, advanced tools, and ongoing vigilance. As technology evolves, so does the need for continual advancement in security measures. Smart contracts face security-related limitations due to blockchain characteristics and programming complexities. Immutability ensures security against unauthorized changes but bugs are permanent once deployed, requiring rigorous testing and auditing beforehand. Their deterministic nature restricts interactions with external systems without oracles, introducing risks if compromised. Common vulnerabilities include reentrancy attacks, integer overflows/underflows, and access control issues from programming errors or inadequate understanding of blockchain requirements. Additionally, the public nature of blockchain data poses privacy challenges; transparency can expose sensitive information if not managed properly. Developers should balance transparency with privacy by implementing cryptographic techniques and off-chain storage solutions where necessary. By understanding these limitations and implementing robust security practices, developers can enhance the resilience of smart contracts against common threats while leveraging their benefits for decentralized applications [111], [28], [2].

Requirements:

[R60] Smart contracts **SHALL** undergo thorough testing and auditing before deployment to identify and mitigate potential vulnerabilities.

[R61] Smart Contracts **SHALL** implement secure mechanisms for handling external data inputs through decentralized oracle networks.

[R62] Developers **SHALL** use safe math libraries or Solidity 0.8.0+ to prevent integer overflow and underflow vulnerabilities.

Recommendations:

[D232] Developers **SHOULD** implement formal verification techniques to mathematically prove the correctness of critical contract functions.

[D233] Smart contracts **SHOULD** use proxy patterns or modular designs to allow for upgrades without compromising security.

[D234] Smart Contracts **SHOULD** employ privacy-preserving techniques such as zero-knowledge proofs to protect sensitive data while maintaining transparency.

4.3.9.2 Key Aspects

Some of the Key Aspects that smart contract developers should consider to ensure robust security are listed herewith. These key aspects include code integrity to prevent vulnerabilities and logical errors, access control to restrict function execution and data modification to authorized parties only, data protection to safeguard sensitive information, asset safety to protect digital assets managed by the contract from unauthorized access or theft, and execution integrity to ensure the contract performs as intended even in the presence of malicious actors. These aspects collectively form a comprehensive security framework for smart contracts, addressing potential vulnerabilities at various levels of contract design and operation [111], [122], [169].

Requirements:

[R63] Smart contracts **SHALL** implement robust mechanisms to ensure code integrity throughout their lifecycle.

[R64] Access control measures **SHALL** be implemented to restrict function calls and state modifications to authorized parties only.

[R65] Smart Contracts **SHALL** implement secure data protection mechanisms for handling sensitive information.

[R66] Smart contracts **SHALL** include measures to protect digital assets from unauthorized access or theft.

[R67] Execution integrity **SHALL** be maintained to ensure the contract performs as intended, even under adversarial conditions.

Recommendations:

[D235] Developers **SHOULD** use formal verification techniques to mathematically prove the correctness of critical contract functions.

[D236] Smart contracts **SHOULD** implement Role-Based Access Control (RBAC) for fine-grained permissions management.

[D237] Smart Contracts **SHOULD** use encryption or off-chain storage solutions for sensitive data that is not to be publicly visible.

[D238] Developers **SHOULD** implement multi-signature or time-lock mechanisms for high-value transactions or critical state changes.

[D239] Smart contracts **SHOULD** include circuit breaker mechanisms to pause operations in case of detected anomalies.

[D240] Smart Contracts **SHOULD** emit events for all significant actions to create an auditable trail of operations.

[D241] Developers **SHOULD** conduct regular security audits and penetration testing on smart contracts.

[D242] Smart contracts **SHOULD** implement secure random number generation techniques when randomness is required.

[D243] Smart Contracts **SHOULD** use established libraries and patterns (e.g. OpenZeppelin) for common security functions.

[D244] Developers **SHOULD** implement comprehensive input validation to prevent injection attacks or unexpected inputs.

4.3.9.3 Common Vulnerabilities and Attacks

4.3.9.3.1 Introduction

Most smart contracts suffer common vulnerabilities and experience common attacks. These include reentrancy attacks, where a malicious contract can repeatedly call back into the vulnerable contract before the first invocation is completed; integer overflow/underflow, which can lead to unexpected arithmetic results; front-running, where attackers exploit the public nature of transactions for their benefit; access control issues resulting from improperly implemented permissions; oracle manipulation, where external data sources are compromised; and Denial of Service (DoS) attacks that prevent the contract from functioning by manipulating gas costs or exploiting logic flaws. Understanding these common vulnerabilities is crucial for developers to implement effective countermeasures and create more secure smart contracts.

The following clauses discuss Internal Threats, with emphasis on threats caused by programming errors, and External Threats [28], [123], [111].

Requirements:

[R68] Smart contracts **SHALL** implement protection against reentrancy attacks using established patterns such as checks-effects-interactions.

[R69] Smart Contracts **SHALL** use safe math libraries or solidity 0.8.0+ to prevent integer overflow and underflow vulnerabilities.

[R70] Smart contracts **SHALL** implement access control mechanisms to prevent unauthorized access to sensitive functions and data.

[R71] Smart Contracts interacting with oracles **SHALL** use decentralized oracle networks or implement oracle result validation mechanisms.

Recommendations:

[D245] Developers **SHOULD** implement transaction ordering dependencies (e.g. commit-reveal schemes) to mitigate front-running attacks.

[D246] Smart contracts **SHOULD** include rate limiting and gas limiting mechanisms to prevent DoS attacks.

[D247] Developers **SHOULD** use static analysis tools to identify potential vulnerabilities before deployment.

[D248] Smart Contracts **SHOULD** implement event monitoring for detecting and responding to suspicious activities.

[D249] Developers **SHOULD** conduct thorough testing, including fuzzing and symbolic execution, to identify potential vulnerabilities.

[D250] Smart contracts **SHOULD** use the latest compiler version and enable all relevant security checks.

[D251] Developers **SHOULD** implement secure randomness generation techniques when randomness is required in the contract.

[D252] Smart Contracts **SHOULD** use standardized and audited libraries for common functionalities to reduce the risk of vulnerabilities.

[D253] Developers **SHOULD** implement proper error handling and revert mechanisms to maintain contract integrity during exceptional conditions.

[D254] Smart contracts **SHOULD** undergo regular security audits by reputable third-party firms.

4.3.9.3.2 Internal Threats

Internal threats to smart contract security refer to vulnerabilities and risks that originate from within the blockchain environment itself, as opposed to external actors or systems. These threats can arise from errors in the smart contract code, inadequate access controls, or flaws in the underlying blockchain platform. Common internal threats include reentrancy attacks, where a function can be repeatedly called before its previous execution is completed, leading to unexpected behaviour or exploitation. Another significant threat is improper handling of integer overflows and underflows, which can result in incorrect calculations and potential financial losses. Additionally, inadequate access control mechanisms can allow unauthorized users to execute sensitive functions or modify critical data within the contract. To mitigate these internal threats, developers should implement rigorous coding standards, conduct thorough testing and audits, and use established security patterns such as the checks-effects-interactions pattern to prevent reentrancy issues. These measures help ensure that smart contracts are resilient against internal threats by enhancing their security posture through careful design and proactive risk management practices [111], [28], [123].

Requirements:

[R72] Smart contracts **SHALL** implement robust access control mechanisms to restrict function execution and data modification to authorized parties only.

[R73] Developers **SHALL** use safe math libraries (such as Solidity 0.8.0+) to prevent integer overflow and underflow vulnerabilities.

Recommendations:

[D255] Developers **SHOULD** use static analysis tools to identify potential vulnerabilities in the code before deployment.

[D256] Smart contracts **SHOULD** implement the checks-effects-interactions pattern to prevent reentrancy attacks.

[D257] Regular security audits **SHOULD** be conducted to identify and address any internal threats or vulnerabilities.

4.3.9.3.3 Smart Contract Programming Errors

Smart contract programming errors are a significant source of internal vulnerabilities that can lead to various attacks and unintended behaviours. These errors often arise from the complexity of smart contract languages, inadequate testing, or misunderstanding of blockchain-specific programming paradigms. Common programming errors include reentrancy vulnerabilities, where a function can be repeatedly called before its initial execution is completed, leading to potential exploits. Integer overflow and underflow errors occur when arithmetic operations exceed the maximum or minimum values that can be stored, resulting in incorrect calculations and potential financial losses. Another frequent issue is improper access control, allowing unauthorized users to execute sensitive functions or modify critical data within the contract. To mitigate these risks, developers should implement rigorous testing protocols, use established security patterns such as checks-effects-interactions, and employ static analysis tools to identify potential vulnerabilities before deployment. By addressing these programming errors through careful design and robust security practices, developers can significantly enhance the security and reliability of smart contracts in blockchain ecosystems [123], [111], [28].

Requirements:

[R74] Smart contracts **SHALL** implement safe math libraries (such as Solidity 0.8.0+) to prevent integer overflow and underflow.

[R75] Contracts **SHALL** include comprehensive input validation to prevent injection attacks and unexpected inputs.

Recommendations:

[D258] Developers **SHOULD** conduct thorough testing, including fuzzing and symbolic execution, to uncover potential vulnerabilities.

[D259] Smart contracts **SHOULD** use static analysis tools to identify common programming errors before deployment.

[D260] Regular code reviews and security audits **SHOULD** be performed to ensure adherence to best practices and identify potential issues.

4.3.9.3.4 External Threats

External threats to smart contracts arise from interactions with entities outside the blockchain environment, including users, oracles, and other external systems. These threats can exploit vulnerabilities in how smart contracts handle external inputs or interact with off-chain data sources. One common external threat is oracle manipulation, where attackers compromise the data provided by oracles to influence the behaviour of smart contracts. Another significant threat is phishing attacks, where malicious actors deceive users into interacting with fraudulent smart contracts. Additionally, Denial of Service (DoS) attacks can be executed by overwhelming a contract with transactions, potentially leading to increased gas costs or service disruption. To mitigate these external threats, developers should implement secure oracle solutions, conduct regular audits of external interactions, and employ robust authentication mechanisms for user interactions [2], [28], [161].

Requirements:

[R76] Smart contracts **SHALL** implement secure mechanisms for handling data from external sources, such as decentralized oracle networks.

[R77] Contracts **SHALL** include input validation and authentication processes to verify the integrity and authenticity of external interactions.

Recommendations:

[D261] Developers **SHOULD** use multi-source oracle solutions to prevent single points of failure and reduce the risk of data manipulation.

[D262] Smart contracts **SHOULD** implement rate limiting and other protective measures to mitigate the impact of potential DoS attacks.

[D263] Regular security audits **SHOULD** be conducted to assess vulnerabilities related to external interactions and inputs.

By addressing these requirements and recommendations, smart contract developers can enhance the security posture of their contracts against external threats while maintaining robust functionality and user trust.

4.3.9.4 Advanced Smart Contract Security

Advanced smart contract security involves implementing sophisticated techniques and methodologies to protect contracts from emerging threats and vulnerabilities. As the complexity and adoption of smart contracts grow, so do the potential attack vectors. Advanced security measures include formal verification, which uses mathematical proofs to ensure the correctness of contract logic, thus preventing common vulnerabilities like reentrancy and integer overflows. Another critical aspect is the use of decentralized oracle networks to securely handle external data inputs, reducing the risk of data manipulation. Additionally, employing multi-signature wallets and time-lock mechanisms can enhance security by requiring multiple approvals for critical transactions, thereby mitigating risks associated with single points of failure. As quantum computing advances, integrating quantum-resistant cryptographic algorithms becomes increasingly important to protect against future threats. Regular security audits and penetration testing are essential to identify and address vulnerabilities proactively. By implementing these advanced security measures, developers can significantly enhance the resilience of smart contracts against both current and future threats in an ever-evolving digital landscape [159], [72], [2].

Requirements:

[R78] Smart contracts **SHALL** implement formal verification for critical functions to mathematically prove their correctness.

[R79] Contracts **SHALL** use decentralized oracle networks for secure handling of external data inputs.

[R80] Quantum-resistant cryptographic algorithms **CAN** be integrated when available and appropriate.

Recommendations:

[D264] Developers **SHOULD** employ multi-signature wallets and time-lock mechanisms for high-value transactions.

[D265] Regular security audits and penetration testing **SHOULD** be conducted to identify potential vulnerabilities.

[D266] Smart contracts **SHOULD** use secure random number generation techniques when randomness is required.

4.3.9.5 Security Best Practices and Culture in Smart Contracts

Security in smart contracts is paramount due to their immutable nature and financial implications, which make vulnerabilities potentially severe and irreversible. A robust security framework combines best practices with a strong security culture to protect contracts, assets, and users from vulnerabilities and attacks. Implementing security best practices involves using proven techniques and methodologies to enhance the robustness of smart contracts. Key practices include formal verification, comprehensive testing strategies, rigorous code reviews, and third-party audits. Developers should apply the principle of least privilege in access control, design fail-safe defaults for unexpected scenarios, implement rate limiting to prevent abuse, use secure upgrade patterns for contract evolution, and maintain comprehensive event monitoring for auditing purposes. By adhering to these practices, developers can significantly reduce vulnerabilities and create secure smart contracts. Fostering a strong security culture involves continuous learning about the latest threats and best practices, implementing bug bounty programs for vulnerability disclosure, developing incident response plans, and conducting regular security assessments. This culture ensures that teams are proactive in identifying and mitigating risks. Continuous learning programs should be established for team members to stay updated on security trends. Bug bounty programs encourage responsible vulnerability disclosure, while incident response plans prepare teams for potential breaches. Regular assessments help maintain a high-security posture [72], [101], [151].

Requirements:

[R81] Smart contracts **SHALL** undergo formal verification for critical functions to mathematically prove their correctness.

[R82] Comprehensive testing suites **SHALL** be implemented, covering unit, integration, and scenario-based tests.

[R83] Smart contracts **SHALL** be subject to both internal code reviews and external security audits before deployment.

[R84] The principle of least privilege **SHALL** be applied in designing access control mechanisms for smart contracts.

[R85] Development teams **SHALL** establish and maintain a continuous learning program focused on smart contract security.

[R86] Smart contract projects **SHALL** implement and maintain an incident response plan for security breaches.

[R87] Regular security assessments and penetration testing **SHALL** be conducted on all deployed smart contracts.

Recommendations:

[D267] Organizations **SHOULD** implement bug bounty programs to encourage the responsible disclosure of vulnerabilities in their smart contracts.

[D268] Developers **SHOULD** participate in security-focused workshops, conferences, and training programs to stay updated on the latest security threats and mitigation techniques.

[D269] Smart contract teams **SHOULD** conduct regular security drills to test and improve their incident response capabilities.

[D270] Organizations **SHOULD** foster a culture of security awareness among all team members involved in smart contract development and management.

[D271] Development teams **SHOULD** implement a secure code review process, including peer reviews and automated security analysis tools.

[D272] Smart contract projects **SHOULD** maintain comprehensive documentation of security practices, incident response procedures, and lessons learned from past security incidents.

[D273] Organizations **SHOULD** consider engaging external security auditors for regular third-party assessments of their smart contract systems.

[D274] Developers **SHOULD** implement fail-safe defaults to ensure graceful handling of unexpected scenarios.

[D275] Smart contracts **SHOULD** include rate limiting mechanisms to prevent abuse and potential DoS attacks.

[D276] Upgradeable contracts **SHOULD** use secure upgrade patterns, such as proxy patterns, to allow for future improvements.

[D277] Comprehensive event monitoring **SHOULD** be implemented for all significant contract actions to facilitate security auditing.

[D278] Developers **SHOULD** use automated analysis tools to identify potential vulnerabilities during the development process.

[D279] Smart contracts **SHOULD** implement input validation for all external data to prevent injection attacks.

[D280] Developers **SHOULD** follow the checks-effects-interactions pattern to prevent reentrancy vulnerabilities.

[D281] Smart Contracts **SHOULD** use standardized and audited libraries (e.g. OpenZeppelin) for common functionalities.

[D282] Developers **SHOULD** implement secure random number generation techniques when randomness is required.

[D283] Smart Contracts **SHOULD** undergo regular security assessments and penetration testing throughout their lifecycle.

4.3.9.6 Tools and Techniques

The Tools and Techniques listed herewith focus on the various software tools and methodologies used to enhance the security of smart contracts throughout their lifecycle. These include static analysis tools that scan code for known vulnerabilities and patterns, dynamic analysis tools that test contracts in simulated environments, fuzzing techniques that use invalid or random data as inputs to uncover potential issues, formal verification tools for mathematically proving contract properties, and specialized security frameworks designed for smart contracts. These tools and techniques collectively provide a comprehensive approach to identifying vulnerabilities, ensuring correct behaviour, and strengthening the overall security posture of smart contracts [37], [52], [111]. Examples of such tools and techniques are described in clause A.1.1.3.

Requirements:

[R88] Smart contract developers **SHALL** use static analysis tools to identify potential vulnerabilities before deployment.

[R89] Dynamic analysis **SHALL** be performed on smart contracts to identify runtime issues in simulated environments.

[R90] Formal verification tools **SHALL** be used for critical smart contract functions to mathematically prove their correctness.

Recommendations:

[D284] Developers **SHOULD** use fuzzing techniques to test smart contracts with unexpected or random inputs.

[D285] Smart contract security frameworks **SHOULD** be utilized to implement standardized security patterns and best practices.

[D286] Continuous integration pipelines **SHOULD** include automated security checks using various analysis tools.

[D287] Developers **SHOULD** use symbolic execution tools to explore multiple execution paths in smart contracts.

[D288] Smart contracts **SHOULD** undergo regular security audits using a combination of automated tools and manual expert review.

[D289] Developers **SHOULD** use gas analysis tools to optimize contract efficiency and prevent potential DoS vulnerabilities.

[D290] Mutation testing **SHOULD** be employed to evaluate the effectiveness of the contract's test suite.

[D291] Smart contracts **SHOULD** be tested on testnets that closely mimic mainnet conditions before final deployment.

[D292] Developers **SHOULD** use tools that analyse contract dependencies and their potential security implications.

[D293] Version control systems **SHOULD** be used in conjunction with security tools to track and manage security-related changes over time.

4.3.9.7 Regulatory and Compliance Considerations

To emphasize the importance of adherence to relevant regulations, such as data protection laws (e.g. GDPR) and financial regulations for contracts dealing with assets or financial services. This clause highlights the need to maintain comprehensive and tamper-proof audit logs for regulatory compliance. Recent research underscores the growing importance of regulatory compliance in smart contract development, particularly in areas like privacy preservation and financial services integration [3], [21], [140].

Requirements:

[R91] Smart contract developers **SHALL** ensure adherence and compliance with relevant regulations and maintain appropriate audit trails.

Recommendations:

[D294] Developers **SHOULD** implement privacy-preserving techniques such as zero-knowledge proofs for sensitive data when required.

[D295] Smart contracts **SHOULD** be designed to minimize the storage of personally identifiable information on-chain.

[D296] Contracts **SHOULD** implement comprehensive event logging to facilitate thorough auditing.

4.3.9.8 Emerging Security Challenges

Today's emerging security challenges in smart contracts include: quantum computing threats, cross-chain security, and privacy-preserving techniques. Smart Contract developers need to prepare for potential vulnerabilities introduced by quantum computing advancements, address security challenges in cross-chain interactions and interoperability scenarios, and implement advanced privacy technologies such as zero-knowledge proofs in a secure manner. These emerging challenges, and additional ones that may arise, require ongoing research and development of new security paradigms to ensure the long-term viability and security of smart contract systems in an evolving technological landscape [83], [159], [161].

Requirements:

[R92] Smart contracts **SHALL** implement quantum-resistant cryptographic algorithms when available and appropriate.

[R93] Cross-chain interactions **SHALL** be secured using robust verification mechanisms and protocols.

[R94] Privacy-preserving techniques **SHALL** be implemented in compliance with relevant data protection regulations.

Recommendations:

[D297] Developers **SHOULD** stay informed about advancements in post-quantum cryptography and plan for future integration.

[D298] Smart contracts **SHOULD** implement modular designs to facilitate easier upgrades of cryptographic components.

[D299] Cross-chain communication protocols **SHOULD** be regularly audited and updated to address new security vulnerabilities.

[D300] Contracts **SHOULD** use zero-knowledge proofs or secure multi-party computation when handling sensitive data across multiple chains.

[D301] Developers **SHOULD** implement and regularly update privacy-enhancing technologies to protect user data in smart contracts.

[D302] Smart contracts **SHOULD** include mechanisms for graceful degradation or shutdown in case of critical vulnerabilities discovered in underlying cryptographic primitives.

4.3.9.9 Security by design

4.3.9.9.1 The importance of Security in the design phase of smart contracts

Security is paramount in smart contract design, as vulnerabilities can lead to significant financial losses and compromise the integrity of the entire system. Implementing robust security measures is essential to protect against various attack vectors and ensure the contract's reliable operation. The topic of security is discussed in depth earlier in the present document in clause 4.3.9. The following text provides a summary and lists requirements and recommendations with the context of smart contract design practices.

4.3.9.9.2 Access Control

Implementing proper access control mechanisms is crucial to ensure that only authorized parties can execute sensitive functions or modify critical state variables. Role-Based Access Control (RBAC) and capability-based systems can provide fine-grained control over contract interactions, minimizing the risk of unauthorized access [175].

[R95] Smart contracts **SHALL** implement robust access control mechanisms.

[D303] Role-Based Access Control (RBAC) **SHOULD** be used for fine-grained permissions management.

4.3.9.9.3 Input Validation

Thorough input validation is essential to prevent malicious or erroneous inputs from compromising the contract's integrity. This includes checking for expected data types, ranges, and formats, as well as implementing safeguards against common attack vectors like integer overflow/underflow [72].

[R96] All inputs to smart contracts **SHALL** be thoroughly validated.

[D304] Contracts **SHOULD** implement checks for expected data types, ranges, and formats.

4.3.9.9.4 Reentrancy Protection

Reentrancy attacks are a type of vulnerability in smart contracts where an external malicious contract can repeatedly call back into the vulnerable contract before the first invocation is completed, potentially draining funds or manipulating the contract's state in unintended ways. This occurs when the contract sends funds or control to an external address before updating its own state, allowing the attacker to re-enter the contract and exploit the inconsistent state. Reentrancy attacks gained notoriety after the 2016 DAO hack on the Ethereum network and remain a significant security concern in smart contract development, necessitating careful design patterns and safeguards to prevent such vulnerabilities. Implementing the checks-effects-interactions pattern and using reentrancy guards can help prevent these vulnerabilities [119].

[D305] Developers **SHOULD** be particularly cautious when interacting with external contracts or transferring funds.

4.3.9.9.5 Gas Limitations and Denial of Service

[D306] Smart contracts **SHOULD** be designed to operate within gas limits to prevent denial of service attacks.

This includes avoiding unbounded loops, implementing efficient data structures, and considering the gas costs of all operations. Proper error handling for out-of-gas scenarios is also crucial [31].

4.3.9.9.6 Upgradability and Modularity

Implementing upgradable patterns and modular design can enhance security by allowing for bug fixes and improvements without compromising the entire system.

[R97] Upgradability mechanisms **SHALL** be carefully secured to prevent unauthorized modifications [93].

4.3.9.9.7 Formal Verification

Formal verification techniques can provide mathematical proof of a contract's correctness and security properties. While complex, these methods offer a high degree of assurance and can catch subtle vulnerabilities that might be missed by traditional testing approaches [102].

4.3.9.9.8 External Calls and Interactions

[R98] When interacting with external contracts or oracles, developers **SHALL** consider potential malicious behaviour or failures.

Implementing proper checks before and after external calls, using pull payment patterns, and carefully managing trust assumptions are essential for secure inter-contract communications [111].

4.3.9.9.9 Error Handling

[R99] Smart contracts **SHALL** implement comprehensive error handling mechanisms.

[D307] Contracts **SHOULD** gracefully handle exceptions and provide clear error messages.

4.3.10 Reusability

4.3.10.1 Definition

Reusability in smart contracts refers to the property that allows contract code, or parts of it, to be used multiple times in different contexts or applications. This concept promotes efficiency, consistency, and reliability in smart contract development. Reusability is a powerful concept in smart contract development that can significantly enhance efficiency, reliability, and standardization. By creating modular, well-designed, and thoroughly tested components, developers can build more robust and cost-effective smart contract systems. As the field matures, the importance of reusability is likely to grow, with increasing emphasis on creating and maintaining high-quality, reusable smart contract components and libraries. This trend will contribute to the overall maturation and professionalization of smart contract development practices [93].

4.3.10.2 Key Aspects

Key aspects of reusability include modularity, standardization, parameterization, and libraries. These aspects contribute to security by allowing focused security audits on individual components and promoting the use of well-tested, standardized code [177].

1) **Modularity:** Designing contracts as composable modules that can be easily integrated into various systems.
2) **Standardization:** Adhering to common interfaces and patterns to ensure compatibility and ease of integration.
3) **Parameterization:** Creating flexible contracts that can be customized through parameters without changing the core code.
4) **Libraries:** Developing and utilizing reusable libraries of common functionalities.

4.3.10.3 Importance

Reusability is important for development efficiency, code quality, consistency, and cost-effectiveness. From a security perspective, reusable components that have undergone thorough security testing can help reduce the overall attack surface [i.32].

1) **Development Efficiency:** Reduces development time and effort by leveraging pre-existing, tested code.
2) **Code Quality:** Reusable components tend to be more thoroughly tested and refined over time.
3) **Consistency:** Promotes consistent implementation of common functionalities across different projects.
4) **Cost-Effectiveness:** Reduces deployment costs by sharing common code across multiple contracts.

4.3.10.4 Implementation Strategies

Implementation strategies include contract templates, inheritance, library contracts, and factory patterns. These strategies can enhance security by promoting the use of well-tested code structures and patterns [175].

1) **Contract Templates:** Creating generalized contract structures that can be customized for specific use cases.
2) **Inheritance:** Utilizing contract inheritance to extend and customize base contracts.
3) **Library Contracts:** Developing separate library contracts that contain reusable functions.
4) **Factory Patterns:** Implementing factory contracts to create and deploy customized instances of contracts.

4.3.10.5 Challenges

Challenges in reusability include complexity management, version control, security considerations, and gas optimization. These challenges can impact security if not properly addressed, potentially introducing vulnerabilities or inefficiencies [31].

1) **Complexity Management:** Balancing generalization and specificity to maintain usability across different contexts.
2) **Version Control:** Managing different versions of reusable components and ensuring compatibility.
3) **Security Considerations:** Ensuring that reusable components are secure across various implementation contexts.
4) **Gas Optimization:** Balancing reusability with gas efficiency, especially for frequently used functions.

4.3.10.6 Best Practices

Best practices include clear documentation, extensive testing, semantic versioning, and using design patterns. These practices contribute to security by ensuring that reusable components are well-understood, thoroughly tested, and properly maintained [72].

1) **Clear Documentation:** Providing comprehensive documentation for reusable components, including usage guidelines and limitations.
2) **Extensive Testing:** Thoroughly testing reusable components across various scenarios and use cases.
3) **Semantic Versioning:** Implementing clear versioning strategies for reusable contracts and libraries.
4) **Design Patterns:** Utilizing established design patterns that promote reusability, such as the proxy pattern for upgradeable contracts.
5) **Community Standards:** Adhering to community-developed standards (e.g. ERC standards in Ethereum) to enhance interoperability and reusability.

4.3.10.7 Examples of Reusable Components

Examples of Reusability include token standards, access control modules, math libraries, and governance modules. These components can enhance security by providing standardized, well-tested implementations of common functionalities [148]. This below list of references provides insights into the implementation, benefits, and challenges of using reusable components in smart contract development:

1) **Token Standards:** Liu, Y., Lu, Q., Xu, X., Zhu, L., & Yao, H. discuss various design patterns in smart contracts, including the implementation of token standards like ERC-20 [46] and ERC-721 [44] as reusable components [93].

2) **Access Control Modules:** Zhu, Y., Qin, Y., Zhou, Z., Song, X., Liu, G., & Chu, W. C. C. present a framework for digital asset management using blockchain, featuring reusable access control modules based on attribute-based access control [175].

3) **Math Libraries:** An article written by Chen, T., Li, X., Luo, X., & Zhang, X. primarily focused on optimization, discusses the importance of efficient math libraries in smart contracts and their impact on gas costs [31].

4) **Governance Modules:** Wang, S., Ouyang, L., Yuan, Y., Ni, X., Han, X., & Wang, F. Y. wrote a comprehensive review that includes discussion on governance modules as reusable components in smart contract ecosystems [148].

5) **Standard Interfaces:** Groce, A., Zhang, Y., & Dziuban, S. introduce the concept of polymorphic smart contracts, which rely heavily on standard interfaces as reusable components [i.32].

6) **Utility Functions:** Zou, W., Lo, D., Kochhar, P. S., Le, X. B. D., Xia, X., Feng, Y., ... & Xu, B. conducted a comprehensive survey where they discuss various aspects of smart contract development, including the use of utility functions as reusable components [177].

7) **Security Patterns:** Huang, Y., Bian, Y., Li, R., Zhao, J. L., & Shi, P. wrote a paper presenting a lifecycle approach to smart contract security, discussing various security patterns that can be implemented as reusable components [72].

8) **Oracle Interfaces:** Al-Breiki, H., Rehman, M. H. U., Salah, K., & Svetinovic, D. discuss blockchain oracles, including standardized interfaces for oracle integration as reusable components in smart contracts [2].

4.3.10.8 Future Trends

Future trends in Reusability include cross-platform reusability, AI-assisted reusability, and ecosystem-specific libraries. These trends may impact security by introducing new ways to create and verify reusable components across different blockchain environments.

Included here are research papers discussing such future trends.

1) **Cross-Platform Reusability:** Belchior, R., Vasconcelos, A., Guerreiro, S., & Correia, M. conducted a comprehensive survey discussing blockchain interoperability, including the development of cross-platform smart contracts and reusable components that can operate across different blockchain environments [14].

2) **AI-Assisted Reusability:** ETSI GS PDL 032 [i.30] explores the integration of AI with blockchain technology, including the potential for AI to assist in creating and optimizing reusable smart contract components.

3) **Ecosystem-Specific Libraries:** Zheng, Z., Xie, S., Dai, H. N., Chen, W., Chen, X., Weng, J., & Imran, M. discuss the development of specialized smart contract platforms and the potential for ecosystem-specific component libraries [169].

4) **Automated Code Generation:** Chakraborty, P., Shahriyar, R., Iqbal, A., & Bosu, A. discuss emerging practices in blockchain software development, including the potential for automated code generation of reusable components [25].

5) **Formal Verification for Reusable Components:** Grishchenko, I., Maffei, M., & Schneidewind, C. discuss advanced static analysis techniques for smart contracts, which could be applied to ensure the correctness and security of reusable components [52].

6) **Dynamic Component Composition:** Liu, Y., Lu, Q., Xu, X., Zhu, L., & Yao, H. focused on current design patterns, however this paper also touches on future trends in smart contract composition, including more dynamic approaches to combining reusable components [93].

7) **Self-Evolving Smart Contracts:** Xu, X., Weber, I., Staples, M., Zhu, L., Bosch, J., Bass, L., ... & Rimba, P. discuss the potential for self-adaptive and self-evolving smart contracts, which could lead to new paradigms in reusable component design [156].

These papers provide insights into emerging trends and future directions in smart contract reusability, covering various aspects of technological advancement and cross-platform integration [95].

4.3.10.9 Requirements and Recommendations

[R100] Smart contracts **SHALL** implement standardized interfaces for common functionalities to ensure interoperability and reusability.

[R101] Reusable components **SHALL** undergo thorough security audits before being approved for widespread use.

[R102] Version control systems **SHALL** be used to manage different versions of reusable components.

[R103] Smart contracts **SHALL** include clear documentation for all reusable components, including usage guidelines and limitations.

[R104] Reusable components **SHALL** adhere to established coding standards and best practices within the specific blockchain ecosystem.

[D308] Developers **SHOULD** use established libraries and design patterns for common functionalities to reduce the risk of security vulnerabilities and improve reusability.

[D309] Smart contract systems **SHOULD** implement a modular architecture to enhance maintainability, reusability, and security.

[D310] Organizations **SHOULD** maintain a curated library of secure, reusable components specific to their ecosystem.

[D311] Developers **SHOULD** implement comprehensive unit tests for all reusable components to ensure their security and functionality.

[D312] Smart contracts **SHOULD** use inheritance and composition patterns to promote code reuse while maintaining security.

[D313] Development teams **SHOULD** regularly review and update reusable components to address newly discovered vulnerabilities or improvements in best practices.

[D314] Organizations **SHOULD** establish a process for vetting and approving third-party libraries and components before integration into smart contracts.

[D315] Developers **SHOULD** consider implementing parameterized designs for reusable components to enhance their flexibility and applicability across different contexts.

[D316] Smart contract platforms **SHOULD** provide tools and frameworks that facilitate the creation and management of reusable components.

[D317] Developers **SHOULD** implement clear error handling and fallback mechanisms in reusable components to ensure robustness in various usage scenarios.

[D318] Organizations **SHOULD** encourage knowledge sharing and documentation of best practices for creating and using reusable components.

[D319] Developers **SHOULD** consider cross-platform compatibility when designing reusable components to enhance their utility across different blockchain environments.

[D320] Smart contract auditors **SHOULD** pay special attention to the integration and usage of reusable components during security reviews.

[D321] Development teams **SHOULD** stay informed about emerging trends in smart contract reusability, such as AI-assisted development and dynamic composition techniques.

[D322] Organizations **SHOULD** implement governance processes for managing the lifecycle of reusable components, including deprecation and replacement strategies.

4.3.11 Composability and Contract Interactions

Composability in smart contracts refers to the ability to combine various contract components and functionalities to create more complex and integrated Decentralized Applications (DApps). This property is essential for building sophisticated blockchain systems where multiple contracts can interact seamlessly, allowing developers to leverage existing functionalities and create new applications without starting from scratch. Composability enhances the modularity and reusability of smart contracts, enabling developers to build on top of existing protocols and integrate with other services. This capability is crucial for the growth of Decentralized Finance (DeFi) and other blockchain ecosystems, where different protocols need to interact efficiently. However, composability also introduces challenges related to security, interoperability, and dependency management, as interactions between contracts can create vulnerabilities if not properly managed [14], [161].

Requirements:

[R105] Smart contracts **SHALL** implement standardized interfaces to ensure compatibility and interoperability with other contracts.

[R106] Cross-contract interactions **SHALL** be designed to maintain security and prevent vulnerabilities such as reentrancy attacks.

[R107] Contracts **SHALL** include mechanisms for managing dependencies and ensuring consistent execution across interactions.

Recommendations:

[D323] Developers **SHOULD** use modular design patterns to facilitate composability and ease of integration with other contracts.

[D324] Smart contracts **SHOULD** implement thorough testing for all interaction pathways to identify potential vulnerabilities or failures.

[D325] Developers **SHOULD** leverage existing libraries and frameworks that provide secure composability features to enhance reliability.

[D326] Contracts **SHOULD** emit events for significant interactions to provide an audit trail and facilitate monitoring of contract activities.

[D327] Developers **SHOULD** consider implementing fallback mechanisms to handle failures in cross-contract interactions gracefully.

4.4 Storage

4.4.1 Introduction

Storage in smart contracts refers to the mechanism by which contract data is persistently stored on the blockchain. Efficient storage management is crucial for the performance, cost-effectiveness, and scalability of smart contracts. Effective storage management is crucial for creating efficient, cost-effective, and scalable smart contracts. Developers should carefully consider storage strategies, balancing factors such as cost, performance, security, and scalability based on the specific requirements of their application.

4.4.2 Types of Storage

4.4.2.1 On-Chain Storage

On-Chain Storage refers to the method of storing data directly on the blockchain as part of the smart contract's state. This data is distributed across all nodes in the network, ensuring high levels of security, immutability, and transparency. Every piece of information stored on-chain is subject to consensus mechanisms and is permanently recorded in the blockchain's history. While on-chain storage provides the highest level of trust and data integrity, it comes with significant trade-offs. It is typically more expensive in terms of transaction costs (gas fees in systems like Ethereum), has limited capacity due to block size restrictions, and can lead to scalability issues as the blockchain grows. On-chain storage is best suited for critical data that requires the full security and transparency guarantees of the blockchain, such as ownership records, token balances, or crucial contract parameters. Developers should carefully consider the balance between the benefits of on-chain storage and its associated costs and limitations when designing smart contracts. Wang, S., Ouyang, L., Yuan, Y., Ni, X., Han, X., & Wang, F. Y. provide a comprehensive overview of smart contract architectures, including different storage types. The papers discuss the distinctions between on-chain and off-chain storage, their respective use cases, and the trade-offs involved [148].

4.4.2.2 Off-Chain Storage

Off-Chain Storage in smart contracts refers to the practice of storing data externally to the blockchain, with only references or cryptographic hashes of the data stored on-chain. This approach addresses the limitations of on-chain storage by offering lower costs, higher capacity, and improved scalability. Off-chain storage also addresses the issue of storing confidential data on a public ledger. Off-chain storage can utilize various external systems, such as distributed file systems (e.g. IPFS), traditional databases, or specialized off-chain storage solutions. While it provides greater flexibility and efficiency, especially for large datasets or frequently changing information, off-chain storage introduces additional complexity and potential security considerations. It requires careful design to maintain data integrity and availability, often involving mechanisms to verify the off-chain data's authenticity using the on-chain references [157].

4.4.2.3 Requirements and Recommendations

[R108] Smart contracts **SHALL** clearly distinguish between on-chain and off-chain storage mechanisms.

[D328] Developers **SHOULD** carefully consider the trade-offs between on-chain and off-chain storage based on the specific needs of their application.

[D329] Contracts **SHOULD** use on-chain storage for critical data that requires immediate availability and consensus.

[D330] Off-chain storage **SHOULD** be considered for large datasets or frequently changing information.

4.4.3 Storage Mechanisms

4.4.3.1 State Variables

State variables in smart contracts are permanent storage variables declared within the contract that persist across multiple function calls and transactions. These variables represent the contract's state and are stored directly on the blockchain, making them accessible to all contract functions. State variables can be of various data types, including primitives (like integers, booleans, addresses) and more complex types (such as structs, arrays, or mappings). They are typically used to store critical contract data, such as ownership information, token balances, or configuration parameters. Each state variable occupies one or more storage slots, depending on its size, and writing to these variables incurs gas costs. Efficient use of state variables is crucial for optimizing contract performance and minimizing transaction costs, as their values are maintained throughout the contract's lifetime and are subject to consensus mechanisms of the blockchain network.

4.4.3.2 Mappings

Mappings in smart contracts are key-value pair data structures that provide efficient storage and retrieval of data. They function similarly to hash tables, allowing for quick lookups based on a unique key. Mappings are particularly useful for associating addresses (keys) with balances (of currency, tokens or other assets), storing user data, or managing relationships between different entities within the contract. Unlike arrays, mappings do not have a length or concept of iteration over all elements, and they automatically initialize all possible keys with default values. This makes mappings gas-efficient for large sets of data where not all keys are used. However, developers should be mindful that mapping data is not iterable by default and requires additional structures if enumeration is needed. Mappings are a fundamental tool for organizing and accessing data in smart contracts, offering a balance between efficiency and functionality. Zhu, Y., Qin, Y., Zhou, Z., Song, X., Liu, G., & Chu, W. C. C. explore various storage mechanisms in the context of digital asset management on blockchain [175]. The papers discuss different data structures like mappings and arrays, which are key components of the storage mechanisms mentioned in this clause.

4.4.3.3 Arrays

Arrays in smart contracts are ordered collections of data elements of the same type, providing a way to store and manage multiple values under a single variable name. They can be fixed-size or dynamic, allowing for flexibility in data storage. Arrays are useful for maintaining lists, such as user addresses, transaction records, or any sequence of related data. Unlike mappings, arrays have a defined length and support iteration, making them suitable for scenarios requiring ordered data or when the need to loop through all elements is important. However, arrays can be less gas-efficient than mappings for large datasets, especially when elements are frequently added or removed, as these operations can be costly in terms of gas consumption. Developers should carefully consider the trade-offs between arrays and other data structures based on their specific use case, particularly in terms of gas costs and required functionality.

4.4.3.4 Structs

Structs in smart contracts are custom data types that allow developers to group related data elements together, creating more complex and organized data structures. They provide a way to encapsulate multiple variables of different types into a single unit, making it easier to manage and manipulate related information as a cohesive entity. Structs are particularly useful for representing complex objects or entities within a contract, such as user profiles, voting records, or financial instruments. They enhance code readability and maintainability by logically organizing related data fields. Structs can be used as standalone variables, as elements in arrays, or as values in mappings, offering flexibility in how data is structured and accessed within the contract. While structs provide powerful organizational capabilities, developers should be mindful of gas costs when working with large or nested struct data, especially in storage operations.

4.4.3.5 Requirements and Recommendations

[R109] Smart contracts **SHALL** implement appropriate storage mechanisms (state variables, mappings, arrays, structs) based on the data structure and access patterns required.

[D331] Developers **SHOULD** use mappings for key-value pairs with quick lookups.

[D332] Arrays **SHOULD** be used for ordered lists that require iteration.

[D333] Structs **SHOULD** be used for grouping related data.

[D334] Developers **SHOULD** consider gas costs when choosing between different storage mechanisms.

4.4.4 Storage Optimization Techniques

4.4.4.1 General discussion

A paper by Chen, T., Li, X., Luo, X., & Zhang, X. focuses on the optimization of smart contracts, including storage optimization techniques. It discusses the impact of poorly optimized storage on gas costs [31].

4.4.4.2 Data Packing

Data Packing in smart contracts refers to the technique of combining multiple smaller variables into a single storage slot to optimize storage usage and reduce gas costs. This method takes advantage of the fact that storage in blockchain systems like Ethereum is organized in 32-byte slots. By carefully arranging and packing multiple variables that collectively fit within 32 bytes into a single slot, developers can significantly reduce the number of storage operations and, consequently, the gas costs associated with storing and retrieving data. This technique is particularly effective for variables that are frequently used together or for storing multiple small-sized pieces of information efficiently.

4.4.4.3 Lazy Loading

Lazy Loading in smart contracts is a storage optimization technique that involves loading data only when it is needed, rather than pre-emptively storing or retrieving all data. This approach reduces unnecessary storage operations and associated gas costs. By deferring the loading of data until it is required for execution, lazy loading can significantly improve contract efficiency, especially when dealing with large datasets or complex data structures. This technique is particularly useful in scenarios where certain data may not be accessed in every transaction, allowing for more cost-effective and performant smart contract operations.

4.4.4.4 Deletion and Cleanup

Deletion and Cleanup in smart contracts refers to the process of managing obsolete or unnecessary data to optimize storage usage and reduce costs. However, this concept presents a challenge when considering PDL data immutability. True deletion is not possible on most DLT/PDL systems, as historical data remains part of the immutable ledger. Instead, smart contracts implement "logical deletion" where data is marked as deleted or archived but remains on the ledger. This approach allows applications to treat the data as if it were deleted while preserving the blockchain's integrity. Developers can implement mechanisms to identify and logically remove unnecessary data, potentially earning gas refunds in some DLT/PDL systems. Care should be taken to ensure that critical historical data is not lost and that the cleanup process itself is gas efficient. Balancing the need for data management with the principle of immutability requires careful design and consideration of the specific use case and regulatory requirements.

4.4.4.5 Requirements and Recommendations

Requirements:

[R110] Smart contracts **SHALL** implement storage optimization techniques to minimize gas costs and improve efficiency.

Recommendations:

[D335] Developers **SHOULD** use data packing techniques to combine multiple smaller variables into a single storage slot.

[D336] Lazy loading **SHOULD** be implemented for data that is not always needed.

[D337] Contracts **SHOULD** include mechanisms for data cleanup and deletion when information is no longer needed.

4.4.5 Cost Considerations

4.4.5.1 Gas Costs

Gas costs in smart contracts refer to the computational expenses associated with executing operations, particularly those involving storage. Understanding and optimizing these costs is crucial for efficient contract design. Storage operations, such as writing or modifying state variables, are among the most expensive in terms of gas consumption. Developers should carefully consider the frequency and necessity of storage operations, using techniques like data packing, lazy loading, and efficient data structures to minimize gas usage. Optimizing gas costs not only reduces transaction fees for users but also improves the overall efficiency and scalability of the contract. It is essential to balance the need for data persistence with the economic implications of frequent storage updates. This topic is discussed in greater depth in clause 10 herewith.

4.4.5.2 Storage Refunds

Storage Refunds in some blockchain systems refer to a mechanism that incentivizes the freeing up of storage space by offering a partial refund of gas costs. However, this concept needs to be understood within the context of blockchain data immutability. In blockchain systems, true deletion of data is not possible due to the immutable nature of the ledger. Instead, when a smart contract "frees up" storage:

- The data is not actually deleted from the blockchain's history.
- The storage slot is marked as available for future use, potentially by overwriting it with new data in a manner that does not break the consistency of the ledger (the topic of redaction is discussed in ETSI GR PDL 018 [i.20].
- The current state of the blockchain reflects this change, but the historical data remains accessible through past blocks.

When a transaction frees up storage in this manner, the system provides a refund to the transaction sender. This refund is calculated based on the amount of storage freed and is typically capped at a certain percentage of the total gas used in the transaction. The purpose of this mechanism is to encourage developers and users to efficiently manage contract storage, helping to control the growth of the blockchain's state size. However, it is important to note that:

- Storage refunds are subject to specific rules and limitations that can vary between different blockchain systems or protocol upgrades.
- The refund does not imply deletion of historical data, which remains part of the immutable blockchain.
- Developers should consider storage refunds as part of their overall gas optimization strategy, but should not rely on them as a primary means of cost reduction.
- When designing contracts that may need to "remove" data, developers should implement logical deletion patterns that mark data as inactive rather than attempting to truly delete it.

In summary, while storage refunds provide an economic incentive for efficient storage management, they operate within the constraints of blockchain immutability and should be approached with a clear understanding of their limitations and implications.

4.4.5.3 Requirements and Recommendations

[R111] Smart contract developers **SHALL** consider gas costs in their storage design and implementation.

[D338] Developers **SHOULD** optimize storage operations to minimize gas costs.

[D339] Contracts **SHOULD** implement storage refund mechanisms where appropriate to incentivize state cleanup.

4.4.6 Storage Data Security

4.4.6.1 Access Control

Implementing proper access controls for storage variables is a critical aspect of smart contract security and functionality [111]. Access control mechanisms ensure that only authorized entities can read from or write to specific storage variables within a contract. This is typically achieved using modifiers, function-level restrictions, and role-based access control patterns. Developers should carefully define and enforce permissions for each storage variable, considering who should be able to modify the data and under what circumstances. This may involve using ownership patterns, multi-signature schemes, or more complex governance models for critical variables. Proper access control helps prevent unauthorized modifications, protects sensitive data, and maintains the integrity of the contract's state. It is important to note that while access controls can restrict direct modifications to storage variables, the data itself remains visible on the blockchain due to its transparent nature. Therefore, access control should be complemented with other security measures, such as encryption or off-chain storage solutions, for truly sensitive information. Regular audits and testing of access control mechanisms are essential to ensure they continue to meet the contract's security requirements as the system evolves. The topic of smart contract security is covered in depth in clause 4.3.9 herewith.

A paper by Praitheeshan, P., Pan, L., Yu, J., Liu, J., & Doss, R. surveys [72] various security aspects of smart contracts, including those related to storage. It discusses access control and data integrity.

4.4.6.2 Data Integrity

Data Integrity in smart contracts refers to the measures and techniques implemented to ensure that stored data remains accurate, consistent, and unaltered throughout its lifecycle. This is crucial for maintaining the reliability and trustworthiness of the contract's operations. Key aspects of data integrity include input validation to prevent invalid or malicious data from being stored, access control to restrict unauthorized modifications, and the use of cryptographic techniques such as hashing to verify data has not been tampered with. Smart contracts should also implement checks and balances to maintain consistency across related data points, especially when multiple storage variables are interdependent. In some cases, using events to log state changes can provide an additional layer of verification. Developers should also consider the implications of contract upgrades on data integrity, ensuring that any changes to the contract's logic do not inadvertently compromise existing data. Regular audits and thorough testing of data manipulation functions are essential to maintain high standards of data integrity in smart contract systems.

4.4.6.3 Requirements and Recommendations

[R112] Smart contracts **SHALL** implement robust access control mechanisms for storage variables.

[R113] Contracts **SHALL** ensure data integrity for all stored information.

[D340] Developers **SHOULD** use role-based access control for managing storage permissions.

[D341] Contracts **SHOULD** implement checks and balances to maintain consistency across related data points.

[D342] Event logging **SHOULD** be used to track significant state changes.

4.4.7 Advanced Storage Patterns

4.4.7.1 Architectural Patterns

A paper by Xu, X., Weber, I., Staples, M., Zhu, L., Bosch, J., Bass, L., ... & Rimba, P. presents a taxonomy of blockchain-based systems, including advanced architectural patterns [156]. Some of these patterns relate to storage, such as off-chain storage solutions and data structures like Merkle trees.

4.4.7.2 Eternal Storage

Eternal Storage is an advanced storage pattern that separates the storage logic from the main contract logic, enhancing upgradeability and data persistence in smart contract systems. This pattern involves creating a dedicated storage contract that holds all state variables, while the main logic contract interacts with this storage contract through carefully designed interfaces. By decoupling storage from logic, developers can upgrade the contract's functionality without risking data loss or corruption. The storage contract remains constant, preserving the state, while new versions of the logic contract can be deployed and linked to the existing storage. This approach offers several advantages: it simplifies contract upgrades, reduces the risk of data loss during upgrades, and allows for more flexible contract evolution over time. However, implementing Eternal Storage requires careful design of the storage interface and access controls to ensure data integrity and security. Despite its complexity, this pattern is particularly valuable for long-lived contracts that may require frequent updates or those managing critical data that requires persistence across multiple contract versions.

4.4.7.3 Commit-Reveal Schemes

Commit-Reveal Schemes are advanced storage patterns used in smart contracts to manage sensitive data through a two-step process. In this approach, participants first submit a cryptographic hash (the "commit") of their data, followed by a later revelation of the actual data (the "reveal"). This pattern is particularly useful for scenarios requiring privacy or preventing front-running in competitive situations. The commit phase allows users to stake their claim or input without revealing the actual content, while the reveal phase verifies the original commitment and processes the revealed data. This technique ensures fairness and prevents manipulation in various applications such as blind auctions, voting systems, or random number generation. By separating the submission of data into two distinct steps, Commit-Reveal Schemes provide a layer of security and fairness, especially in time-sensitive or competitive smart contract interactions. However, implementing this pattern requires careful consideration of timing mechanisms and proper handling of both phases to ensure the integrity and effectiveness of the process.

4.4.7.4 Merkle Trees

Merkle Trees are advanced data structures used in smart contracts for efficiently managing and verifying large datasets. In this pattern, data is organized in a tree-like structure where each non-leaf node is a hash of its child nodes, and leaf nodes contain the actual data or its hash. This hierarchical hashing allows for efficient proof of inclusion or exclusion of specific data points without needing to store or transmit the entire dataset on-chain. Smart contracts can store only the root hash of the Merkle Tree on-chain, while keeping the bulk of the data off-chain. When specific data needs to be verified, only the relevant branch of the tree (known as a Merkle proof) needs to be provided, significantly reducing storage and computational costs. This makes Merkle Trees particularly useful for applications involving large datasets, such as token distributions, voting systems, or complex state management, where they enable scalable and gas-efficient operations while maintaining data integrity and verifiability. The use of Merkle Trees in smart contracts exemplifies a powerful synergy between off-chain data management and on-chain verification, addressing key challenges in blockchain scalability and efficiency.

4.4.7.5 Requirements and Recommendations

[R114] Smart contracts implementing advanced storage patterns **SHALL** ensure that these patterns do not introduce new vulnerabilities.

[D343] Developers **SHOULD** consider using eternal storage patterns for upgradeable contracts.

[D344] Commit-reveal schemes **SHOULD** be used for scenarios requiring privacy or preventing front-running.

[D345] Merkle trees **SHOULD** be considered for efficient verification of large datasets.

4.4.8 Challenges and Considerations

4.4.8.1 Scalability

The rapid growth and increasing complexity of blockchain networks and smart contract applications have exposed significant scalability challenges, particularly in the realm of data storage and management. As these systems expand, they face mounting difficulties in efficiently handling large volumes of data while maintaining performance, cost-effectiveness, and the core principles of blockchain technology.

Key issues include:

1) **Data Volume:** The exponential growth of on-chain data as smart contracts handles more complex operations and user bases expand, leading to potential network congestion and performance degradation.

2) **Storage Costs:** Increasing expenses associated with storing large amounts of data on-chain, which can make certain applications economically unfeasible.

3) **Network Performance:** Potential slowdowns in transaction processing and contract execution due to the burden of managing and accessing large datasets.

4) **Data Integrity:** Balancing the need for data availability and verification with the practical limitations of on-chain storage.

5) **Decentralization:** Maintaining the decentralized nature of blockchain systems while implementing scalable storage solutions.

6) **Architectural Complexity:** Designing smart contracts and blockchain systems that can efficiently handle growing data requirements without compromising security or functionality.

7) **Cross-chain Compatibility:** Ensuring that scalability solutions are compatible across different blockchain platforms and can support interoperability.

8) **Regulatory Compliance:** Adhering to data protection and privacy regulations while implementing scalable storage solutions.

Addressing these challenges requires innovative approaches that balance on-chain and off-chain storage, optimize data structures, and leverage emerging technologies like sharding and layer-2 solutions. The development of effective scalability solutions is crucial for the continued growth and adoption of blockchain technology, enabling smart contracts to handle increasing data loads without sacrificing performance, security, or decentralization. Successful strategies should not only solve current scalability issues but also anticipate future growth and technological advancements in the blockchain ecosystem [154], [103], [158], [19], [108] and [14].

Scalability can be addressed through architectural and functional modelling as described herewith.

1) **Layered Architecture:** Implementing a layered architecture can separate concerns and distribute workloads more effectively. This approach allows for the integration of Layer-2 solutions, such as state channels and rollups, which can handle transactions off-chain and only settle on-chain when necessary, significantly improving scalability.

2) **Sharding:** Sharding divides the blockchain into smaller, manageable pieces called shards, each capable of processing transactions independently. This architectural approach enhances scalability by allowing parallel processing of transactions across different shards.

3) **Modular Design:** A modular architecture enables components to be developed, tested, and deployed independently. This flexibility can improve scalability by allowing specific modules to be optimized or scaled without affecting the entire system.

4) **Efficient State Management:** Implementing efficient state management techniques, such as using Merkle trees or other data structures that optimize storage and retrieval processes, can enhance scalability by reducing the computational load on the network.

5) **Optimized Consensus Mechanisms:** Adopting consensus mechanisms that require less computational power and time, such as Proof-of-Stake (PoS) or Delegated Proof-of-Stake (DPoS), can improve transaction throughput and scalability.

6) **Interoperability Protocols:** Ensuring interoperability with other blockchain networks allows for workload distribution across multiple chains. This can alleviate congestion on a single network and improve overall scalability.

7) **Implement Adaptive Scaling Techniques:** Adaptive scaling methods can dynamically adjust resources based on current demand to maintain performance levels. This is useful to ensuring proper operation of a smart contract under changing loads.

8) **Focus on Gas Optimization:** Continuously monitoring smart contract performance and updating scalability solutions as needed helps accommodating growing transaction volumes and evolving network conditions.

By addressing these architectural and functional requirements, smart contract platforms can achieve better scalability, ensuring they can handle increased demand while maintaining performance and reliability.

Additional examples of innovative approaches that address such scalability issues are listed in clause A.1.1.4.

4.4.8.2 Privacy

Managing sensitive data while maintaining transparency is a significant challenge in smart contract development, given the inherently public nature of blockchain systems. While transparency is a key feature of blockchain technology, it can conflict with the need to protect confidential information. Developers should employ various techniques to balance these competing requirements. These may include on-chain encryption, where sensitive data is encrypted before being stored on the blockchain; off-chain storage solutions, where only hashes or references to sensitive data are stored on-chain; zero-knowledge proofs [84], allowing verification of information without revealing the underlying data; and secure multi-party computation protocols [167]. It is crucial to carefully consider what data absolutely needs to be on-chain and what can be managed off-chain. Additionally, developers should be aware of regulatory requirements like GDPR and design their data management strategies accordingly [140]. As the field evolves, new privacy-preserving technologies are emerging, offering more sophisticated solutions to this challenge [43].

4.4.8.3 Long-Term Storage

Long-term storage in smart contracts presents unique challenges due to the ever-growing nature of blockchain data and the potential for technological obsolescence. As blockchain networks expand, concerns arise about data accessibility, storage costs, and the environmental impact of maintaining vast amounts of data indefinitely. Recent research has explored various solutions, including data compression techniques to reduce on-chain storage requirements, hybrid storage models that combine on-chain and off-chain data management, and novel consensus mechanisms that allow for more efficient data pruning without compromising security. Additionally, there is growing interest in blockchain interoperability and data migration strategies to ensure long-term data accessibility as technologies evolve. These approaches aim to balance the need for data persistence with practical considerations of scalability and sustainability. However, developers have to carefully consider the long-term implications of their data storage decisions, including potential regulatory requirements for data retention and the challenges of managing access to historical data over extended periods.

4.4.8.4 Requirements and recommendations

[R115] The smart contract architecture **SHALL** be designed to scale efficiently as the PDL network grows.

[R116] Contracts handling sensitive data **SHALL** implement appropriate privacy measures.

[D346] Smart contract developers **SHOULD** utilize data structures like Merkle trees for efficient storage and retrieval processes, minimizing computational load on the network.

[D347] Developers **SHOULD** consider layer-2 solutions or sharding for scalability.

[D348] Privacy-preserving techniques like zero-knowledge proofs **SHOULD** be used when dealing with sensitive data.

[D349] Contracts **SHOULD** implement mechanisms to handle potential state inconsistencies in long-term storage.

[D350] Smart contract developers **SHOULD** employ consensus mechanisms that reduce computational power requirements, such as Proof-of-Stake (PoS) or Delegated Proof-of-Stake (DPoS), to improve transaction throughput without compromising security.

[D351] Smart contract developers **SHOULD** use adaptive scaling methods that dynamically adjust resources based on current demand to maintain performance levels.

[D352] Smart contract developers **SHOULD** optimize smart contract code to reduce gas consumption, which can lead to more efficient use of network resources and improved scalability.

[D353] Sharding-friendly data structures and logic **SHOULD** be implemented to support network scalability.

4.4.9 Future Trends

4.4.9.1 Decentralized Storage Solutions

Decentralized storage solutions are emerging as a crucial complement to blockchain-based smart contracts, addressing scalability and cost issues associated with on-chain data storage [147], [104], [154], [73], [168]. Integration with decentralized storage networks like InterPlanetary File System (IPFS) and Filecoin offers smart contracts the ability to store large amounts of data off-chain while maintaining a cryptographic link to the data on-chain. This approach significantly reduces on-chain storage costs and improves scalability. Recent advancements include the development of hybrid storage models that combine the immutability of blockchain with the scalability of decentralized storage, enhanced data retrieval mechanisms, and improved incentive structures for storage providers. Researchers are also exploring novel consensus mechanisms tailored for decentralized storage networks to ensure data availability and integrity. However, challenges remain in areas such as data privacy, access control, and long-term data persistence. As these technologies mature, they promise to enable more complex and data-intensive smart contract applications while maintaining the decentralized ethos of blockchain systems. Macrinici, D., Cartofeanu, C., & Gao, S. have conducted a systematic mapping study that provides an overview of smart contract applications and discusses emerging trends in the field. It includes discussion on future developments in storage solutions for smart contract [95].

[D354] Developers **SHOULD** stay informed about emerging storage solutions like decentralized storage networks.

4.4.9.2 Layer-2 Storage Solutions

Layer-2 storage solutions are emerging as a promising approach to address the scalability and cost limitations of on-chain data storage while leveraging the security of the main blockchain. These solutions involve storing data off-chain but anchoring cryptographic proofs or commitments on the main chain, thus inheriting its security guarantees. Recent research has explored various Layer-2 storage architectures, including state channels, sidechains, and rollups (both optimistic and zero-knowledge) [78], [54], [154], [95]. These approaches significantly reduce on-chain storage requirements and transaction costs while maintaining data integrity and availability. Advancements in zero-knowledge proof systems have enabled more efficient and privacy-preserving off-chain storage solutions. Researchers are also investigating hybrid models that combine different Layer-2 technologies to optimize specific use cases. However, challenges remain in areas such as data availability, cross-layer communication, and user experience. As these technologies mature, they promise to enable more data-intensive and complex smart contract applications while maintaining the security and decentralization benefits of the underlying blockchain.

[D355] Smart contract designers **SHOULD** consider the potential of layer-2 storage solutions in their long-term planning.

4.4.10 Best Practices

4.4.10.1 General Discussion

Smart contract developers should adhere to the following best practices for efficient and secure storage management:

1) **Minimize On-Chain Storage:** Reduce on-chain storage to optimize costs and enhance performance. Store only essential data on-chain, using off-chain solutions for large datasets or frequently changing information. Implement techniques such as data compression or encoding to minimize storage requirements [30].

2) **Efficient Data Structures:** Select appropriate data structures for optimal storage and retrieval. Use mappings for key-value pairs with quick lookups, arrays for ordered lists that require iteration, and structs for grouping related data. Consider gas costs when choosing between different data structures, especially for large datasets [93].

3) **Access Controls and Validation:** Implement robust access controls to restrict who can modify storage variables. Use modifiers or dedicated functions to enforce permissions. Validate all input data before storage to ensure data integrity and prevent malicious inputs [175].

4) **Regular Audits and Optimization:** Conduct periodic audits of storage usage to identify inefficiencies or unused data. Implement cleanup mechanisms to remove or archive obsolete data. Optimize storage layouts to take advantage of storage slots and reduce gas costs [72].

5) **Long-Term Considerations:** Evaluate the long-term implications of storage decisions. Consider data migration strategies for potential contract upgrades. Plan for scalability by designing storage structures that can accommodate future growth. Be mindful of the immutable nature of blockchain data when deciding what to store on-chain [14].

By following these best practices, developers can create more efficient, cost-effective, and secure smart contracts with optimized storage management.

4.4.10.2 Requirements and Recommendations

[R117] Smart contracts **SHALL** implement storage optimization techniques to minimize gas costs and improve efficiency.

[R118] Developers **SHALL** use appropriate data structures for efficient storage and retrieval based on the specific use case.

[R119] Smart contracts **SHALL** implement robust access control mechanisms for all storage operations.

[R120] Smart Contracts **SHALL** include comprehensive event logging for all significant state changes and important function calls.

[R121] Smart contract systems **SHALL** implement mechanisms for handling potential differences in transaction speed and cost across interacting systems.

[D356] Developers **SHOULD** minimize on-chain storage to optimize costs and enhance performance.

[D357] Contracts **SHOULD** use mappings for key-value pairs with quick lookups, arrays for ordered lists that require iteration, and structs for grouping related data.

[D358] Smart contracts **SHOULD** implement Role-Based Access Control (RBAC) for fine-grained permissions management.

[D359] Developers **SHOULD** conduct periodic audits of storage usage to identify inefficiencies or unused data.

[D360] Contracts **SHOULD** implement mechanisms to handle potential state rollbacks due to blockchain reorganizations.

[D361] Developers **SHOULD** use constant and immutable variables where appropriate to optimize gas costs and improve code readability.

[D362] Smart contracts **SHOULD** implement a clear separation between mutable and immutable state to enhance security and auditability.

[D363] Developers **SHOULD** consider implementing state checkpoints for complex contracts to allow for easier debugging and state verification.

4.5 Modern Smart Contract Platforms and Languages

4.5.1 Introduction

The landscape of smart contract development has evolved significantly, with various platforms and languages emerging to address different needs and use cases. This clause provides an overview of some prominent smart contract platforms and languages as of 2024.

4.5.2 Ethereum and Solidity

Ethereum remains one of the most widely used platforms for smart contract development, with Solidity as its primary programming language. Recent developments have significantly enhanced the capabilities and security of smart contracts on this platform.

Key features:

1) Object-oriented, high-level language
2) Statically typed
3) Supports inheritance, libraries, and complex user-defined types
4) Large developer community and extensive tooling

Recent advancements:

1) **Improved Security Features:** Solidity 0.8.0 (released in December 2020) introduced built-in overflow checks for arithmetic operations, significantly reducing the risk of integer overflow vulnerabilities [7].
2) **Gas Optimization:** Recent versions of Solidity have focused on gas optimization, introducing features like tight variable packing and more efficient storage layouts [25].
3) **Enhanced Type System:** The introduction of user-defined value types and custom errors in Solidity 0.8.0 has improved code readability and error handling [177].
4) **Ethereum 2.0 Compatibility:** With the ongoing transition to Ethereum 2.0, Solidity has been adapting to support new features like sharding and proof-of-stake consensus [153].
5) **Formal Verification:** There is an increasing focus on formal verification tools specifically designed for Solidity, enhancing smart contract reliability [53].

These advancements have further solidified Ethereum and Solidity's position in the smart contract ecosystem, addressing previous limitations and preparing for future scalability challenges.

4.5.3 Polkadot and Ink

Polkadot has emerged as a significant player in the blockchain ecosystem, offering a unique approach to interoperability and scalability. Ink, Polkadot's smart contract language, has been gaining traction as a robust alternative for developers.

Key features:

1) Based on Rust, providing strong safety guarantees
2) Designed for cross-chain compatibility
3) Supports WebAssembly (Wasm) for efficient execution
4) Focuses on interoperability within the Polkadot ecosystem

Recent advancements:

1) **Substrate Integration:** Ink has been closely integrated with Substrate, Polkadot's blockchain development framework, allowing for seamless deployment of smart contracts across parachains [161].
2) **Cross-Chain Functionality:** Recent updates to Ink have enhanced its ability to facilitate cross-chain operations, leveraging Polkadot's unique architecture [156].
3) **Performance Optimizations:** The Ink! 3.0 release in 2021 introduced significant performance improvements, reducing contract size and enhancing execution speed [109].
4) **Enhanced Developer Tools:** The ecosystem has seen the development of new testing frameworks and IDEs specifically designed for Ink, improving the developer experience [42].
5) **Security Enhancements:** Recent research has focused on formal verification methods for Ink contracts, aiming to provide stronger security guarantees [52].

These developments have positioned Polkadot and Ink as strong contenders in the smart contract space, offering a unique value proposition centred around interoperability and scalability.

4.5.4 Cardano and Plutus

Cardano has become a prominent blockchain platform, with Plutus as its smart contract language. Recent developments have significantly enhanced Cardano's smart contract capabilities, positioning it as a strong competitor in the blockchain space.

Key features:

1) Functional programming paradigm based on Haskell
2) Strong type system for enhanced security
3) Built-in support for formal verification
4) Designed for high assurance applications

Recent advancements:

1) **Plutus Platform Launch:** The Alonzo hard fork in September 2021 marked the official introduction of smart contract functionality on Cardano, enabling the deployment of Plutus scripts on the mainnet [24].
2) **Extended UTXO Model:** Cardano implements an extended UTXO model, which combines the benefits of Bitcoin's UTXO model with Ethereum's account-based model, offering unique advantages for smart contract design [26].
3) **Plutus Application Backend (PAB):** The introduction of PAB has simplified the process of building and deploying Decentralized Applications (DApps) on Cardano [77].
4) **Formal Verification Enhancements:** Recent research has focused on improving formal verification techniques for Plutus contracts, leveraging its functional programming roots [i.31].
5) **Scalability Solutions:** Ongoing development of layer-2 solutions, such as Hydra, aims to enhance the scalability of Plutus smart contracts [27].

These developments have significantly boosted Cardano's capabilities in the smart contract domain, offering a unique approach that emphasizes security and formal verification.

4.5.5 Algorand and TEAL/PyTeal

Algorand has emerged as a notable player in the blockchain space, offering a unique approach to smart contract development with its Transaction Execution Approval Language (TEAL) and the higher-level PyTeal.

Key features:

1) Low-level, stack-based language (TEAL)
2) Higher-level abstractions available through PyTeal
3) Designed for efficiency and security
4) Focus on fast finality and scalability

Recent advancements:

1) **AVM 1.0 and TEAL 3.0:** The Algorand Virtual Machine (AVM) 1.0 release in 2020 introduced significant enhancements to TEAL, including loops and subroutines, greatly expanding its capabilities [5].
2) **Stateful Smart Contracts:** Algorand implemented stateful smart contracts in 2020, allowing for more complex and persistent applications [29].
3) **PyTeal Improvements:** Recent updates to PyTeal have introduced more intuitive syntax and additional high-level constructs, making it easier for developers to write complex smart contracts [4].
4) **Atomic Transfers:** Algorand's atomic transfers feature has been leveraged to create more sophisticated smart contract interactions, enabling complex multi-party transactions [155].

5) **AVM 1.1 and TEAL 5.0:** The latest updates have introduced new opcodes and increased the number of available scratch spaces, further enhancing TEAL's capabilities [6].

These developments have significantly expanded Algorand's smart contract functionality, positioning it as a compelling platform for developers seeking efficiency and security.

4.5.6 Cosmos and CosmWasm

The Cosmos ecosystem has gained significant traction in recent years, with CosmWasm emerging as its primary smart contract platform. CosmWasm allows smart contracts written in Rust to be compiled to WebAssembly (Wasm), offering a unique approach to blockchain interoperability and scalability.

Key features:

1) Interoperability across Cosmos-based chains
2) Rust's safety features
3) Efficient execution through WebAssembly
4) Designed for multi-chain environments

Recent advancements:

1) **CosmWasm 1.0:** The release of CosmWasm 1.0 in 2021 marked a significant milestone, introducing stability and new features for production use [34].
2) **Inter-Blockchain Communication (IBC) Integration:** CosmWasm has been integrated with the IBC protocol, allowing smart contracts to communicate across different blockchain networks within the Cosmos ecosystem [85].
3) **Multichain Contracts:** Recent developments have enabled the deployment of a single smart contract across multiple Cosmos chains, enhancing interoperability and reducing development complexity [1].
4) **CosmJS Integration:** Improvements in CosmJS have made it easier to interact with CosmWasm contracts from JavaScript applications, broadening the developer base [33].
5) **Security Enhancements:** Recent research has focused on formal verification methods for CosmWasm [33] smart contracts, leveraging Rust's strong type system [125].

These advancements have positioned CosmWasm as a powerful tool for building interoperable and scalable decentralized applications within the Cosmos ecosystem.

4.5.7 Tezos and Michelson/LIGO

Tezos has established itself as a unique blockchain platform with a focus on formal verification and self-amendment. Its smart contract ecosystem, centred around Michelson and the high-level language LIGO, has seen significant developments in recent years.

Key features:

1) **Michelson:** Stack-based, strongly-typed language
2) **LIGO:** High-level language compiling to Michelson
3) Formal verification capabilities
4) Self-amending protocol

Recent advancements:

1) **LIGO Evolution:** LIGO has undergone significant improvements, introducing new syntax options (CameLIGO, ReasonLIGO) and enhanced tooling, making it more accessible to developers from various programming backgrounds [139].

2) **Optimistic Rollups:** Tezos has implemented optimistic rollups, allowing for more scalable and efficient smart contract execution [115].

3) **Michelson Improvements:** Recent protocol upgrades have introduced new Michelson instructions and optimizations, enhancing the expressiveness and efficiency of smart contracts [138].

4) **Smart Contract Optimization:** Research has focused on gas optimization techniques specific to Tezos smart contracts, improving their cost-effectiveness [16].

5) **Formal Verification Advancements:** New tools and methodologies for formal verification of Michelson contracts have been developed, further enhancing Tezos' focus on contract correctness [118].

These developments have strengthened Tezos' position as a platform for secure and verifiable smart contracts, particularly in domains requiring high assurance.

These advancements highlight Tezos' continued focus on providing a robust, formally verifiable smart contract platform, with improvements in both low-level (Michelson) and high-level (LIGO) programming options. The platform's emphasis on security, verifiability, and self-amendment positions it uniquely in the evolving landscape of blockchain technologies.

4.5.8 Emerging Trends

Several trends are shaping the evolution of smart contract platforms and languages.

- **Interoperability**: Increasing focus on cross-chain compatibility and communication.
- **Scalability**: Development of layer-2 solutions and more efficient consensus mechanisms.
- **Security**: Greater emphasis on formal verification and auditable code.
- **Accessibility**: Creation of more user-friendly, high-level languages and development tools.
- **Specialization**: Platforms tailored for specific use cases (e.g. DeFi, NFTs, enterprise applications).

Here are some papers and reports that discuss those trends:

1) **Interoperability:** Zhang et al. (2022) have issued a report that highlights the importance of cross-chain bridges and smart contract interoperability for enabling asset transfer across diverse blockchain networks [166].

2) **Scalability:** Rapid Innovation (2023) hosted an article that discusses Layer-2 solutions as a means to improve scalability by reducing transaction costs and enhancing the performance of decentralized applications [116].

3) **Security:** Zhou et al. published a paper that examines vulnerabilities in blockchain systems and discusses current remedies to these security challenges [171].

4) **Accessibility:** Rapid Innovation (2023 explores how low-code and no-code platforms are making smart contracts more accessible to a broader audience, including those without technical expertise [117].

5) **Specialization:** Tatum Blog (n.d) posted a blog that explains how different smart contract platforms like Ethereum, Cardano, and Hyperledger Fabric offer specialized features tailored to specific use cases, such as enterprise applications or secure financial transactions [130].

As the field continues to evolve, developers should stay informed about the latest advancements and choose platforms and languages that best suit their project requirements, considering factors such as security, scalability, ease of use, and ecosystem support.

5 Smart Contracts - Lifecycle phases

5.1 Introduction

The lifecycle of a smart contract encompasses several distinct phases from conception to retirement. Understanding this lifecycle is crucial for effective development, deployment, and management of smart contracts. Throughout these stages, it is crucial to maintain transparency, adhere to best practices, and prioritize security to ensure the contract's integrity and effectiveness throughout its lifecycle. The specific implementation of each phase may vary depending on the blockchain platform, the complexity of the contract, and the governance model chosen for the project.

The main stages, discussed in clause 5.2 are Planning, Development and Testing, Deployment and Execution, Maintenance and Upgrade, Retirement and Deprecation. Clause 5.3 discusses Governance and Upgrade Models.

5.2 Planning Phase

5.2.1 Description and recent research

The planning phase is a critical first step in the smart contract lifecycle, laying the foundation for a successful and secure implementation.

The planning phase should involve collaboration between business stakeholders, legal experts, and technical teams to ensure a comprehensive and well-rounded approach. Thorough planning can significantly reduce risks, improve efficiency, and enhance the overall quality of the smart contract implementation.

Recent research emphasizes the importance of this phase in mitigating risks and ensuring contract success. For instance, a study by Zheng et al. (2021) highlights the critical role of comprehensive planning in reducing vulnerabilities in smart contracts [169]. Additionally, work by Li et al. (2020) underscores the importance of stakeholder analysis and governance considerations in the early stages of smart contract development [90].

This phase involves several key activities listed in the clauses below.

5.2.2 Defining the contract's purpose and requirements

The planning phase of a smart contract's lifecycle begins with a clear definition of its purpose and requirements. This foundational step ensures that the contract is aligned with business objectives and functions as intended.

- **Clearly articulate the contract's objectives and functionalities**: Establish a comprehensive understanding of what the smart contract is meant to achieve, including its primary goals and the specific tasks it will automate or facilitate.
- **Identify specific business rules and logic to be encoded**: Determine the essential business rules that are to be embedded within the contract. These rules dictate how the contract will operate under various conditions.
- **Define expected inputs, outputs, and interactions**: Specify the data that will be input into the contract, the outputs it will generate, and how it will interact with other systems or contracts. This clarity helps in designing robust and efficient workflows.

5.2.3 Identifying Stakeholders and Their Interactions

Identifying stakeholders and understanding their interactions with the smart contract is crucial for ensuring that all parties' needs are met and that the contract operates smoothly within its ecosystem.

- **Map out all parties involved in the contract's ecosystem**: Identify all stakeholders, including users, administrators, and external entities that will interact with the contract.
- **Define roles, responsibilities, and access rights for each stakeholder**: Clearly delineate what each stakeholder can do within the system. This includes setting permissions and access levels to ensure security and compliance.

- **Outline the flow of interactions between stakeholders and the contract**: Create a detailed map of how stakeholders will interact with the contract, including transaction flows and decision-making processes.

5.2.4 Outlining the Contract's Logic and State Variables

Designing the logic and state variables of a smart contract is a critical step in ensuring that it functions correctly and efficiently stores necessary data.

- **Design the contract's core functions and workflows**: Develop a blueprint for how the contract will execute its tasks, focusing on creating efficient and logical workflows.
- **Identify and define state variables to store contract data**: Determine which pieces of data need to be stored persistently within the contract, ensuring they are accessible for future transactions.
- **Consider data types, structures, and storage optimization techniques**: Choose appropriate data types and structures to optimize storage usage and performance, taking into account factors like gas costs on blockchain platforms.

5.2.5 Considering Security, Scalability, and Interoperability Needs

Addressing security, scalability, and interoperability during the planning phase helps mitigate risks and ensures that the smart contract can grow with future demands.

- **Conduct a preliminary risk assessment to identify potential vulnerabilities**: Evaluate potential security threats early in the process to design mitigation strategies that protect against exploits.
- **Evaluate scalability requirements and potential solutions (e.g. Layer-2, sharding)**: Assess how the contract can handle increased loads or transactions over time, considering solutions like Layer-2 protocols or sharding to enhance performance.
- **Assess interoperability needs with other contracts or external systems**: Determine how the contract will interact with other systems or contracts, ensuring compatibility through standards or protocols.

5.2.6 Evaluating Governance and Upgrade Models

Planning for governance and upgrades ensures that a smart contract remains adaptable to changing requirements or improvements without compromising its integrity.

- **Determine the governance structure for contract management**: Establish who will have control over making decisions about changes or updates to the contract, including voting mechanisms or administrative controls.
- **Consider upgrade strategies (e.g. proxy patterns, modular design)**: Plan for future upgrades by implementing design patterns that allow for modifications without disrupting existing functionalities.
- **Plan for potential future modifications and their implications**: Anticipate possible changes in business needs or technological advancements that may necessitate updates to the contract, ensuring these can be implemented smoothly.

5.3 Development and Testing Phase

5.3.1 Description and recent research

The development and testing phase is crucial in transforming the planned smart contract into a functional, secure, and efficient piece of code. Recent research emphasizes the importance of rigorous development and testing practices in smart contract creation. For instance, a study by Zou et al. (2021) highlights the effectiveness of combining static and dynamic analysis techniques in identifying smart contract vulnerabilities [177]. Additionally, work by Gao et al. (2020) demonstrates the value of formal verification methods in enhancing smart contract reliability [48].

The development and testing phase should be iterative, with continuous refinement based on test results and audit findings. This approach helps ensure the production of high-quality, secure smart contracts that meet the intended requirements and can withstand potential attacks or unexpected scenarios [149], [28].

This phase encompasses several critical activities described in the following clauses.

5.3.2 Writing the contract code in a suitable language

To properly fulfil this task a developer should follow the steps outlined herewith:

- Select an appropriate language based on the target platform (e.g. Solidity for Ethereum, Rust for Solana)
- Implement the contract logic following best coding practices and design patterns
- Ensure code readability and maintainability through proper documentation and structuring

5.3.3 Implementing security best practices and optimizations

Best practices should be used when developing a testing smart contracts as described herewith:

- Apply secure coding techniques to prevent common vulnerabilities (e.g. reentrancy, overflow/underflow)
- Optimize gas usage and storage efficiency
- Implement access control mechanisms and input validation

5.3.4 Conducting thorough testing

5.3.4.1 Introduction

Testing smart contracts is a critical phase in the development lifecycle, ensuring that the contracts function correctly, securely, and efficiently. Given the immutable nature of blockchain deployments, rigorous testing helps prevent costly errors and vulnerabilities. This clause outlines various strategies and tools for effectively testing smart contracts. By employing these comprehensive testing strategies and tools, developers can enhance the reliability, security, and efficiency of their smart contracts while minimizing risks associated with deployment on blockchain networks.

The testing phase should cover the following aspects:

- Develop comprehensive unit tests for individual functions
- Perform integration tests to verify interactions between contract components
- Conduct scenario-based testing to simulate real-world use cases
- Utilize automated testing tools and frameworks specific to smart contracts

5.3.4.2 Testing Strategies

Testing strategies for smart contracts encompass a range of methodologies to ensure comprehensive evaluation of contract functionality and security. These strategies include:

- **Unit Testing**: Focuses on individual functions within the contract to ensure they work as intended. Tools like Truffle or Hardhat (see notes below) can automate unit tests, providing immediate feedback on code changes.
- **Integration Testing**: Evaluates interactions between different components or contracts to ensure they work together correctly. This is crucial for complex systems where multiple contracts interact.
- **End-to-End Testing**: Simulates real-world scenarios to verify that the entire system functions as expected from start to finish. This type of testing often involves deploying the contract on a testnet and interacting with it as a user would.

- **Stress Testing**: Assesses how contracts perform under high-load conditions to identify potential bottlenecks or failures. Stress testing helps ensure that the contract can handle large volumes of transactions without degrading performance.

These strategies align with the key aspects outlined in clause 12.1 (Introduction), focusing on functionality, security, gas efficiency, and compliance with business logic [31], [148].

NOTE 1: **Truffle** is an open-source development environment, testing framework, and asset pipeline for Ethereum. It provides a suite of tools that make it easier to develop, test, and deploy smart contracts.

NOTE 2: **Hardhat** is a flexible, extensible development environment and task runner for building, testing, and deploying Ethereum smart contracts. It is designed to help developers manage and automate the recurring tasks inherent to blockchain projects.

Requirements:

[R122] Smart contracts **SHALL** undergo unit, integration, and end-to-end testing before deployment.

[R123] Stress testing **SHALL** be conducted to evaluate contract performance under load.

Recommendations:

[D364] Developers **SHOULD** use automated testing frameworks to streamline the testing process.

[D365] Regular updates to test cases **SHOULD** be made to cover new features or changes in contract logic.

5.3.4.3 Generalized Testing Targets

Generalized testing targets refer to specific aspects of a smart contract that should be evaluated during the testing process:

- **Functionality**: Ensures that each function performs its intended task correctly. For example, testing a token contract's transfer function to ensure it accurately updates balances.
- **Security**: Identifies vulnerabilities such as reentrancy attacks or improper access controls. For instance, verifying that only authorized users can execute administrative functions within a contract.
- **Gas Efficiency**: Evaluates whether the contract uses computational resources optimally to minimize transaction costs. This might involve analysing gas consumption patterns in commonly used functions to identify potential optimizations.
- **Compliance with Business Logic**: Verifies that the contract adheres to specified business rules and requirements. An example would be ensuring that a loan contract correctly enforces interest calculations and repayment schedules according to agreed terms.

Requirements:

[R124] Smart contracts **SHALL** be tested for functionality, security vulnerabilities, and gas efficiency.

[R125] Compliance with specified business logic **SHALL** be verified through tests.

Recommendations:

[D366] Developers **SHOULD** prioritize tests based on risk assessment to focus on critical areas first.

[D367] Test coverage reports **SHOULD** be generated to ensure all targets are adequately addressed.

5.3.4.4 Testing Checklist

A testing checklist serves as a comprehensive guide for developers to ensure all necessary aspects of smart contract testing are covered. This checklist includes verifying input validation, access control mechanisms, event emissions, error handling, and edge cases. By following a structured checklist, developers can systematically address potential issues and improve contract reliability.

- **Input Validation**: Ensure that all inputs to the smart contract are validated to prevent invalid or malicious data from causing unexpected behaviour. For example, a token transfer function should validate that the sender has sufficient balance before proceeding.
- **Access Control Mechanisms**: Verify that access control is properly implemented to restrict sensitive functions to authorized users only. For instance, administrative functions should only be callable by the contract owner or designated roles.
- **Event Emissions**: Check that events are emitted for significant state changes or actions within the contract to provide an audit trail. For example, emitting an event when tokens are transferred can help track transactions.
- **Error Handling**: Ensure that the contract handles errors gracefully and reverts transactions when necessary to maintain state integrity. This includes using require statements to enforce preconditions and revert on failure.
- **Edge Cases**: Test edge cases such as boundary conditions and unexpected inputs to ensure the contract behaves correctly under all scenarios. This might include testing with maximum integer values or zero balances.

By adhering to this checklist, developers can enhance the robustness and security of their smart contracts, reducing the risk of vulnerabilities and ensuring reliable operation.

Requirements:

[R126] Smart contracts **SHALL** have a detailed testing checklist covering all critical areas.

[R127] Input validation and access control **SHALL** be key components of the checklist.

Recommendations:

[D368] Developers **SHOULD** regularly update the checklist to incorporate lessons learned from previous projects.

[D369] Peer reviews of the checklist **SHOULD** be conducted to ensure completeness and accuracy.

5.3.4.5 Offline Testing

Offline testing involves simulating smart contract execution in a controlled environment without deploying it on the blockchain. This allows developers to identify bugs and optimize performance without incurring gas costs or affecting live data. Offline testing is crucial for verifying contract logic, ensuring security, and optimizing gas usage before deployment on a testnet or mainnet. Tools like Ganache (see note below) provide local blockchain environments that mimic Ethereum's mainnet conditions, allowing developers to execute transactions and test contract interactions in isolation.

NOTE: **Ganache** is a personal blockchain for Ethereum development that allows developers to create a local Ethereum network for testing smart contracts. Ganache enables developers to test their smart contracts in a controlled environment without spending real ETH or waiting for transactions to be mined on a public network. This makes it an essential tool for rapid development and testing of Ethereum applications.

Examples of Offline Testing methods:

- **Simulated Blockchain Environment**: Tools like Ganache create a personal blockchain that allows developers to test contracts in an environment that simulates the Ethereum network. This setup enables developers to perform various tests without the risk of affecting real assets or incurring costs.
- **Transaction Simulation**: Offline testing tools can simulate transactions to verify how a contract handles different inputs and scenarios. For example, developers can test edge cases such as maximum integer values or invalid input data to ensure robust error handling.

- **Performance Optimization**: By analysing gas consumption during offline testing, developers can identify inefficient code paths and optimize them for better performance. This step is essential for reducing transaction costs once the contract is deployed on the mainnet.

Requirements:

[R128] Smart contracts **SHALL** undergo offline testing before any deployment on testnets or mainnets.

[R129] Offline environments **SHALL** mimic mainnet conditions as closely as possible.

Recommendations:

[D370] Developers **SHOULD** use tools like Ganache or Hardhat for efficient offline testing.

[D371] Offline test results **SHOULD** be documented and analysed to guide further development.

5.3.4.6 Online Monitoring

Online monitoring involves tracking smart contract performance and behaviour after deployment on a live network. This continuous oversight is essential for identifying anomalies, security breaches, or unexpected behaviour through logging and alert systems. By maintaining real-time surveillance, developers can ensure the integrity of smart contracts and quickly address any issues that arise post-deployment.

Examples of Aspects:

- **Anomaly Detection**: Online monitoring tools can detect unusual patterns in transaction activity, such as sudden spikes in transaction volume or unexpected changes in contract state. For instance, using platforms like Tenderly or Etherscan, developers can set up alerts for transactions that deviate from expected norms.
- **Security Breach Alerts**: Monitoring systems can provide immediate notifications if a potential security breach is detected. This includes unauthorized access attempts or suspicious interactions with the contract. Tools like Forta Network can help identify and alert developers to these threats in real-time.
- **Performance Metrics**: Tracking performance metrics such as gas usage, transaction throughput, and response times helps ensure that the contract operates efficiently. Developers can use these insights to optimize contract performance and reduce costs.

Requirements:

[R130] Smart contracts deployed on live networks **SHALL** have monitoring systems in place.

[R131] Alerts for anomalies or security breaches **SHALL** be configured as part of the monitoring setup.

Recommendations:

[D372] Developers **SHOULD** integrate monitoring tools like Etherscan or Tenderly for real-time insights.

[D373] Regular reviews of monitoring data **SHOULD** be conducted to identify trends or potential issues.

5.3.4.7 Property-Based Testing Frameworks

Property-based testing frameworks allow developers to define properties that should always hold true for their smart contracts. These frameworks automatically generate test cases to verify these properties under various conditions, helping uncover edge cases that might not be considered in traditional example-based tests. By specifying general properties rather than specific scenarios, developers can ensure their contracts are robust against a wide range of inputs.

Examples of Frameworks and Test Cases:

- **QuickCheck**: Originally developed for Haskell, QuickCheck has inspired similar frameworks in other languages. It generates random inputs to test properties defined by the developer. For example, a property might state that the sum of two balances should always equal the total supply of tokens in a contract.
- **Hypothesis**: A property-based testing framework for Python that can be used to test smart contracts written in Solidity through Python bindings. An example test case might involve ensuring that a contract's function to transfer tokens never results in negative balances.

- **Echidna**: Specifically designed for Ethereum smart contracts, Echidna is a fuzzer that uses property-based testing principles to find violations of specified properties. For instance, it can test that a contract's invariant - that no more than a certain number of tokens can be minted - is never violated.

By using these frameworks, developers can systematically explore the behaviour of their smart contracts across a wide range of inputs and conditions, increasing confidence in their correctness and security [87], [53].

Requirements:

[R132] Smart contracts **SHALL** use property-based testing frameworks where applicable to validate key properties.

[R133] Properties defined in tests **SHALL** reflect critical business logic and security requirements.

Recommendations:

[D374] Developers **SHOULD** leverage frameworks like QuickCheck or Hypothesis for property-based testing.

[D375] Regular updates to property definitions **SHOULD** be made as contract logic evolves.

Property-based testing frameworks allow developers to define properties that should always hold true for their smart contracts. These frameworks automatically generate test cases to verify these properties under various conditions. This approach helps uncover edge cases that might not be considered in traditional example-based tests.

Requirements:

[R134] Smart contracts **SHALL** use property-based testing frameworks where applicable to validate key properties.

[R135] Properties defined in tests **SHALL** reflect critical business logic and security requirements.

Recommendations:

[D376] Developers **SHOULD** leverage frameworks like QuickCheck or Hypothesis for property-based testing.

[D377] Regular updates to property definitions **SHOULD** be made as contract logic evolves.

5.3.4.8 Symbolic Execution Tools

Symbolic execution tools are essential for analysing smart contract code by exploring all possible execution paths using symbolic inputs rather than concrete values. This technique helps identify vulnerabilities such as reentrancy attacks, integer overflows, or access control issues by examining how different inputs affect contract behaviour. By simulating numerous execution scenarios, symbolic execution can uncover edge cases and potential security flaws that might not be detected through conventional testing methods.

Examples of Symbolic Execution Tools:

- **MythX**: A comprehensive security analysis service that uses symbolic execution to detect vulnerabilities in Ethereum smart contracts. MythX integrates with development environments like Truffle and Remix, providing developers with detailed reports on potential security issues.
- **Manticore**: An open-source symbolic execution tool designed for analysing smart contracts and binaries. Manticore allows developers to explore multiple execution paths and identify vulnerabilities by simulating various input conditions.
- **Oyente**: One of the first symbolic execution tools developed specifically for Ethereum smart contracts. Oyente analyses bytecode to detect common security issues such as reentrancy and transaction-ordering dependence.

By leveraging these tools, developers can systematically explore the behaviour of their smart contracts across a wide range of inputs and conditions, increasing confidence in their correctness and security [28], [111], [123].

Requirements:

[R136] Smart contracts **SHALL** undergo analysis with symbolic execution tools before deployment.

[R137] Identified vulnerabilities from symbolic execution **SHALL** be addressed prior to release.

Recommendations:

[D378] Developers **SHOULD** use tools like MythX or Manticore for symbolic execution analysis.

[D379] Results from symbolic execution **SHOULD** inform additional test case development.

5.3.4.9 SMT Solvers for Smart Contracts

Satisfiability Modulo Theories (SMT) solvers are powerful tools used in formal verification processes to mathematically prove properties about smart contracts. By encoding contract logic into logical formulas, SMT solvers can verify correctness with respect to specified properties or invariants. This approach is particularly useful for ensuring that smart contracts adhere to critical business logic and security constraints, providing a high level of assurance before deployment.

Examples of SMT Solvers:

- **Z3**: Developed by Microsoft Research, Z3 is a widely used SMT solver that supports a variety of theories and is capable of handling complex logical expressions. It is often employed in the formal verification of smart contracts to ensure that they meet desired properties such as safety and liveness.
- **CVC4**: An open-source SMT solver that supports a wide range of theories and is known for its efficiency in solving complex verification problems. CVC4 can be integrated into smart contract development workflows to assist in proving the correctness of contract logic.
- **MathSAT**: A solver designed for industrial applications, MathSAT provides robust support for reasoning about mathematical formulas and is used in verifying properties of smart contracts, particularly those involving arithmetic operations.

These tools help developers ensure their smart contracts are free from logical errors and vulnerabilities by providing rigorous mathematical proofs of their correctness [52], [80], [141].

Requirements:

[R138] Critical smart contracts **SHALL**, where feasible, utilize SMT solvers for formal verification.

[R139] Properties verified by SMT solvers **SHALL** align with essential business logic and security constraints.

Recommendations:

[D380] Developers **SHOULD** integrate SMT solvers like Z3 into their verification workflows.

[D381] Results from SMT solver analyses **SHOULD** guide improvements in contract design and implementation.

5.3.5 Performing code reviews and audits

Developers **SHOULD**:

[D382] Conduct internal code reviews involving multiple developers

[D383] Engage external security experts for comprehensive audits

[D384] Use automated analysis tools to identify potential vulnerabilities

[D385] Address and rectify all identified issues and vulnerabilities

5.4 Deployment and Execution Phase

5.4.1 Discussion and recent research

The deployment and execution phase marks the transition of a smart contract from development to live operation. Recent research emphasizes the importance of careful deployment practices and ongoing monitoring. For instance, a study by Zhang et al. (2020) highlights the critical role of bytecode verification in ensuring contract integrity [165]. Additionally, work by Liu et al. (2021) underscores the importance of comprehensive monitoring strategies in identifying and mitigating runtime issues in smart contracts [91].

The deployment and execution phase requires meticulous attention to detail and robust operational procedures to ensure the contract functions as intended in the live environment. Continuous monitoring and the ability to respond quickly to any issues are crucial for maintaining the contract's security and effectiveness over time [111], [170].

This critical phase involves several key steps listed in the following clauses.

5.4.2 Compiling the contract to bytecode

Compiling a smart contract to bytecode is a critical step in the deployment process, transforming human-readable code into a format that can be executed on the blockchain's virtual machine:

- Use the appropriate compiler version for the target blockchain platform
- Ensure all dependencies are resolved and included in the compilation process
- Generate the contract's Application Binary Interface (ABI) for interaction

5.4.3 Selecting the appropriate network for deployment

Selecting the appropriate network for deploying a smart contract is a critical decision that can significantly impact the contract's performance, security, and overall success. This clause explores the key factors to consider when choosing a blockchain network for smart contract deployment.

Developers **SHOULD:**

[D386] Choose between mainnet (production) or testnet based on readiness and testing needs

[D387] Consider gas costs and network congestion when planning deployment timing

[D388] Ensure sufficient funds are available in the deploying account for transaction fees

5.4.4 Executing the deployment transaction

Executing the deployment transaction is a critical step in the lifecycle of a smart contract, marking its transition from code to an active entity on the blockchain. This process involves sending a specially crafted transaction that includes the contract's bytecode and any initialization parameters. Understanding the intricacies of this step is crucial for developers to ensure a successful and secure deployment.

Developers **SHOULD**:

[D389] Initiate the deployment transaction with appropriate gas limits and prices

[D390] Monitor the transaction status until confirmation

[D391] Record the assigned contract address for future interactions

5.4.5 Verifying the deployed contract's bytecode

The verification process involves comparing the bytecode of the deployed contract against the expected bytecode generated from the source code. By performing this verification, developers and users can confirm that the contract running on the blockchain matches the intended implementation, mitigating risks of malicious modifications or deployment errors.

Developers **SHOULD**:

[D392] Compare the on-chain bytecode with the compiled source code

[D393] Use blockchain explorers or verification tools to ensure code integrity

[D394] Publish and verify the source code on blockchain explorers for transparency

5.4.6 Monitoring the contract's execution and user interactions

The below recommendations address the importance of ongoing monitoring and the various techniques and tools available for tracking smart contract behaviour post-deployment.

Developers **SHOULD**:

[D395] Implement logging and event emission for important contract state changes

[D396] Set up monitoring tools to track contract usage, gas consumption, and potential issues

[D397] Establish a system for responding to unexpected behaviours or emergencies

5.5 Maintenance, Update and Upgrade Phases

5.5.1 Introduction

Updating smart contracts is a crucial aspect of maintaining and improving blockchain applications, presenting unique challenges due to the inherent immutability of blockchain technology. While this immutability is fundamental for ensuring security and trust, it complicates the implementation of necessary changes, such as bug fixes, feature enhancements, or adaptations to new regulatory requirements. Consequently, updating smart contracts requires careful planning and execution to maintain security, functionality, and compliance with evolving needs. This maintenance and upgrade phase is vital for ensuring the long-term viability and relevance of deployed smart contracts. Recent research emphasizes effective maintenance strategies; for instance, studies by Pranomporn et al. (2021) highlight challenges and best practices in upgrading smart contracts [112], while Wang et al. (2020) underscore the importance of governance mechanisms in managing contract evolution [148]. Balancing necessary changes with the integrity and security of the contract necessitates well-defined processes and clear communication with all stakeholders throughout this phase, enabling developers to create robust, adaptable blockchain applications by exploring various methodologies and strategies for smart contract updates [156], [172].

5.5.2 Update Situations

Updates to smart contracts may be required in several scenarios:

- **Bug Fixes**: Addressing vulnerabilities or errors discovered post-deployment.
- **Feature Enhancements**: Adding new functionalities or improving existing ones.
- **Regulatory Compliance**: Adapting to new legal requirements or standards.
- **Performance Improvements**: Optimizing contract code for better efficiency or reduced gas costs.

Requirements:

[R140] Smart contracts **SHALL** be designed with potential updates in mind from the outset.

[R141] Developers **SHALL** identify and document all scenarios that may necessitate an update.

Recommendations:

[D398] Regular reviews of contract performance and compliance **SHOULD** be performed to anticipate necessary updates.

[D399] Stakeholders **SHOULD** be engaged in identifying update needs based on user feedback and market trends.

5.5.3 Strategies of Updating

Several strategies can be employed to update smart contracts:

- **Proxy Patterns**: Using a proxy contract that delegates calls to an upgradable logic contract.
- **Modular Design**: Designing contracts with separate modules that can be individually updated.
- **Parameterization**: Allowing configurable parameters that can be adjusted without altering the core code.

Requirements:

[R142] Smart contracts **SHALL** implement secure update mechanisms that preserve data integrity.

[R143] Update strategies **SHALL** ensure minimal disruption to existing functionalities.

Recommendations:

[D400] Proxy patterns **SHOULD** be used to enable seamless upgrades while maintaining state continuity.

[D401] Modular design principles **SHOULD** be implemented to facilitate targeted updates without affecting the entire system.

5.5.4 Upgrading Through Versioning

Versioning involves deploying new versions of a contract while maintaining backward compatibility:

- **Semantic Versioning**: Using a systematic approach to version numbers to indicate the nature of changes (e.g. major, minor, patch).
- **Backward Compatibility**: Ensuring new versions do not disrupt existing integrations or user interactions.

Requirements:

[R144] Smart contracts **SHALL** include version identifiers in their metadata.

[R145] New contract versions **SHALL** maintain backward compatibility where possible.

Recommendations:

[D402] Version changes **SHOULD** be documented comprehensively to inform users and developers of updates.

[D403] Migration scripts **SHOULD** be implemented to transition data smoothly between versions.

5.5.5 Updating Steps

The process of updating a smart contract typically involves several key steps:

1) **Assessment and Planning:** Evaluate the need for an update and plan the implementation strategy.
2) **Development and Testing:** Develop the updated contract code and conduct thorough testing.
3) **Deployment Preparation:** Prepare deployment scripts and ensure all stakeholders are informed.
4) **Deployment Execution:** Deploy the updated contract on the blockchain network.
5) **Post-deployment Monitoring:** Monitor the updated contract for any issues or anomalies.

Requirements:

[R146] A detailed update plan **SHALL** be created before initiating any changes.

[R147] Comprehensive testing **SHALL** be conducted to validate updates before deployment.

Recommendations:

[D404] Stakeholders **SHOULD** be engaged throughout the update process to ensure alignment with business goals.

[D405] Automated testing frameworks **SHOULD** be used to streamline the testing phase.

5.5.6 Checklist Before Redeployment

Before redeploying an updated smart contract, developers should verify:

- All tests have passed successfully.
- Security audits have been completed and any vulnerabilities addressed.
- Documentation is up-to-date with changes clearly outlined.
- Backup mechanisms are in place for data migration if needed.

Requirements:

[R148] A comprehensive checklist **SHALL** be completed before redeployment to ensure readiness.

Recommendations:

[D406] Peer reviews of the checklist items **SHOULD** be conducted to validate completeness and accuracy.

5.5.7 Securely Inactivating Old Contract

When updating a smart contract, it is essential to securely deactivate the old version:

- Implement mechanisms such as self-destruct functions or access restrictions to prevent further interactions with the old contract.

Requirements:

[R149] Old contracts **SHALL** be securely inactivated upon deployment of a new version.

Recommendations:

[D407] Users **SHOULD** be notified about deactivation timelines and provide guidance on transitioning to the new contract version.

5.5.8 Governing the Upgrade of Smart Contracts

5.5.8.1 Discussion and recent research

Smart Contracts may reach a stage where they have to undergo an upgrade or modification. Change management of objects that are immutable by definition requires careful planning and governance. In parallel, upgrading smart contracts necessitates modifying or enhancing their functionality with minimal disruption to the existing system or user experience. Governance and upgrade models are thus crucial aspects of smart contract management, ensuring adaptability while maintaining security and decentralization. As such the upgrade may be considered part of the lifecycle of smart contracts.

Effective governance is crucial for managing smart contract upgrades.

[D408] Clear governance frameworks that outline roles, responsibilities, and decision-making processes for upgrades **SHOULD** be established.

[D409] Robust upgrade mechanisms that balance flexibility with security and immutability concerns **SHOULD** be implemented.

[D410] Transparency in decision-making by documenting proposals, voting outcomes, and implementation steps **SHOULD** be ensured.

[D411] Developers **SHOULD** facilitate stakeholder engagement through open communication channels during upgrade discussions.

Incorporating these governance practices ensures that smart contract upgrades are conducted transparently, securely, and in alignment with stakeholder interests while maintaining system integrity throughout their lifecycle.

Recent research highlights the importance of well-designed governance and upgrade models. A study by Wang et al. (2021) emphasizes the need for flexible yet secure upgrade mechanisms in smart contracts [148]. Additionally, work by Liu et al. (2020) underscores the importance of balancing decentralized governance with efficient decision-making in blockchain systems [92].

Effective governance and upgrade models are essential for the long-term viability of smart contracts, allowing them to adapt to changing requirements and technological advancements while maintaining the trust and security expected in blockchain systems [164], [160].

Governing upgrades encompasses several key elements.

5.5.8.2 Governance and upgrade models of Smart Contracts

5.5.8.2.1 Similarities and Differences Between Blockchain Platform Governance and Smart Contract Change Governance

Blockchain platform governance and smart contract change management both play crucial roles in maintaining the integrity, functionality, and adaptability of blockchain ecosystems. While they share some similarities, they also exhibit distinct differences due to their specific focuses and operational scopes.

5.5.8.2.2 Similarities

The similarities can be summarized as follows:

- **Decentralized Decision-Making:** Both blockchain platform governance and smart contract change management often rely on decentralized decision-making processes. This typically involves stakeholder voting or consensus mechanisms to approve changes, ensuring that no single entity has unilateral control over decisions.
- **Security Concerns:** Security is a paramount concern in both contexts. Ensuring that changes do not introduce vulnerabilities is critical, whether it involves updating the underlying blockchain protocol or modifying a smart contract.
- **Transparency:** Both processes emphasize transparency to build trust among participants. Decisions, proposals, and changes are usually documented and made accessible to stakeholders to ensure accountability.

5.5.8.2.3 Differences

The main differences are listed herewith:

- **Scope of Governance:**
    a) *Blockchain Platform Governance:* This encompasses broader aspects such as consensus protocol upgrades, network rules, transaction fees, and overall network policies. It affects the entire blockchain ecosystem.
    b) *Smart Contract Change Management:* This is more focused on individual contracts or applications running on the blockchain. It involves updating logic, fixing bugs, or adding features specific to a contract.

- **Stakeholder Involvement:**

  a) **Blockchain Platform Governance:** Typically involves a wide range of participants including miners, developers, users, and sometimes external entities like regulatory bodies.

  b) **Smart Contract Change Management:** Primarily involves the contract developers, users of the contract, and possibly a governance body if the contract is part of a larger Decentralized Application (DApp).

- **Frequency and Impact of Changes:**

  a) **Blockchain Platform Governance:** Changes are less frequent but have a significant impact on the entire network. They require extensive testing and consensus due to their wide-reaching implications.

  b) **Smart Contract Change Management:** Changes can be more frequent as they often pertain to specific functionalities or bug fixes within a single application. However, they have to be carefully managed to avoid disrupting dependent systems.

5.5.8.2.4 Recommendations for Effective Smart Contract Governance

5.5.8.2.4.1 Establish Clear Governance Frameworks

[D412] **Stakeholder Involvement:** All relevant stakeholders **SHOULD** be engaged in the governance processes to ensure that diverse perspectives are considered.

[D413] **Defined Roles and Responsibilities:** Roles and responsibilities for those involved in governance to streamline decision-making **SHOULD** be clearly outlined.

5.5.8.2.4.2 Implement Robust Upgrade Mechanisms

[D414] **Proxy Patterns:** Proxy contracts **SHOULD** be used to facilitate upgrades without disrupting existing functionality.

[D415] **Version Control:** Clear versioning **SHOULD** be maintained to track changes and manage compatibility across contract versions.

5.5.8.2.4.3 Ensure Security and Compliance

[D416] **Security Audits:** Regular security audits **SHOULD** be conducted before and after upgrades to identify vulnerabilities.

[D417] **Compliance Checks:** Developers **SHOULD** ensure that smart contracts comply with relevant regulations, such as data protection laws.

5.5.8.2.4.4 Facilitate Transparent Decision-Making

[D418] **Documentation:** Comprehensive documentation of all governance decisions and processes **SHOULD** be maintained to ensure transparency.

[D419] **Voting Systems:** Transparent voting mechanisms for stakeholders **SHOULD** be implemented to propose and approve changes.

5.5.8.2.4.5 Balance Flexibility with Stability

[D420] **Emergency Protocols:** Developers **SHOULD** establish protocols for emergency interventions to address critical issues promptly.

[D421] **Gradual Implementation:** Developers **SHOULD** consider phased rollouts of significant updates to minimize disruptions.

5.5.8.2.5 Designing upgrade patterns

Designing upgrades of smart contracts is very similar to designing upgrades of blockchain platforms thus similar approaches should be followed.

Developers **SHOULD:**

[D422] Implement proxy patterns to allow contract logic upgrades

[D423] Use modular design to facilitate partial upgrades and reduce risks

[D424] Implement state migration strategies for major upgrades

[D425] Ensure upgrade mechanisms are secure against unauthorized changes

5.5.8.2.6 Establishing Processes for Proposing, Voting on, and Implementing Changes in Smart Contract Change Management

5.5.8.2.6.1 Introduction

Effective management of changes in smart contracts requires a structured approach to ensure that modifications are made transparently, securely, and with stakeholder consensus. By following these structured processes, organizations can effectively manage smart contract changes while maintaining security, transparency, and stakeholder trust. The following clauses list recommendations to consider when establishing processes for proposing, voting on, and implementing changes in smart contract change management.

5.5.8.2.6.2 Proposal Submission

Developers **SHOULD**:

[D426] **Clear Submission Guidelines:** Define clear guidelines for submitting change proposals, including required documentation such as the rationale, potential impacts, and technical specifications of the proposed change.

[D427] **Stakeholder Identification:** Identify all relevant stakeholders who should be involved in the proposal process, including developers, users, and governance bodies.

[D428] **Public Accessibility:** Ensure that all proposals are publicly accessible to allow for community review and feedback.

5.5.8.2.6.3 Voting Mechanisms

Developers **SHOULD**:

[D429] **Voting Eligibility:** Establish criteria for who can vote on proposals, which may include token holders or designated governance participants.

[D430] **Voting Process:** Define the voting process, including how votes are cast (e.g. on-chain voting systems) and the duration of the voting period.

[D431] **Quorum Requirements:** Set quorum requirements to ensure that a sufficient number of stakeholders participate in the decision-making process.

5.5.8.2.6.4 Implementation Procedures

Developers **SHOULD**:

[D432] **Implementation Timeline:** Develop a timeline for implementing approved changes, allowing time for necessary preparations such as code audits and testing.

[D433] **Testing and Auditing:** Conduct thorough testing and security audits of the proposed changes before deployment to ensure they do not introduce vulnerabilities.

[D434] **Rollback Mechanisms:** Implement rollback mechanisms to revert changes if unforeseen issues arise post-deployment.

5.5.8.2.6.5 Communication and Documentation

Developers **SHOULD**:

[D435] **Transparent Communication:** Maintain open channels of communication with stakeholders throughout the change management process to provide updates and gather feedback.

[D436] **Comprehensive Documentation:** Document all stages of the change process, including proposal details, voting results, implementation steps, and any issues encountered.

5.5.8.2.7 Balancing upgradability with security and immutability

Similar to upgrades of blockchain platforms, governing the upgrade of smart-contracts requires maintaining a fine balance between security, upgradeability and immutability.

Developers **SHOULD**:

[D437] Implement role-based access control for upgrade functions

[D438] Use time-delayed upgrades to allow for community review and potential reversion

[D439] Maintain immutable core functionalities while allowing peripheral upgrades

[D440] Implement event logging for all governance actions and upgrades for auditability

5.5.8.3 Requirements and Recommendations

Requirements:

[R150] Smart contracts **SHALL** implement clearly defined governance mechanisms for managing upgrades and critical decisions.

[R151] Any upgrades or significant changes to smart contracts **SHALL** require multi-signature approval from designated governance participants.

[R152] The governance model **SHALL** include mechanisms for stakeholders to propose and vote on changes to the smart contract system.

[R153] Smart contracts **SHALL** emit events for all governance actions to maintain a transparent, on-chain record of decisions.

[R154] Upgrade mechanisms **SHALL** include time-locks to allow for community review and potential reversion of proposed changes.

Recommendations:

[D441] Governance models **SHOULD** be designed to balance decentralization with efficient decision-making processes.

[D442] Tiered governance structures **SHOULD** be considered, with different approval thresholds for various types of contract changes.

[D443] On-chain voting mechanisms **SHOULD** be implemented to enable transparent and verifiable governance decisions.

[D444] Governance participants **SHOULD** be required to stake tokens or otherwise demonstrate long-term commitment to the project.

[D445] The governance process **SHOULD** include a formal proposal stage with sufficient time for community discussion and analysis.

[D446] Upgrade patterns such as proxy contracts **SHOULD** be used to facilitate smoother contract upgrades while preserving data and contract addresses.

[D447] Emergency response procedures **SHOULD** be established within the governance model to address critical vulnerabilities quickly.

[D448] Governance actions **SHOULD** be subject to a challenge period during which stakeholders can contest decisions.

[D449] The governance model **SHOULD** include mechanisms for delegating voting power to trusted representatives.

[D450] Regular governance health checks **SHOULD** be conducted to assess the effectiveness of the model and propose improvements.

[D451] Clear documentation **SHOULD** be maintained explaining the governance process and how stakeholders can participate.

[D452] The upgrade process **SHOULD** include comprehensive testing of new contract versions in a staging environment before deployment.

[D453] Governance participants **SHOULD** be provided with education and resources to make informed decisions about proposed changes.

[D454] The governance model **SHOULD** include provisions for gradual parameter adjustments without requiring full contract upgrades.

[D455] A system of checks and balances **SHOULD** be implemented to prevent any single entity from having disproportionate control over the governance process.

5.6 Retirement or Deprecation Phase

5.6.1 Discussion and recent research

The retirement or deprecation phase is the final stage in a smart contract's lifecycle, addressing the conclusion of its operational life.

Recent research highlights the importance of proper contract retirement procedures. For instance, a study by Li et al. (2021) emphasizes the need for well-planned deprecation strategies to mitigate risks associated with abandoned contracts [90]. Additionally, work by Chen et al. (2020) underscores the importance of user consideration and data preservation in the contract retirement process [30].

The retirement or deprecation phase requires careful planning and execution to ensure a smooth transition for all stakeholders while maintaining the integrity and security of the blockchain ecosystem. It is crucial to consider the long-term implications of contract retirement, including data preservation, user impact, and potential regulatory requirements [164], [72].

Unlike previous lifecycle phases, retirement/deprecation phases are not initiated or coordinated by the developers but rather by stakeholders involved with governance of the PDL. Such stakeholders are defined in clause 6.2.5 herewith.

[D456] Retirement and Deprecation phases **SHOULD** be planned and coordinated by stakeholders handling governance tasks.

[O11] The deployment of those phases **MAY** be assigned to developers for implementation.

This phase involves several key considerations and actions.

5.6.2 Deciding when a contract should be retired

This clause explores the key considerations and criteria for determining when a smart contract should be retired from active use on the PDL.

Governance stakeholders **SHOULD**:

[D457] Assess the contract's ongoing relevance and usage.

[D458] Evaluate if the contract has been superseded by improved versions.

[D459] Consider regulatory changes that may necessitate retirement.

[D460] Analyze the cost-benefit of maintaining the contract versus retiring it.

5.6.3 Implementing a graceful shutdown process

Implementing a graceful shutdown process for smart contracts is a critical aspect of responsible PDL development and management. This process ensures that contracts can be retired or replaced without disrupting user operations or compromising data integrity.

Governance stakeholders **SHOULD**:

[D461] Execute pre-designed termination functions, if available.

[D462] Ensure all pending transactions and operations are completed.

[D463] Securely transfer any remaining assets to designated addresses.

[D464] Implement state freezing mechanisms to prevent further interactions.

5.6.4 Ensuring users are notified and given time to extract assets or data

When retiring a smart contract, it is crucial to consider the impact on users who may have assets or data stored within the contract. This clause explores the importance of proper notification and providing adequate time for users to extract their assets or data before a contract is fully decommissioned. Implementing a well-designed notification and extraction process not only demonstrates good faith and responsibility towards users but also helps mitigate potential legal and reputational risks associated with abrupt contract termination.

Governance stakeholders **SHOULD**:

[D465] Provide clear, timely communication to all stakeholders about the retirement.

[D466] Offer a grace period for users to withdraw assets or export data.

[D467] Implement user-friendly processes for asset extraction and data retrieval.

[D468] Maintain support channels for assisting users during the transition.

5.6.5 Potentially deploying a replacement contract

As smart contract systems evolve and business requirements change, there may be situations where deploying a replacement contract becomes necessary. This process, while potentially complex, is crucial for maintaining the relevance and effectiveness of blockchain-based applications. This clause explores the considerations and best practices for deploying replacement contracts, including strategies for ensuring continuity of service, preserving critical data, and managing the transition for users and interconnected systems.

Governance stakeholders and developers **SHOULD** implement:

[D469] Design and develop an improved version of the contract, if necessary.

[D470] Implement migration paths for users' assets and data to the new contract.

[D471] Ensure backward compatibility or provide clear upgrade instructions.

[D472] Deploy and thoroughly test the replacement contract before full transition.

5.7 Requirements and Recommendations

Requirements:

[R155] The PDL governance **SHALL** establish clear roles and responsibilities for smart contract lifecycle management.

[R156] Smart contracts **SHALL** undergo thorough testing and auditing before deployment to the production environment.

[R157] A formal change management process **SHALL** be implemented for any modifications to deployed smart contracts.

[R158] Smart contracts **SHALL** include mechanisms for pausing or terminating functionality in case of discovered vulnerabilities.

[R159] All smart contract deployments and upgrades **SHALL** be documented, including rationale and approval processes.

Recommendations:

[D473] Developers **SHOULD** use standardized templates and libraries when creating new smart contracts to promote consistency and security.

[D474] Smart contracts **SHOULD** be designed with upgradeability in mind, using patterns like proxy contracts where appropriate.

[D475] A staging environment **SHOULD** be used to test smart contracts in conditions closely mimicking the production environment before final deployment.

[D476] Continuous monitoring tools **SHOULD** be implemented to track smart contract performance and detect anomalies.

[D477] Regular security audits **SHOULD** be conducted on deployed smart contracts, especially before major upgrades.

[D478] Education and training programs **SHOULD** be provided to all stakeholders involved in the smart contract lifecycle.

[D479] Post-deployment reviews **SHOULD** be conducted to capture lessons learned and improve future development processes.

[D480] Contracts nearing end-of-life **SHOULD** have a clear deprecation plan, including timelines for sunsetting functionality.

[D481] Version control systems **SHOULD** be used to manage smart contract code and track changes over time.

[D482] Formal verification techniques **SHOULD** be considered for critical smart contract functions to mathematically prove correctness.

6 Requirements for Designing a Smart Contract

6.1 Smart Contract Facets

6.1.1 Categories of Facets

Smart contracts are multifaceted entities that can serve various roles within a Permissioned Distributed Ledger (PDL) ecosystem. When designing smart contracts, it is crucial to consider these different facets to ensure the contract meets its intended purpose and integrates seamlessly with the broader system. By considering these facets during the design phase, developers can create more comprehensive and effective smart contracts that fulfil their intended roles within the PDL ecosystem. Zheng, Z., Xie, S., Dai, H. N., Chen, W., Chen, X., Weng, J., & Imran, M. provide a comprehensive overview of smart contract platforms and discusses various facets such as interoperability, security, and scalability [169].

[R160] Smart contracts with business/operational roles **SHALL** incorporate both foundational and functional attributes as needed.

[D483] These contracts **SHOULD** be designed with flexibility to adapt to changing business needs.

The main facets of smart contracts can be categorized as follows.

6.1.2 Foundational Role

Smart contracts can serve a foundational role by defining core system rules, governance structures, and operational parameters. These contracts are often deployed at the genesis of the PDL or during major system upgrades. They establish the fundamental framework within which other contracts and transactions operate. Xu, X., Weber, I., Staples, M., Zhu, L., Bosch, J., Bass, L., ... & Rimba, P. outline the foundational aspects of blockchain systems and their architectural design considerations [156].

[R161] Smart contracts serving a foundational role **SHALL** define core system rules and operational parameters.

[D484] Foundational smart contracts **SHOULD** be designed with immutability and long-term stability in mind, as they form the basis of the entire system.

[D485] These contracts **SHOULD** include clear mechanisms for potential future upgrades or modifications, if allowed by the system's governance model.

6.1.3 Functional Role

In their functional role, smart contracts act as active components that execute specific business logic, manage assets, or control access within the PDL. These contracts handle day-to-day operations and interactions between participants. A paper by Wang, S., Ouyang, L., Yuan, Y., Ni, X., Han, X., & Wang, F. Y. discusses the functional roles of smart contracts in various applications and their architectural implications [148].

[R162] Functional smart contracts **SHALL** implement specific business logic and manage assets within the PDL.

[D486] Functional smart contracts **SHOULD** be modular and reusable where possible, to promote efficiency and reduce redundancy in the system.

[D487] These contracts **SHOULD** include comprehensive error handling and fail-safe mechanisms to ensure system stability.

6.1.4 Governance Role

Smart contracts can embody governance mechanisms, enabling on-chain decision-making, voting, and policy enforcement. These contracts translate organizational rules and procedures into executable code. Belchior, R., Vasconcelos, A., Guerreiro, S., & Correia, M. explore governance roles within blockchain systems and how they affect interoperability and system integration [14].

[D488] Governance smart contracts **SHOULD** incorporate flexible parameterization to allow for adjustments without requiring full contract replacement.

[R163] These contracts **SHALL** include transparent logging and event emission to ensure all governance actions are traceable and auditable.

6.1.5 Interoperability Role

Smart contracts can facilitate interoperability between different systems or layers within a PDL ecosystem. They may act as bridges, oracles, or interfaces that enable cross-chain or cross-layer communication. A paper by Zamyatin, A., Al-Bassam, M., Zindros, D., Kokoris-Kogias, E., Moreno-Sanchez, P., Kiayias, A., & Knottenbelt, W. J. provides insight into interoperability mechanisms across different blockchain platforms and their implications for smart contract design [161].

[D489] Interoperability-focused smart contracts **SHOULD** implement robust validation mechanisms for external data or cross-chain messages.

[D490] These contracts **SHOULD** be designed with version compatibility in mind to ensure long-term functionality across system upgrades.

6.2 Actors

6.2.1 Distinct roles in a smart contract

When designing smart contracts for a Permissioned Distributed Ledger (PDL), it is crucial to identify and define the various actors involved in the contract's lifecycle. These actors play distinct roles in the creation, deployment, execution, and management of smart contracts. Understanding these actors and their responsibilities is essential for designing secure, efficient, and effective smart contracts. By carefully considering these actors and their requirements during the design phase, developers can create smart contracts that are more robust, secure, and aligned with the needs of all stakeholders in the PDL ecosystem.

[R164] The PDL governance **SHALL** establish clear roles and responsibilities for smart contract lifecycle management.

[D491] Mechanisms **SHOULD** be in place to manage stakeholder interactions and dispute resolutions.

6.2.2 Contract Developer

The Contract Developer is a key actor in the smart contract ecosystem, responsible for designing, coding, testing, and optimizing smart contracts for deployment on a Permissioned Distributed Ledger (PDL). This role requires a deep understanding of both the technical aspects of blockchain technology and the specific business logic the contract aims to implement. Contract Developers should be proficient in specialized programming languages such as Solidity or Vyper, and have a strong grasp of cryptographic principles, data structures, and distributed systems. They are tasked with translating complex business requirements into efficient, secure, and bug-free code. This involves not only writing the core functionality but also implementing robust error handling, access controls, and upgrade mechanisms. Contract Developers should also be adept at using development tools, testing frameworks, and security analysis software to ensure the reliability and safety of their code. As smart contracts often deal with valuable assets or critical processes, developers should maintain a security-first mindset, anticipating potential vulnerabilities and attack vectors. Continuous learning is crucial in this role, as the field of smart contract development is rapidly evolving with new best practices, design patterns, and security considerations emerging regularly.

[D492] Developers **SHOULD** have a thorough understanding of the PDL platform, its programming language, and best practices for smart contract development [169].

[R165] Developers **SHALL** implement comprehensive testing suites to validate the contract's functionality and security [72].

6.2.3 Contract Owner

The Contract Owner is a critical stakeholder in the lifecycle of a smart contract, typically the entity or individual responsible for deploying the contract to the Permissioned Distributed Ledger (PDL). This role carries significant authority and responsibility within the contract's ecosystem. The Contract Owner usually possesses special privileges, such as the ability to upgrade the contract, modify critical parameters, or even pause its functionality in case of emergencies. They are often tasked with managing access controls, determining who can interact with specific contract functions, and potentially appointing other administrative roles. In many cases, the Contract Owner represents the business entity or consortium that initiated the smart contract project. Their responsibilities extend beyond deployment, including overseeing the contract's operation, coordinating with developers for updates or bug fixes, and potentially managing any funds or assets controlled by the contract. The Contract Owner has to maintain a delicate balance between exercising control and fostering trust among users, as excessive centralization can undermine the principles of decentralized systems. In some advanced governance models, the concept of contract ownership may evolve into a more distributed form, with decisions being made collectively by stakeholders through voting mechanisms encoded within the contract itself [129], [92].

[R166] The contract **SHALL** include clearly defined owner functions with appropriate access controls.

[R167] Ownership transfer mechanisms **SHALL** be implemented to allow changes in contract management.

6.2.4 Contract Users

Contract Users represent the diverse group of individuals, organizations, or even other smart contracts that interact with a deployed smart contract on a Permissioned Distributed Ledger (PDL). These actors form the primary audience for whom the smart contract is designed and implemented. Contract Users may have varying levels of permissions and capabilities within the contract's ecosystem, ranging from basic interactions like querying data to more complex operations such as initiating transactions or modifying contract states. They may include customers, service providers, stakeholders, or automated systems, each with their own set of rights and responsibilities defined by the contract's logic and access control mechanisms. The experiences and needs of Contract Users significantly influence the design and functionality of smart contracts, as developers have to ensure intuitive interfaces, efficient execution, and appropriate safeguards for different user roles. Contract Users may interact with the smart contract through various means, such as Decentralized Applications (DApps), APIs, or directly through the PDL's interface. Their actions and transactions form the core operational data of the smart contract, driving its state changes and triggering predefined outcomes. As such, understanding the diverse requirements, technical capabilities, and potential behaviours of Contract Users is crucial for creating effective, user-friendly, and secure smart contracts that fulfil their intended purpose within the PDL ecosystem [39].

[O12] The contract **MAY** implement role-based access control to manage different levels of user permissions.

[R168] User interactions **SHALL** be logged and emit events for transparency and auditability.

6.2.5 Governance Body

The Governance Body in a Permissioned Distributed Ledger (PDL) ecosystem serves as the overarching authority responsible for maintaining the integrity, security, and efficiency of the network, including the management of smart contracts. This entity, which may be composed of representatives from various stakeholders, establishes and enforces the rules and policies that govern the PDL. In the context of smart contracts, the Governance Body plays a crucial role in approving new contracts for deployment, overseeing major upgrades, and intervening in case of critical issues or disputes. They may have special privileges encoded within the smart contracts themselves, such as the ability to pause operations in emergencies or to trigger upgrades [148]. The Governance Body also ensures that smart contracts align with regulatory requirements and the overall objectives of the PDL consortium [175]. Implementing a robust governance structure is essential for maintaining trust among participants and adapting to changing needs and circumstances over time [156], [148], [175].

[R169] The contract **SHALL** include hooks or interfaces for governance-level controls, such as pausing or upgrading.

[D493] Governance decisions affecting the contract **SHOULD** be implemented through a multi-signature or time-locked mechanism for enhanced security.

6.2.6 Auditors

Auditors play a critical role in ensuring the security, efficiency, and compliance of smart contracts within a Permissioned Distributed Ledger (PDL) ecosystem. These specialized professionals conduct comprehensive reviews of smart contract code, analysing it for potential vulnerabilities, logical flaws, and adherence to best practices. Their work extends beyond static code analysis to include dynamic testing and simulation of various execution scenarios. Auditors also evaluate the contract's on-chain behaviour post-deployment, examining transaction patterns and state changes to verify that the contract operates as intended in real-world conditions. In the context of PDLs, auditors may additionally focus on compliance with specific regulatory requirements or consortium rules [111]. The increasing complexity of smart contracts has led to the development of automated auditing tools and formal verification methods to complement manual auditing processes [52]. Regular audits throughout a smart contract's lifecycle are crucial for maintaining trust in the system and quickly identifying any emerging issues or vulnerabilities [91]. Consequently, smart contract designers have to prioritize auditability from the outset, implementing clear documentation, standardized patterns, and comprehensive event logging to facilitate thorough and effective auditing processes [95], [111].

[D494] The contract code **SHOULD** be well-documented and follow standardized patterns to facilitate auditing.

[R170] The contract **SHALL** implement comprehensive event logging to aid in post-deployment auditing.

6.2.7 Oracles

Oracles play a crucial role in bridging the gap between smart contracts and the external world. These entities provide external data to smart contracts, enabling them to interact with off-chain information and trigger contract executions based on real-world events. Oracles can be software interfaces that pull data from web APIs, hardware devices that capture physical world data, or even human experts providing specialized information. The integrity and reliability of oracle data are paramount, as smart contracts rely on this information to execute critical functions. Recent research has focused on developing decentralized oracle networks to mitigate single points of failure and enhance data reliability [2]. When designing smart contracts that interact with oracles, developers have to implement robust validation mechanisms and consider potential attack vectors such as oracle manipulation or outdated data [96]. Additionally, incorporating multiple oracle sources and implementing a consensus mechanism for critical data inputs can significantly enhance the security and reliability of oracle-dependent smart contracts [156].

[D495] The contract **SHOULD** implement a decentralized oracle pattern to avoid single points of failure.

[R171] Oracle data **SHALL** be validated and include timestamps to ensure freshness and reliability.

6.3 Requirements During Design

6.3.1 Key considerations

When designing smart contracts for a Permissioned Distributed Ledger (PDL), several key requirements should be considered to ensure the contract's security, efficiency, and effectiveness. These requirements guide the development process and help create robust, reliable smart contracts. By adhering to these requirements during the design phase, developers can create smart contracts that are more secure, efficient, and aligned with the needs of the PDL ecosystem.

[R172] Smart contracts **SHALL** be designed with consideration for their entire lifecycle, from deployment to termination.

6.3.2 Security

Security is paramount in smart contract design, as vulnerabilities can lead to significant financial losses and compromise the integrity of the entire system. It encompasses measures to protect against unauthorized access, data breaches, and malicious attacks. Secure smart contracts should be resistant to common vulnerabilities and implement strong access controls to ensure that only authorized parties can execute critical functions or modify contract states [129], [72], [52].

[D496] Comprehensive access control mechanisms **SHOULD** be implemented to restrict function calls and state modifications to authorized parties only.

[D497] Secure coding practices **SHOULD** be used to prevent common vulnerabilities such as reentrancy, integer overflow, and unauthorized access.

[D498] Formal verification techniques **SHOULD** be incorporated to mathematically prove the correctness of critical contract functions.

6.3.3 Scalability

Scalability in smart contract design refers to the ability of the contract to handle increasing amounts of work or accommodate growth without compromising performance. This is crucial for ensuring that the contract remains efficient and cost-effective as the number of users or transactions increases. Scalable smart contracts should minimize computational and storage costs while allowing for future expansion and upgrades [31], [92], [156].

[D499] Contracts **SHOULD** be designed with gas optimization in mind, minimizing computational and storage costs.

[D500] Modular design patterns **SHOULD** be implemented to allow for contract upgradability and extensibility.

[O13] Off-chain storage solutions **MAY** be considered for large datasets to reduce on-chain storage requirements.

6.3.4 Interoperability

Interoperability in smart contracts refers to the ability of the contract to interact seamlessly with other contracts, systems, or even different blockchain networks. This is essential for creating interconnected ecosystems and enabling complex multi-contract operations. Interoperable smart contracts should use standardized interfaces and be designed with cross-chain compatibility in mind when necessary [148], [161].

[D501] Contracts **SHOULD** be designed with standardized interfaces to facilitate interaction with other contracts and systems.

[D502] Cross-chain compatibility features **SHOULD** be implemented when required, using secure bridge protocols.

6.3.5 Auditability

Auditability in smart contracts refers to the ability to trace and verify all actions and state changes within the contract. This is crucial for transparency, debugging, and ensuring compliance with regulatory requirements. Auditable smart contracts should implement comprehensive logging mechanisms and be written in a clear, well-documented manner to facilitate review and analysis [39], [177].

[R173] Implement comprehensive event logging **SHALL** be implemented for all significant state changes and important function calls.

[D503] Clear, well-documented code **SHOULD** be provided with inline comments explaining complex logic.

6.3.6 Privacy

Privacy in smart contract design involves protecting sensitive information and ensuring that confidential data is not exposed on the public ledger. This is particularly important in permissioned environments where certain information should only be accessible to authorized parties. Privacy-preserving smart contracts should minimize on-chain storage of sensitive data and implement cryptographic techniques to protect confidential information when necessary [83], [3], [21].

[R174] Implement privacy-preserving techniques such as zero-knowledge proofs **SHALL** be implemented for sensitive data when required.

[D504] Contracts **SHOULD** be designed to minimize the storage of personally identifiable information on-chain.

6.3.7 Governance

Governance in smart contracts refers to the mechanisms that allow for the management and evolution of the contract over time, while respecting the principle of immutability inherent to blockchain systems. This presents a unique challenge: how to adapt and improve contracts that are, by design, unchangeable once deployed. Effective governance in smart contracts strikes a balance between preserving the integrity and trust provided by immutability, and the need for flexibility to address unforeseen issues, changing requirements, or emergent vulnerabilities. This often involves implementing upgradeable patterns, parameter adjustment mechanisms, or modular designs that allow for controlled changes without compromising the contract's core logic or historical data. Well-governed smart contracts should include transparent, decentralized decision-making processes for any potential modifications, ensuring that all stakeholders have a voice in significant changes while maintaining the security and reliability of the system [175], [18], [174], [40], [60], [23].

[D505] Upgradeable contract patterns (such as proxy patterns) that allow for logic updates while preserving data and contract address **SHOULD** be implemented.

[D506] Parameter adjustment mechanisms that allow for fine-tuning of contract behaviour without changing core logic **SHOULD** be Incorporated.

[D507] Modular contract systems where individual components can be replaced or updated without affecting the entire system **SHOULD** be designed.

[R175] Time-locks and multi-signature requirements for critical operations **SHALL** be implemented to ensure transparency and consensus in governance actions.

[R176] Event emission for all governance actions **SHALL** be included to maintain a transparent, on-chain record of changes.

[D508] Implementation of a Decentralized Autonomous Organization (DAO) structure **SHOULD** be considered for major governance decisions, allowing stakeholders to vote on proposed changes.

6.3.8 Error Handling

Error handling in smart contracts involves managing unexpected situations or inputs gracefully to prevent system failures or vulnerabilities. Robust error handling is crucial for maintaining the stability and security of the contract, especially in high-stakes environments. Smart contracts with effective error handling should be able to manage exceptions without compromising the integrity of the system or exposing vulnerabilities.

[R177] Robust error handling and exception management **SHALL** be implemented to gracefully handle unexpected situations.

[R178] Contracts **SHALL** be designed with circuit breaker mechanisms to allow pausing of contract functionality in case of emergencies.

6.4 Available technologies evaluation and selection

6.4.1 Introduction

When designing smart contracts for Permissioned Distributed Ledgers (PDLs), developers should consider the various technologies available to ensure efficient, secure, and effective smart contract implementation. The choice of technologies can significantly impact the smart contract's performance, security, and interoperability.

6.4.2 Programming Languages

Smart contract programming languages are specialized to cater to the unique requirements of blockchain environments. These languages should balance expressiveness with security, allowing developers to implement complex logic while minimizing the risk of vulnerabilities. Popular languages like Solidity (for Ethereum-based systems) and Go (for Hyperledger Fabric) continue to evolve, with newer alternatives like Rust gaining traction for their enhanced safety features [177].

[R179] Smart contracts **SHALL** be written in programming languages supported by the target PDL platform.

[D509] Developers **SHOULD** choose widely adopted languages with robust tooling and community support.

6.4.3 Development Frameworks

Development frameworks provide essential tools and libraries that streamline the smart contract creation process. These frameworks often include testing suites, deployment scripts, and interaction utilities. Examples include Truffle for Ethereum-based systems and Hyperledger Composer for Fabric. The choice of framework can significantly impact development efficiency and contract quality [71].

[R180] Any external libraries used in smart contract development **SHALL** be approved by the PDL governance.

[D510] Libraries **SHOULD** be thoroughly audited and their versions carefully managed.

6.4.4 Security Analysis Tools

Given the critical nature of smart contracts, security analysis tools are essential for identifying vulnerabilities before deployment. Static and dynamic analysis tools, as well as formal verification systems, help developers catch potential issues early in the development process. Tools like Mythril, Slither, and Manticore have become integral parts of the smart contract development lifecycle [52].

[R181] Smart contract developers **SHALL** utilize security analysis tools to identify potential vulnerabilities before deployment.

[R182] Smart contract developers **SHALL** employ static and dynamic analysis tools to ensure comprehensive security coverage.

[D511] Smart contract developers **SHOULD** use formal verification tools to mathematically prove the correctness of critical contract functions.

[D512] Smart contract developers **SHOULD** incorporate fuzzing techniques to test contracts with unexpected or random inputs.

[D513] Smart contract developers **SHOULD** regularly update and integrate automated security checks within continuous integration pipelines.

6.4.5 Oracle Services

Oracles bridge the gap between smart contracts and external data sources, allowing contracts to interact with real-world information. Decentralized oracle networks like Chainlink have gained prominence for their ability to provide reliable, tamper-resistant data to smart contracts. The integration of oracle services should be carefully considered to maintain the contract's security and decentralization [2].

[R183] Smart contracts **SHALL** implement secure oracle services to fetch external data reliably.

[R184] Oracles **SHALL** ensure data integrity and authenticity when providing information to smart contracts.

Smart contract developers **SHOULD:**

[D514] Use decentralized oracle networks to avoid reliance on a single data source.

[D515] Implement validation mechanisms for oracle data to prevent manipulation or inaccuracies.

[D516] Consider using multiple oracles and aggregating their results for enhanced reliability.

6.4.6 Interoperability Protocols

As the blockchain ecosystem grows more diverse, interoperability protocols are becoming increasingly important. These technologies allow smart contracts to interact across different blockchain networks, expanding their utility and reach. Projects like Polkadot and Cosmos are at the forefront of enabling cross-chain communication for smart contracts [161].

[R185] Smart contracts **SHALL** adhere to standardized interoperability protocols for cross-chain communication.

[R186] Interoperability mechanisms **SHALL NOT** compromise the security or integrity of the smart contract.

Smart contract developers **SHOULD:**

[D517] Implement cross-chain bridges and atomic swap protocols for secure asset transfers between different blockchains.

[D518] Use standardized data formats and interfaces to facilitate seamless integration with other systems.

[D519] Consider employing modular designs to enable easier interoperability with various platforms.

6.4.7 Privacy-Enhancing Technologies

In permissioned environments, preserving data privacy while maintaining transparency is crucial. Zero-knowledge proofs, secure multi-party computation, and other privacy-enhancing technologies are being integrated into smart contract platforms to enable confidential transactions and computations [83].

[R187] Smart contracts handling sensitive information **SHALL** implement privacy-enhancing technologies.

[R188] Privacy measures **SHALL** comply with relevant data protection regulations, such as GDPR.

Smart contract developers **SHOULD:**

[D520] Utilize zero-knowledge proofs or secure multi-party computation techniques for privacy-preserving operations.

[D521] Implement encryption for sensitive on-chain data, ensuring only authorized parties can access it.

[D522] Consider using privacy-focused blockchain platforms or solutions that support confidential transactions.

6.4.8 Scalability Solutions

As blockchain networks face scalability challenges, various layer-2 solutions and sharding technologies are being developed. These can significantly impact smart contract design, allowing for more complex operations without overwhelming the main chain. Solutions like Optimistic Rollups and zk-Rollups are becoming increasingly relevant for smart contract developers [156].

[R189] Smart contracts **SHALL** incorporate scalability solutions to handle increased transaction volumes efficiently.

[R190] Scalability measures **SHALL** maintain the security and performance of the smart contract.

Smart contract developers **SHOULD:**

[D523] Implement layer-2 solutions, such as state channels or rollups, to enhance transaction throughput.

[D524] Optimize gas usage in smart contracts by employing efficient algorithms and data structures.

[D525] Consider sharding or partitioning strategies for large-scale applications requiring high performance.

6.5 Auditability considerations

6.5.1 Definition of Auditability

Auditability is a crucial aspect of smart contract design, ensuring transparency, accountability, and trust in the contract's operations. It allows stakeholders to verify the contract's behaviour, trace transactions, and detect any anomalies or potential vulnerabilities.

[R191] Smart contract code and all associated libraries **SHALL** be available for auditing purposes.

[D526] Contracts **SHOULD** use standardized, well-documented patterns to facilitate easier auditing.

6.5.2 Code Transparency

Smart contract code should be open and accessible for review by all relevant parties. This transparency allows for community-driven audits and helps build trust among users. Developers should strive to write clean, well-documented code that is easy to understand and analyse. Using established design patterns and following coding standards can significantly enhance code readability and auditability [177].

[D527] Smart contract code **SHOULD** be open and accessible for review by all relevant parties.

6.5.3 Event Logging

Implementing comprehensive event logging is essential for tracking the contract's state changes and important operations. Events should be emitted for all significant actions, providing a detailed audit trail that can be easily queried and analysed. This not only aids in debugging and monitoring but also serves as a transparent record of the contract's activities [39].

[R192] All smart contract transactions and communications **SHALL** be available for audit to authorized parties.

[D528] Comprehensive event logging **SHOULD** be implemented to facilitate thorough auditing.

6.5.4 Formal Verification

Formal verification techniques can be employed to mathematically prove the correctness of smart contract behaviour. While complex, these methods provide a high degree of assurance about the contract's adherence to its specifications. Incorporating formal verification into the development process can significantly enhance the contract's auditability and reliability [102].

[R193] Verification processes **SHALL** ensure that the contract logic aligns with its intended behaviour and specifications.

[D529] Smart contracts **SHOULD** undergo formal verification to mathematically prove the correctness of critical functions.

Smart contract developers **SHOULD**:

[D530] Use formal verification tools such as Coq, Isabelle, or K to validate smart contract logic.

[D531] Prioritize formal verification for high-value contracts or those handling sensitive data.

[D532] Integrate formal verification into the development lifecycle to catch issues early.

6.5.5 Automated Analysis Tools

Leveraging automated analysis tools can greatly enhance the auditability of smart contracts. Static analysis tools, dynamic analysis frameworks, and symbolic execution engines can help identify potential vulnerabilities, gas inefficiencies, and logical flaws. Integrating these tools into the development workflow ensures continuous auditing throughout the contract's lifecycle [28].

Smart contract developers **SHOULD**:

[D533] Implement automated analysis tools to identify potential vulnerabilities and inefficiencies in smart contract code. Such tools should cover both static and dynamic analysis to provide comprehensive security coverage.

[D534] Use tools like MythX, Slither, or Oyente for static analysis of smart contracts.

[D535] Incorporate fuzz testing and symbolic execution to explore multiple execution paths and uncover hidden issues.

[D536] Regularly update analysis tools to incorporate the latest security checks and improvements.

6.5.6 Version Control and Change Management

Maintaining a clear record of contract versions and changes is crucial for auditability. Implementing robust version control practices and documenting all modifications, including the rationale behind them, ensures that the contract's evolution can be traced and audited. This is particularly important for upgradeable contracts [92].

Smart contract developers **SHALL**:

[R194] Utilize version control systems to manage changes in smart contract code, ensuring traceability and accountability.

[R195] Implement change management processes to handle updates and modifications systematically.

Smart contract developers **SHOULD**:

[D537] Use platforms like Git for version control, maintaining a clear history of code changes.

[D538] Establish a formal process for proposing, reviewing, and approving changes to smart contracts.

[D539] Document all changes thoroughly, including rationale and potential impacts on the system.

6.5.7 External Audits

While internal auditing processes are essential, engaging third-party auditors provides an additional layer of scrutiny and credibility. External audits can uncover overlooked vulnerabilities and offer fresh perspectives on the contract's design and implementation. Establishing a regular external audit schedule is a best practice for critical smart contracts [111].

[R196] Smart contract developers **SHALL** conduct external audits of smart contracts by reputable third-party firms before deployment.

[R197] Such audits **SHALL** assess both security vulnerabilities and compliance with specifications.

Smart contract developers **SHOULD**:

[D540] Schedule regular audits as part of the contract maintenance routine, especially after significant updates.

[D541] Choose auditors with expertise in blockchain technology and a proven track record in smart contract security.

[D542] Address all findings from audits promptly, implementing necessary fixes or improvements before deployment.

6.6 Designing and implementing Input and Output methods to Smart Contracts

6.6.1 Generalized Input/Output Requirements

6.6.1.1 Introduction

Smart contracts often need to interact with external data sources and systems to fulfil their intended functions. However, this interaction introduces potential security and reliability risks. Therefore, it is crucial to establish robust requirements for data inputs and outputs. By implementing these requirements and recommendations, smart contracts can interact with both internal and external data sources in a secure and reliable manner, enhancing their functionality while mitigating associated risks.

Generalized Input/Output Requirements form the foundational principles for managing data flow in smart contract systems. These requirements encompass a set of universal best practices and standards that apply to all types of data interactions, whether internal or external to the blockchain. They address critical aspects such as data integrity, format validation, error handling, and rate limiting. By establishing these generalized requirements, smart contract developers can create more robust, secure, and interoperable systems that can reliably process inputs and generate outputs. These requirements serve as a crucial safeguard against common vulnerabilities and ensure that smart contracts can effectively manage the diverse range of data they encounter, from on-chain transactions to off-chain oracle inputs. Adhering to these generalized requirements not only enhances the functionality and security of individual smart contracts but also contributes to the overall reliability and trustworthiness of decentralized applications built on blockchain platforms.

6.6.1.2 Data Integrity and Authenticity

Data Integrity and Authenticity are critical aspects of smart contract data inputs and outputs. Integrity ensures that data remains unaltered and complete throughout its lifecycle, from input to storage and output. Authenticity verifies that the data originates from a legitimate and expected source. E.g. Implementing hash functions to verify data integrity in smart contracts ensures that the data has not been altered since it was originally created.

In smart contracts, these properties are typically enforced through cryptographic mechanisms such as digital signatures and hash functions. For inputs, smart contracts should verify the integrity and authenticity of data before processing it, rejecting any data that fails these checks. For outputs, contracts should provide cryptographic proofs that allow external systems to verify the integrity and authenticity of the data produced. Implementing robust measures for data integrity and authenticity helps prevent attacks such as data tampering and unauthorized access, thereby enhancing the overall security and reliability of smart contract systems.

Zheng, Z., Xie, S., Dai, H. N., Chen, W., Chen, X., Weng, J., & Imran, M. discuss methods for ensuring data integrity and authenticity in smart contracts, emphasizing cryptographic techniques and secure data handling [169].

[R198] All data inputs to smart contracts **SHALL** be cryptographically signed to ensure integrity and authenticity.

[R199] Smart contracts **SHALL** verify the signatures of input data before processing it.

[D543] Developers **SHOULD** implement multi-signature schemes for highly sensitive data inputs to enhance security.

6.6.1.3 Data Format and Validation

Data Format and Validation are essential considerations for ensuring the reliability and security of smart contract operations. Smart contracts should strictly define the expected format of input data, including data types, structures, and acceptable ranges or patterns. E.g. Using JSON schema validation within smart contracts to ensure that incoming data adheres to expected formats before processing.

Rigorous validation routines should be implemented to check all incoming data against these predefined formats and constraints. This process helps prevent issues such as type errors, buffer overflows, or logical inconsistencies that could arise from malformed or unexpected inputs. For outputs, smart contracts should ensure data is formatted consistently and in compliance with any relevant standards or protocols. Proper data formatting and validation not only enhance the contract's robustness but also improve interoperability with other systems and contracts. Furthermore, clear error handling and reporting mechanisms should be in place to manage and log any validation failures, providing valuable feedback for debugging and security analysis. Xu et al. [156] outline the importance of standardized data formats and validation processes in blockchain systems to ensure interoperability and correctness.

[R200] Smart contracts **SHALL** implement strict input validation to check data types, ranges, and formats.

[R201] Any data that fails validation **SHALL** be rejected, and the transaction reverted.

[D544] Developers **SHOULD** use standardized data formats (e.g. JSON, Protobuf) for interoperability.

6.6.1.4 Error Handling

Error Handling is a crucial aspect of smart contract design, particularly when dealing with data inputs and outputs. Robust error handling mechanisms ensure that smart contracts can gracefully manage unexpected situations, invalid inputs, or failed operations without compromising the integrity of the system. E.g. Implementing try-catch blocks in smart contract code to handle exceptions gracefully and revert transactions when errors occur.

For inputs, contracts should implement comprehensive checks and provide clear, specific error messages when validation fails. These errors should be carefully designed to be informative for debugging purposes while avoiding the exposure of sensitive information. For outputs, error handling should account for potential failures in external calls or state changes, implementing patterns like checks-effects-interactions to maintain consistency. Additionally, contracts should emit events for significant errors, allowing for off-chain monitoring and analysis. Proper error handling not only enhances the reliability and security of smart contracts but also improves their maintainability and user experience by providing clear feedback and preventing unintended state changes. Zou, W., Lo, D., Kochhar, P. S., Le, X. B. D., Xia, X., Feng, Y., & Xu, B. [177] discuss error handling mechanisms in smart contracts to prevent failures and ensure robustness.

[R202] Smart contracts **SHALL** implement comprehensive error handling for all input/output operations.

[R203] Error messages **SHALL** be logged but not exposed directly to end-users to prevent information leakage.

6.6.1.5 Rate Limiting

Rate Limiting is an essential mechanism for protecting smart contracts against potential abuse or overload, particularly when dealing with external data inputs or frequent interactions. By implementing rate limiting, contracts can restrict the frequency of certain operations or data inputs within a given time frame. This helps prevent denial-of-service attacks, where an attacker might attempt to overwhelm the contract or the underlying blockchain network with a flood of transactions. E.g. Using token bucket algorithms within smart contracts to limit the number of transactions a user can perform within a given time frame. For data inputs, rate limiting can be applied to external oracle updates or user-driven data submissions. In the context of outputs, it can control the frequency of state updates or external calls. Smart contracts can implement rate limiting through various methods, such as enforcing minimum time intervals between operations, using token bucket algorithms, or leveraging global transaction limits set by the blockchain protocol. Effective rate limiting not only enhances the security and stability of individual smart contracts but also contributes to the overall health of the blockchain ecosystem by preventing resource exhaustion. Wang, S., Ouyang, L., Yuan, Y., Ni, X., Han, X., & Wang, F. Y. [148] cover strategies for implementing rate limiting in blockchain applications to prevent abuse and ensure fair resource allocation.

[D545] Developers **SHOULD** implement rate limiting for external data inputs to prevent DoS attacks.

6.6.1.6 Randomness

Generating secure random numbers within smart contracts is challenging due to the deterministic nature of blockchain execution.

[D546] When randomness is required, developers **SHOULD** consider using external sources of entropy or implementing multi-party computation schemes.

It is crucial to understand the limitations and potential vulnerabilities associated with on-chain randomness [92].

6.6.1.7 Governance Inputs

For contracts with upgradeability or parameter adjustment features, governance inputs play a crucial role.

[D547] These inputs, often in the form of votes or multi-signature approvals, **SHOULD** be carefully designed to ensure security, fairness, and resistance to manipulation.

Implementing time-locks and quorum requirements can enhance the security of governance-related inputs [151].

6.6.2 Internal Data Inputs

6.6.2.1 Introduction

Internal data inputs in smart contracts refer to the data that originates and is processed within the blockchain ecosystem itself, without relying on external sources. These inputs are crucial for the functioning of smart contracts as they determine how contracts interact with other on-chain elements, such as other contracts, tokens, and blockchain-specific events. The management of internal data inputs involves ensuring that data is accurately captured, securely stored, and efficiently processed to maintain the integrity and reliability of the smart contract's operations. This clause explores the mechanisms and best practices for handling internal data inputs, focusing on aspects such as inter-contract communication, on-chain data sources, and the implications of using blockchain-native data in smart contract logic. By effectively managing internal data inputs, developers can enhance the functionality and security of smart contracts, enabling them to perform complex operations autonomously within a decentralized environment.

6.6.2.2 Inter-Contract Communication

Inter-contract communication refers to the ability of smart contracts to interact with one another within the blockchain ecosystem, enabling them to call functions, share data, and coordinate actions. This capability is essential for building complex Decentralized Applications (DApps) where multiple contracts work together to execute intricate workflows. Effective inter-contract communication relies on well-defined interfaces and standards, such as ERC interfaces in Ethereum, which ensure compatibility and interoperability between different contracts. It also involves careful consideration of security aspects, as improper handling of inter-contract calls can lead to vulnerabilities like reentrancy attacks. Recent advancements have focused on enhancing the robustness and efficiency of these interactions through techniques like atomic swaps and cross-chain bridges, which facilitate secure and seamless communication across different blockchain networks. By leveraging these mechanisms, developers can create more versatile and interconnected smart contract systems that extend beyond the limitations of individual contracts.

Zamyatin, A., Al-Bassam, M., Zindros, D., Kokoris-Kogias, E., Moreno-Sanchez, P., Kiayias, A., & Knottenbelt, W. J. [161] discusses interoperability within blockchain networks and the importance of effective inter-contract communication for achieving it.

Xu, X., Weber, I., Staples, M., Zhu, L., Bosch, J., Bass, L., & Rimba, P. [156] outline architectural considerations for blockchain systems, including the role of inter-contract communication in system design.

Belchior, R., Vasconcelos, A., Guerreiro, S., & Correia, M. [14] discuss interoperability within blockchain networks and the importance of effective inter-contract communication for achieving it.

[R204] Smart contracts **SHALL** use well-defined interfaces for inter-contract communication.

[D548] Developers **SHOULD** implement access control mechanisms to restrict which contracts can call sensitive functions.

[D549] Developers **SHOULD** use the "pull over push" pattern for payments to prevent reentrancy attacks.

EXAMPLE: A payment contract could be designed with separate functions for recording payments and withdrawing funds. The recording function would update an internal balance, while the withdrawal function would first set the balance to zero before initiating the transfer. This "pull over push" pattern helps prevent reentrancy attacks by ensuring that the contract's state is updated before any external calls are made.

6.6.2.3 On-Chain Data Sources

On-chain data sources refer to the information that is natively available within the blockchain ecosystem, which smart contracts can access and utilize without relying on external inputs. This data includes transaction details, block information, and other contract states that are stored on the blockchain. Utilizing on-chain data sources ensures that smart contracts operate with high reliability and security, as this data is inherently part of the blockchain's immutable ledger. Smart contracts can leverage on-chain data to trigger actions, verify conditions, and interact with other contracts, enabling complex Decentralized Applications (DApps) to function autonomously. The use of on-chain data sources minimizes the need for external dependencies, thereby reducing potential vulnerabilities associated with off-chain interactions. Recent advancements in blockchain technology have enhanced the efficiency and accessibility of on-chain data, allowing developers to build more robust and scalable smart contract solutions.

Implementing safeguards against reentrancy attacks and other vulnerabilities associated with contract interactions is crucial [169], [156], [154].

[D550] When designing contracts that interact with on-chain data, developers **SHOULD** consider the potential for malicious contracts, state inconsistencies, and gas limitations.

[R205] Smart contracts **SHALL** verify the authenticity of on-chain data sources (e.g. other contracts, storage).

[D551] Developers **SHOULD** use cryptographic commitments or zero-knowledge proofs for privacy-preserving on-chain data access.

6.6.3 External Data Inputs

6.6.3.1 Definition

External data inputs come from off-chain sources. External data inputs are critical to the functionality and versatility of smart contracts, enabling them to interact with real-world information and events. Unlike internal data inputs, which are confined to the blockchain environment, external data inputs are sourced from outside the blockchain, typically through oracles or other data feeds. These inputs allow smart contracts to execute based on conditions that reflect real-world scenarios, such as flight status, network congestion, or survey data. The integration of external data inputs expands the potential use cases for smart contracts by allowing them to automate processes that depend on dynamic and real-time information. However, incorporating external data also introduces challenges related to data integrity, authenticity, and security. Ensuring that external data is accurate and tamper-proof is crucial for maintaining the reliability and trustworthiness of smart contracts. As such, developers have to carefully design mechanisms for sourcing and validating external data to mitigate risks and enhance the contract's functionality.

6.6.3.2 Oracles

6.6.3.2.1 Definition

Oracles play a crucial role in bridging the gap between blockchain-based smart contracts and external data sources, enabling smart contracts to interact with real-world information. They act as trusted intermediaries that fetch, verify, and relay data from outside the blockchain to the smart contract, allowing it to execute based on dynamic inputs such as financial market data, weather conditions, or sports results. The use of oracles expands the functionality of smart contracts by providing them with access to external events and information that are not natively available on the blockchain. However, integrating oracles introduces challenges related to data integrity, trust, and security, as the reliability of a smart contract's execution becomes dependent on the accuracy and honesty of the oracle. Recent advancements have focused on developing decentralized oracle networks, such as Chainlink (see note below). which aim to mitigate these risks by distributing trust across multiple nodes and ensuring that no single point of failure can compromise the system's integrity.

NOTE: **Chainlink**: Next-Gen Blockchain Oracle Network (2020). Chainlink documentation provides insights into how decentralized oracle networks function and their role in enhancing smart contract capabilities.

Zamyatin, A., et al. discuss the importance of secure communication mechanisms like oracles in enabling cross-chain interactions and data integration [161].

Al-Breiki, H., et al. review various oracle solutions and highlights ongoing challenges in ensuring their reliability and security within blockchain ecosystems [2].

[D552] Smart contracts **SHOULD** use decentralized oracle networks instead of single-source oracles to enhance reliability.

[D553] Developers **SHOULD** Implement a time delay between receiving oracle data and acting on it to allow for dispute resolution.

[D554] Developers **SHOULD** Use crypto-economic incentives to ensure oracle honesty.

EXAMPLE: A smart contract could be designed to interact with a decentralized oracle network like Chainlink. It would include functions to request data (such as price information) from the oracle network and to receive and process the response. The contract would specify the data source, the type of data required, and any processing needed (such as scaling the result). The oracle network would then fetch this data, process it as specified, and return it to the contract in a separate transaction.

6.6.3.2.2 Oracle selection

Selecting the appropriate oracle is a critical aspect of integrating external data inputs into smart contracts, as it directly impacts the reliability, security, and accuracy of the data being fed into the blockchain. Oracles serve as bridges between the off-chain world and on-chain environments, providing smart contracts with access to real-world data necessary for executing complex operations. When selecting an oracle, several factors have to be considered: the trust model (centralized vs. decentralized), data source reliability, latency, cost, and security features. Decentralized oracles, such as Chainlink, are often preferred for their ability to reduce single points of failure and enhance trust through distributed data validation. Additionally, the oracle's ability to provide cryptographic proofs of data authenticity and integrity is crucial for maintaining the security of smart contracts. Recent advancements in oracle technology have focused on improving scalability and reducing latency while ensuring robust security measures are in place to protect against data manipulation and other attacks.

[R206] Smart contracts **SHALL** use oracle services that provide cryptographic proofs of their computations.

[D555] Developers **SHOULD** implement a reputation system for oracles based on their historical accuracy and reliability.

[D556] Developers **SHOULD** use specialized oracles for different types of data (e.g. financial data, IoT data) to leverage domain expertise.

6.6.3.2.3 Data Aggregation

Data aggregation in the context of oracles as external data sources for smart contracts involves the process of collecting, verifying, and consolidating data from multiple sources to ensure accuracy and reliability before it is fed into the blockchain. This is crucial because smart contracts often rely on real-world data to execute functions, and any inaccuracies can lead to incorrect contract behaviour. Aggregating data from multiple oracles helps mitigate risks associated with single points of failure or manipulation by providing a consensus on the data's validity. Recent advancements in decentralized oracle networks, such as Chainlink, have emphasized the importance of robust data aggregation mechanisms to enhance the security and trustworthiness of smart contracts. These systems use multiple independent nodes to fetch and verify data, ensuring that the aggregated result is both accurate and tamper-proof. By implementing effective data aggregation strategies, smart contracts can achieve higher levels of reliability and resilience against potential attacks or errors in external data inputs.

[D557] Developers **SHOULD** implement robust aggregation methods (e.g. median) for data from multiple oracles to resist outliers and attacks.

[D558] Developers **SHOULD** use weighted aggregation based on oracle reputation scores.

EXAMPLE: A data aggregator contract could be designed to collect reports from multiple oracles. Each oracle would submit its data along with a timestamp. The contract would store this information along with the oracle's reputation score. When aggregating the data, the contract would consider only recent reports (e.g. within the last hour) and weight each report by the oracle's reputation. The final result would be a weighted average of the valid reports. This approach helps to mitigate the impact of outliers or malicious reports.

6.6.3.2.4 Oracle Security

Oracle security is a critical consideration when integrating external data sources into smart contracts, as oracles act as the bridge between on-chain and off-chain data. Ensuring the security of oracles is essential to maintaining the integrity and reliability of the smart contracts that depend on them. Oracles have to be protected against various threats, such as data tampering, unauthorized access, and denial-of-service attacks. Decentralized oracle networks, like Chainlink, enhance security by distributing trust across multiple nodes, reducing the risk of a single point of failure and increasing resistance to manipulation. Additionally, implementing cryptographic proofs and secure data transmission protocols can further safeguard the data being fed into smart contracts. Recent advancements in oracle technology have focused on improving security measures to ensure that smart contracts receive accurate and reliable data inputs, which is crucial for their correct execution and the prevention of potential exploits.

Developers **SHOULD:**

[D559] Implement circuit breakers to halt contract execution if oracle data deviates significantly from expected ranges.

[D560] Use commit-reveal schemes for time-sensitive oracle data to prevent front-running.

[D561] Implement oracle rotation to distribute trust and prevent single points of failure.

6.6.3.3 Off-Chain Data Sources

Off-chain data sources refer to information that originates outside the blockchain ecosystem and is accessed by smart contracts to execute specific functions or make decisions based on real-world events. These data sources are crucial for enabling smart contracts to interact with external systems and environments, thereby extending their functionality beyond the blockchain. Off-chain data can include anything from financial market prices and weather data to supply chain information and IoT sensor readings [163]. To incorporate this data securely and reliably, smart contracts often rely on oracles, which act as trusted intermediaries that fetch, verify, and transmit the data to the blockchain. The integration of off-chain data presents challenges such as ensuring data integrity, authenticity, and minimizing trust dependencies. Recent advancements in decentralized oracle networks aim to address these challenges by distributing trust across multiple nodes, thereby enhancing the security and reliability of off-chain data inputs in smart contracts.

Al-Breiki, et al. [2] discuss the challenges in ensuring the reliability and security of off-chain data integration.

Zamyatin, et al. [161] explore secure communication mechanisms for integrating off-chain data into blockchain systems.

[D562] Developers **SHOULD** Implement multiple independent data sources for critical off-chain data to ensure reliability.

[D563] Developers **SHOULD** Use secure hardware (e.g. TEEs) for sensitive off-chain computations when possible.

[D564] Developers **SHOULD** Implement a challenge-response mechanism for large off-chain datasets.

6.6.3.4 User Inputs

User inputs are the primary means by which external actors interact with smart contracts.

[R207] These inputs **SHALL** be carefully validated to prevent malicious actions or unintended behaviours.

[R208] Smart contracts **SHALL** only accept input from authorized sources.

Implementing robust input validation mechanisms, including range checks, type checks, and format validations, is essential to maintain the contract's integrity and security [72].

6.6.3.5 Time-Based Inputs

Many smart contracts rely on time-based triggers or conditions. However, blockchain timestamps can be manipulated to some extent by miners.

[D565] Developers **SHOULD** be aware of these limitations and implement appropriate safeguards when using time-based inputs.

Considering block numbers as a proxy for time or using commit-reveal schemes can mitigate some of these risks [47].

6.6.4 Smart Contract Outputs

6.6.4.1 Introduction

Smart contract outputs consist of several types of data as described in the following clauses. By carefully designing these output methods, smart contracts can effectively communicate their state changes, provide necessary data, and interact with both on-chain and off-chain systems in a secure and efficient manner.

6.6.4.2 Event Emission

Smart contracts can emit events to log important state changes and actions. Events serve as an efficient way to notify external systems or user interfaces about contract activities.

When designing event structures the developers **SHOULD**:

[D566] Include relevant data as indexed parameters for efficient filtering

[D567] Emit events for all significant state changes

[D568] Use descriptive names for events and their parameters

6.6.4.3 Return Values

Functions in smart contracts can return values to provide immediate feedback or data to the caller. When implementing return values the developers **SHOULD**:

[D569] Define clear and consistent return types for functions

[D570] Use multiple return values when appropriate to provide comprehensive information

[D571] Consider using structs for complex return data

6.6.4.4 State Updates

Smart contracts modify their internal state as a result of function executions. Proper state management is crucial for maintaining contract integrity.

When updating state developers **SHOULD**:

[D572] Ensure all state changes are atomic and consistent

[D573] Use appropriate data structures (e.g. mappings, arrays) for efficient state storage

[D574] Implement access controls to protect sensitive state variables

6.6.4.5 External Calls

Smart contracts can make calls to other contracts or external addresses. When designing external calls developers **SHOULD**:

[D575] Implement proper error handling for failed calls

[D576] Be aware of potential reentrancy vulnerabilities

[D577] Use the appropriate call methods (e.g. transfer, send, call) based on the specific requirements

6.6.4.6 Off-Chain Notifications

While not directly part of the blockchain, smart contracts can trigger off-chain notifications through oracles or other middleware. Developers **SHOULD** Consider:

[D578] Implementing webhook-like mechanisms for notifying external systems

[D579] Using oracles to bridge on-chain events with off-chain processes

[D580] Ensuring proper authentication and encryption for off-chain communications

6.7 Using a universal clock

6.7.1 The criticality of Universal Time

The concept of time in smart contracts is crucial for many applications, yet it presents unique challenges in distributed systems, as certain nodes may reside in different time-zones and others may rely on an unsynchronized internal clock. A universal clock mechanism is essential for ensuring consistency and fairness in time-dependent contract executions.

6.7.2 Time Representation

[R209] Smart contracts **SHALL** have a standardized way of representing time to ensure consistent execution across all nodes.

Most blockchain platforms use block timestamps or block numbers as a proxy for time. Developers should be aware of the granularity and potential manipulation of these time representations when designing time-sensitive contracts [47].

6.7.3 Consensus on Time

In distributed systems, achieving consensus on the current time is challenging. Smart contract platforms typically rely on the consensus mechanism to agree on block timestamps. However, there can be slight variations between nodes.

[D581] Developers **SHOULD** account for these potential discrepancies in time-critical applications [74].

6.7.4 Time Drift Mitigation

Over time, the cumulative effect of small inconsistencies in timekeeping can lead to significant drift. Implementing mechanisms to periodically synchronize contract time with a trusted external time source can help mitigate this issue. However, care has to be taken to maintain the decentralized nature of the system [153].

[D582] Contracts **SHOULD** implement mechanisms to handle potential time discrepancies between nodes.

6.7.5 Time-based Triggers

Many smart contracts rely on time-based triggers for executing certain functions.

[D583] Developers **SHOULD** implement time-based triggers with care, considering potential manipulations by miners or validators.

Using block numbers as a proxy for time or implementing commit-reveal schemes can provide more reliable time-based executions [89].

6.7.6 Time Zones and Localization

For contracts that interact with real-world events across different time zones, proper handling of time zone conversions is crucial. Storing all timestamps in UTC and performing conversions only when necessary can help avoid confusion and errors related to time zone differences [145].

[R210] Smart contracts **SHALL** use the time/zone format defined by the PDL governance.

6.7.7 Time Oracles

In cases where high precision or tamper-resistant time data is required, time oracles can be employed. These external services provide verified time data to smart contracts. However, the use of oracles introduces additional trust considerations that should be carefully managed [2].

[O14] Time Oracles **MAY** be used.

6.8 Terminatability considerations

6.8.1 Problem Definition of Terminatability

The ability to safely terminate or deactivate a smart contract is a crucial design consideration, especially for long-running or upgradeable contracts. Proper termination mechanisms ensure that contracts can be retired gracefully when they are no longer needed or when critical issues are discovered.

[R211] Smart contracts **SHALL** include a mechanism for termination or deactivation.

[D584] Termination mechanisms **SHOULD** include proper state cleanup and event emission.

6.8.2 Self-Destruction Mechanisms

Self-destruction (or "selfdestruct" in Solidity) is a built-in feature in some smart contract platforms that allows for the complete removal of a contract from the blockchain. While powerful, this mechanism should be used cautiously and with proper access controls to prevent unauthorized termination [175].

[D585] Developers **SHOULD** consider the implications of self-destruction on contract dependencies and user assets.

NOTE: It is important to understand that self-destruction does not violate the principle of blockchain immutability. When a contract self-destructs, it does not erase its past transactions or states from the blockchain's history. Instead, it renders the contract address invalid for future interactions and typically transfers any remaining balance to a designated address. The contract's bytecode is replaced with an empty value, but all previous interactions with the contract remain permanently recorded on the blockchain. This nuance highlights the need for careful consideration when implementing self-destruction mechanisms, as they have permanent and irreversible effects on the contract's future usability while preserving its historical record [10].

6.8.3 Graceful Shutdown

Implementing a graceful shutdown process allows for a controlled deactivation of the contract's functionalities. This typically involves a multi-step process where the contract enters a "shutdown mode," preventing new operations while allowing users to withdraw their assets or complete ongoing transactions [92].

[D586] Smart Contracts **SHOULD** include mechanisms for change and graceful termination.

6.8.4 Time-Based Termination

For contracts with a predetermined lifespan, implementing time-based termination can ensure that the contract automatically ceases operations after a specific date or block number. This approach requires careful consideration of time representation in blockchain environments and potential manipulations of block timestamps [47].

[O15] Smart contracts **MAY** include mechanisms to automatically terminate after a predefined time period or date.

[R212] The termination process **SHALL** ensure that all pending transactions are processed before the contract is closed.

Smart contract developers **SHOULD**:

[D587] Use blockchain timestamps to trigger time-based termination reliably.

[D588] Implement notifications for stakeholders prior to termination to allow for any necessary actions.

[D589] Consider using a grace period after the termination trigger to handle any last-minute transactions.

6.8.5 Condition-Based Termination

Contracts can be designed to terminate based on specific conditions, such as reaching a certain state, completing a particular task, or responding to external triggers.

[D590] Smart contracts **SHOULD** define specific conditions under which they will terminate automatically.

[R213] The conditions for such termination **SHALL** be clearly documented and accessible to all stakeholders involved in the contract and resistant to manipulation to prevent premature or delayed termination [72].

Smart contract developers **SHOULD**:

[D591] Use oracles or other reliable data sources to verify external conditions that trigger termination.

[D592] Implement thorough testing of condition-based triggers to prevent premature or unintended terminations.

[D593] Allow for manual override or review processes in case of disputes over condition satisfaction.

6.8.6 Governance-Controlled Termination

For contracts with complex ecosystems or high-value assets, implementing a governance-controlled termination process can provide additional security and flexibility. This typically involves multi-signature approval or stakeholder voting to initiate the termination process [148].

[D594] Smart contracts **SHOULD** include governance mechanisms that allow authorized parties to terminate the contract if necessary.

[R214] The governance process **SHALL** be transparent and involve multiple stakeholders to prevent misuse.

Smart contract developers **SHOULD**:

[D595] Use multi-signature approvals or voting systems for governance-controlled termination decisions.

[D596] Document all governance processes and decisions related to termination for auditability.

[D597] Implement safeguards against malicious or unauthorized termination attempts by requiring consensus among stakeholders.

6.8.7 Data Preservation and State Finalization

When terminating a contract, it is crucial to consider the preservation of important data and the finalization of the contract's state. Implementing mechanisms to archive data or transfer it to a successor contract can ensure continuity and maintain historical records [145].

[R215] Upon termination, smart contracts **SHALL** preserve critical data and finalize their state to ensure integrity and auditability.

[R216] Data preservation mechanisms **SHALL** comply with relevant legal and regulatory requirements.

[O16] Smart contract developers **MAY** use off-chain storage solutions for preserving large datasets while maintaining references on-chain.

Smart contract developers **SHOULD**:

[D598] Implement mechanisms to provide stakeholders with access to preserved data after contract termination.

[D599] Consider using Merkle trees or similar structures for efficient verification of preserved data integrity.

6.9 Security aspects of smart contract design

The security aspects of smart contract design are discussed in depth in clause 4.3.9.9.

7 Architectural requirements for Smart Contracts

7.1 Reusability

7.1.1 Definition of Reusability

Reusability is a crucial architectural requirement for smart contracts, as it promotes efficiency, reduces development time, and minimizes the potential for errors. By creating contracts that can be easily adapted and reused, developers can build more robust and maintainable systems. Reusable smart contracts also contribute to standardization within the blockchain ecosystem, fostering interoperability and reducing the learning curve for new developers. Reusability is discussed in depth in clause 4.3.9.1 herewith. The following text provides requirements and recommendations with the specific context of Architecture of smart contracts.

7.1.2 Contract Templates

Developing standardized contract templates for common use cases enhances reusability and reduces development time. These templates can be customized for specific needs while maintaining a consistent and tested foundation. Contract templates serve as building blocks for more complex systems and can significantly speed up the development process while ensuring best practices are followed [92].

[R217] Smart contract systems **SHALL** implement standardized templates for common use cases.

[D600] Developers **SHOULD** create a library of customizable contract templates for different scenarios.

[D601] Templates **SHOULD** be regularly reviewed and updated to incorporate best practices and security improvements.

7.1.3 Library Development

Creating reusable libraries for common functionalities allows for code sharing across multiple contracts. This approach reduces redundancy and improves overall code quality. Libraries can encapsulate complex logic, security features, or standard implementations of common patterns, making them valuable resources for the entire development community [92].

[R218] Reusable code libraries **SHALL** be developed and maintained for common smart contract functionalities.

[D602] Libraries **SHOULD** be thoroughly tested and audited before being used in production contracts.

[D603] Version control and dependency management **SHOULD** be implemented for all libraries.

7.1.4 Inheritance and Composition

Utilizing inheritance and composition patterns enables the creation of flexible and reusable contract structures. This allows for the extension of existing contracts with new functionalities while maintaining a modular design. These object-oriented principles, when applied to smart contracts, provide a powerful way to create extensible and maintainable code [177].

[R219] Smart contracts **SHALL** utilize inheritance and composition patterns to promote code reuse and extensibility.

[D604] Developers **SHOULD** favour composition over deep inheritance hierarchies to maintain flexibility.

[D605] Contract interfaces **SHOULD** be used to define standard behaviours that can be implemented by multiple contracts.

7.2 Self-destruction

7.2.1 Definition

Self-destruction mechanisms in smart contracts are essential for managing the contract lifecycle and maintaining the overall health of the blockchain network. By allowing contracts to be safely terminated when they are no longer needed, self-destruction helps prevent the accumulation of obsolete contracts and reduces unnecessary blockchain bloat.

7.2.2 Controlled Termination

Implementing controlled self-destruction mechanisms allows for the safe termination of contracts when they are no longer needed. This helps in managing the contract lifecycle and reducing unnecessary blockchain bloat. Controlled termination should include safeguards to ensure that only authorized parties can initiate the self-destruction process [175].

[R220] Smart contracts **SHALL** include a secure mechanism for self-destruction when they are no longer needed.

[D606] Self-destruction functions **SHOULD** be protected with multi-signature or time-lock mechanisms.

[D607] The ability to initiate self-destruction **SHOULD** be limited to authorized roles or governance processes.

7.2.3 State Cleanup

Designing proper state cleanup procedures ensures that all relevant data is appropriately handled or transferred before contract termination. This may involve transferring remaining funds, settling outstanding transactions, or archiving important data. Proper state cleanup is crucial to prevent loss of assets or information [175].

[R221] Self-destruction processes **SHALL** include comprehensive state cleanup procedures.

[D608] Contracts **SHOULD** implement functions to transfer any remaining assets before self-destruction.

[D609] Important data **SHOULD** be archived or transferred to a designated storage contract before termination.

7.2.4 Event Emission

Emitting events upon self-destruction provides a clear audit trail and notifies relevant parties of the contract's termination. These events can be monitored off-chain to trigger necessary actions or updates in connected systems, ensuring smooth transitions and maintaining system integrity [175].

[R222] Smart contracts **SHALL** emit events upon initiation and completion of the self-destruction process.

[D610] Emitted events **SHOULD** include relevant details such as the initiator and timestamp.

[D611] Systems interacting with the contract **SHOULD** monitor these events to update their state accordingly.

7.3 Data Ownership

7.3.1 Definition

Data ownership is a critical concern in smart contract architecture, especially in permissioned environments where privacy and access control are paramount. Properly implemented data ownership mechanisms ensure that sensitive information is protected while still allowing for necessary operations and interactions within the contract ecosystem.

7.3.2 Access Control Mechanisms

Implementing robust access control mechanisms ensures that data within the contract is only accessible or modifiable by authorized parties. This can involve Role-Based Access Control (RBAC) systems, multi-signature requirements, or other advanced permission structures that align with the specific needs of the contract and its stakeholders [164].

[R223] Smart contracts **SHALL** implement robust access control mechanisms to protect data ownership.

[D612] Role-Based Access Control (RBAC) **SHOULD** be used to manage permissions granularly.

[D613] Access control lists **SHOULD** be updatable by authorized governance processes.

7.3.3 Data Portability

Designing contracts with data portability in mind allows for the transfer of ownership or migration of data to new contracts when necessary. This is particularly important for long-lived systems where contract upgrades or migrations may be required. Data portability ensures that valuable information can be preserved and transferred securely [164].

[R224] Smart contract systems **SHALL** provide mechanisms for securely transferring data ownership or migrating data.

[D614] Contracts **SHOULD** implement functions to export data in a standardized format.

[D615] Data migration processes **SHOULD** include verification steps to ensure data integrity.

7.3.4 Privacy-Preserving Techniques

Incorporating privacy-preserving techniques, such as zero-knowledge proofs, can protect sensitive data while still allowing for necessary computations and verifications. These advanced cryptographic methods enable contracts to operate on confidential data without exposing the underlying information, striking a balance between transparency and privacy [83].

[R225] Smart contracts dealing with sensitive data **SHALL** incorporate privacy-preserving techniques.

[D616] Zero-knowledge proofs **SHOULD** be used where possible to verify computations without revealing data.

[D617] Contracts **SHOULD** minimize on-chain storage of sensitive data, preferring off-chain storage with on-chain verification.

7.4 Reference Architecture

7.4.1 Problem statement

A well-designed reference architecture provides a blueprint for creating robust, scalable, and maintainable smart contracts. It defines the overall structure and key components of the contract system, ensuring consistency and best practices across different implementations. ETSI has published ETSI GS PDL 012 [45] which defines a reference architecture of PDLs. While not specific to Smart Contracts, it should be adhered to where applicable.

[D618] Developers **SHOULD** adhere to the ETSI PDL Reference Architecture defined in ETSI GS PDL 012 [45].

7.4.2 Modular Design

Modularity is a fundamental architectural principle for smart contract design, promoting flexibility, reusability, and maintainability. In the context of smart contracts, modularity involves breaking down complex systems into smaller, self-contained components or modules, each responsible for specific functionalities. This approach allows developers to create more manageable and upgradeable smart contract systems, facilitating easier maintenance, upgrades, and reuse of contract components. Modular design enhances code reusability, allowing common functionalities to be shared across different contracts or projects, and supports better scalability and upgradability, as specific modules can be replaced or upgraded without affecting the entire system. Furthermore, it facilitates easier testing and auditing, as individual components can be verified independently. Implementing modularity in smart contracts often involves using design patterns such as proxy contracts, libraries, and factory patterns. By adhering to modular architecture, smart contract developers can create more robust, flexible, and efficient systems that are better equipped to evolve with changing requirements and technological advancements in the blockchain ecosystem, while also creating more adaptable systems that are easier to maintain and audit [148].

[R226] Smart contracts **SHALL** be designed with a modular architecture to enhance maintainability and reusability.

[D619] Contracts **SHOULD** be divided into logical modules with clear interfaces between them.

[D620] Each module **SHOULD** have a single responsibility to enhance maintainability.

[D621] Developers **SHOULD** use design patterns such as the proxy pattern to enable upgradeable contract logic.

7.4.3 Standardized Interfaces

Implementing standardized interfaces enhances interoperability and allows for easier integration with other contracts and systems. Standardized interfaces, such as those defined by ERC standards in Ethereum, provide a common language for contract interactions, promoting ecosystem-wide compatibility and reducing integration complexities [148].

[R227] Smart contracts **SHALL** implement standardized interfaces for common functionalities.

[D622] Developers **SHOULD** adhere to established standards (e.g. ERC standards for Ethereum) where applicable.

[D623] Custom interfaces **SHOULD** be well-documented and follow consistent patterns.

7.4.4 Separation of functionalities

Clearly separating different functionalities (e.g. logic, data storage, access control) within the contract architecture improves maintainability and security. This principle helps in managing complexity by isolating different aspects of the contract, making it easier to update, debug, and secure individual components without affecting the entire system [148].

[R228] Smart contract architecture **SHALL** clearly separate different functionalities.

[D624] Logic, data storage, and access control **SHOULD** be implemented in separate contract components.

[D625] Interaction between separated components **SHOULD** be through well-defined interfaces.

7.5 Scalability Solutions

7.5.1 Problem statement

Scalability is a significant challenge in blockchain systems, and smart contract architecture should consider scalability solutions to ensure performance and cost-effectiveness as the network grows. Implementing scalable smart contracts is crucial for supporting large-scale, high-throughput applications. The clauses below list approaches that may be considered to improve smart-contract scalability.

7.5.2 Layer-2 Integration

Layer-2 solutions represent a critical approach to smart contract offloading, addressing scalability and efficiency challenges inherent in base layer blockchain networks. These solutions operate as secondary frameworks built on top of the main blockchain, processing transactions and executing smart contract logic off-chain while leveraging the security of the underlying blockchain. Popular Layer-2 technologies include state channels, sidechains, rollups (both optimistic and zero-knowledge), and plasma chains. By moving computation and data storage off the main chain, Layer-2 solutions can significantly increase transaction throughput, reduce gas costs, and improve overall system performance. For instance, rollups can bundle multiple transactions into a single on-chain submission, dramatically reducing fees and increasing processing speed. Designing contracts with Layer-2 scalability solutions in mind can significantly improve transaction throughput and reduce costs, allowing for faster and cheaper transactions while still benefiting from the security of the main chain. Moreover, some Layer-2 solutions offer enhanced privacy features, allowing for confidential transactions and selective data disclosure. When implementing Layer-2 solutions, developers have to carefully consider trade-offs between scalability, security, and decentralization, ensuring that the chosen approach aligns with the specific requirements of their decentralized application while maintaining the integrity and trust guarantees provided by the base layer blockchain [156].

[R229] Smart contract systems **SHALL** be designed to integrate with Layer-2 scalability solutions where appropriate.

[D626] Developers **SHOULD** consider implementing state channels or rollups for high-frequency, low-value transactions.

[D627] Rollup-friendly design patterns **SHOULD** be adopted to enable efficient off-chain computation.

7.5.3 Sharding Compatibility

Ensuring contracts are compatible with sharding architectures can enhance scalability in blockchain networks that implement sharding. Sharding divides the network into smaller partitions, each capable of processing transactions independently. Smart contracts designed with sharding in mind can take full advantage of this parallel processing capability [156].

[R230] Smart contracts **SHALL** be designed to be compatible with sharding architectures.

[D628] Contracts **SHOULD** minimize cross-shard operations to maintain efficiency in sharded environments.

[D629] Data structures **SHOULD** be designed to allow efficient partitioning across shards.

7.5.4 Gas Optimization

Implementing gas-efficient coding practices and data structures helps in reducing transaction costs and improving overall scalability. This includes optimizing storage usage, minimizing expensive operations, and designing efficient algorithms. Gas optimization is crucial for making smart contracts economically viable on public blockchains [31].

[R231] Smart contracts **SHALL** implement gas-efficient coding practices and data structures.

[D630] Loops **SHOULD** be optimized to minimize gas consumption.

[D631] Storage patterns **SHOULD** be chosen to minimize gas costs for frequent operations.

7.6 Privacy-Preserving Smart Contracts

7.6.1 Problem statement

The inherent transparency of blockchain technology, while beneficial for ensuring trust and accountability, presents significant challenges when dealing with sensitive or confidential information in smart contracts. This transparency can potentially expose private data, intellectual property, or business-critical information to unauthorized parties, limiting the adoption of blockchain solutions in industries where data privacy is paramount, such as healthcare, finance, and legal services.

Challenges:

1) **Data Exposure:** Traditional smart contracts store all data on-chain, making it visible to all network participants.

2) **Regulatory Compliance:** Many industries are subject to strict data protection regulations, which can conflict with blockchain's transparent nature.

3) **Competitive Advantage:** Businesses may be reluctant to use blockchain if it means exposing proprietary information to competitors.

4) **User Privacy:** Individual users may be hesitant to interact with smart contracts that could reveal their personal or financial information.

Privacy-preserving smart contracts address the challenge of maintaining confidentiality in a transparent blockchain environment. By incorporating privacy-preserving mechanisms such as those listed in the following clauses, smart contracts can maintain the benefits of blockchain technology while protecting sensitive information. This approach opens up new possibilities for blockchain applications in fields where data confidentiality is crucial, potentially accelerating the adoption of blockchain solutions across various industries and use cases.

7.6.2 Zero-Knowledge Proofs

Incorporating zero-knowledge proof technologies allows for verification of computations without revealing underlying data. This powerful cryptographic technique enables contracts to prove the validity of a statement or computation without disclosing any information beyond the truth of the statement itself, maintaining privacy while ensuring correctness [9].

[R232] Smart contracts handling sensitive data **SHALL** incorporate zero-knowledge proof technologies where applicable.

[D632] zk-SNARKs or zk-STARKs **SHOULD** be used for privacy-preserving verifications.

[D633] The choice of zero-knowledge proof system **SHOULD** consider the trade-offs between proof size, verification time, and setup requirements.

NOTE 1: Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (zk-SNARK) is a cryptographic proof system that enables one party to prove to another that they know a value, without revealing any information about the value itself. zk-SNARKs are characterized by their succinctness, meaning the proofs are very small and can be verified quickly, and non-interactivity, which means they do not require interaction between the prover and verifier after the initial setup phase. This technology is widely used in blockchain applications to enhance privacy and scalability, as it allows for the verification of complex computations without revealing the underlying data or requiring extensive computational resources from the verifier.

NOTE 2: Zero-Knowledge Scalable Transparent Argument of Knowledge (zk-STARK) is a cryptographic proof system designed to provide similar functionalities to zk-SNARKs but with greater scalability and transparency. Unlike zk-SNARKs, zk-STARKs do not rely on a trusted setup phase, making them more secure and easier to implement. They use hash functions instead of elliptic curves, which contributes to their scalability and resistance to quantum attacks. zk-STARKs generate larger proofs than zk-SNARKs but are more efficient in terms of proving time and verification speed, making them suitable for applications requiring high throughput and robust security assurances.

7.6.3 Secure Multi-Party Computation

Implementing secure multi-party computation techniques enables collaborative computations while keeping individual inputs private. This allows multiple parties to jointly compute a function over their inputs while keeping those inputs secret, opening up possibilities for privacy-preserving collaborations and data analysis [11].

[R233] Smart contracts requiring collaborative computations on private data **SHALL** implement secure multi-party computation techniques.

[D634] Threshold cryptography **SHOULD** be used to distribute trust among multiple parties.

[D635] The number of parties involved in the computation **SHOULD** be carefully chosen to balance security and efficiency.

7.6.4 Encrypted Data Processing

Designing contracts to work with encrypted data, such as through homomorphic encryption, allows for computations on sensitive data without exposing it. This advanced technique enables contracts to perform operations on encrypted data, producing encrypted results that can be decrypted only by authorized parties, thus maintaining data confidentiality throughout the computation process [11].

[R234] Smart contracts operating on sensitive data **SHALL** support computations on encrypted data where necessary.

[D636] Homomorphic encryption **SHOULD** be used for operations on encrypted data when full data confidentiality is required.

[D637] The performance impact of encrypted data processing **SHOULD** be carefully evaluated and optimized.

7.7 Smart Contract Offloading

7.7.1 Introduction

Smart Contract Offloading is an advanced architectural approach that addresses multiple challenges in blockchain systems, including scalability, privacy, and data management. This technique involves strategically relocating certain computations, data storage, or processing tasks away from the main blockchain to enhance system performance, reduce costs, and improve data privacy. By offloading select operations to external systems, secondary chains, or privacy-preserving computation environments, smart contracts can overcome limitations such as high transaction fees, limited on-chain storage, slow processing times, and lack of data confidentiality.

This clause explores various offloading strategies, including the use of sidechains, layer-2 solutions, secure enclaves, and zero-knowledge proof systems. It discusses how these approaches can be leveraged not only for performance optimization but also for implementing robust data segregation and privacy-preserving computations. The discussion will cover techniques for maintaining data confidentiality while still leveraging the transparency and immutability benefits of blockchain technology.

This clause discusses the trade-offs between on-chain and off-chain execution, considering factors such as security, decentralization, data availability, and regulatory compliance. The clause also provides insights into designing hybrid architectures that balance the need for public verifiability with requirements for data protection and selective disclosure.

Understanding and effectively utilizing smart contract offloading techniques is crucial for creating scalable, efficient, and privacy-aware decentralized applications that can meet the complex demands of enterprise and regulated environments while still adhering to the core principles of blockchain technology.

7.7.2 Sidechain Integration

Sidechain integration represents a powerful approach to smart contract offloading, offering a balance between scalability, customization, and security. Sidechains are separate blockchain networks that run parallel to the main chain, connected through a two-way peg mechanism that allows for the secure transfer of assets and data. By offloading smart contract execution and data storage to sidechains, developers can significantly reduce congestion on the main network while maintaining a connection to its security and liquidity. Sidechains can be optimized for specific use cases, allowing for tailored consensus mechanisms, faster block times, and specialized functionality. This flexibility enables the creation of application-specific chains that can handle complex computations or high-frequency transactions more efficiently than the main chain. Furthermore, sidechains can offer enhanced privacy features, as not all data needs to be broadcast to the entire network. When implementing sidechain integration, developers have to carefully design the cross-chain communication protocols and consider the security implications of the sidechain's validator set. Effective sidechain integration can dramatically improve the scalability and functionality of decentralized applications while leveraging the robustness of the main blockchain network.

[R235] The PDL architecture **SHALL** allow for sidechain integration to offload specific smart contract operations.

[D638] Sidechains **SHOULD** be used for specialized operations that require different consensus rules or higher transaction throughput.

7.7.3 Off-chain Computation

Off-chain computation is a crucial strategy for smart contract offloading, addressing limitations in on-chain processing power and cost-effectiveness. This approach involves executing complex or resource-intensive calculations outside the blockchain environment, while using the blockchain primarily for data integrity and settlement. Off-chain computation can be implemented through various technologies, including Trusted Execution Environments (TEEs), decentralized oracle networks, and verifiable computation protocols. By moving intensive computations off-chain, smart contracts can handle more complex logic, process larger datasets, and operate with greater efficiency. This is particularly beneficial for applications requiring real-time data processing, machine learning algorithms, or compliance with data privacy regulations. The results of off-chain computations are typically submitted back to the blockchain with cryptographic proofs of correctness, ensuring transparency and verifiability. Implementing off-chain computation requires careful consideration of the trust model, as it introduces potential centralization risks. Developers have to design robust verification mechanisms and consider hybrid approaches that balance the benefits of off-chain processing with the security guarantees of on-chain execution. When properly implemented, off-chain computation can significantly enhance the capabilities and performance of smart contract systems, enabling more sophisticated and scalable decentralized applications.

[R236] Smart contracts **SHALL** be designed to leverage off-chain computation where possible, with on-chain verification.

[D639] Complex computations **SHOULD** be performed off-chain, with results verified and recorded on-chain to reduce blockchain bloat.

NOTE: Trusted Execution Environments (TEEs) are secure areas within a main processor that provide a protected environment for executing code and processing data. TEEs are isolated from the rest of the device and the operating system, ensuring that data and code within the TEE are protected from unauthorized access, modification, and tampering.

7.8 Design Patterns

7.8.1 Introduction and problem statement

Smart contract development has evolved significantly since its inception, giving rise to a rich ecosystem of design patterns that address various challenges in blockchain-based applications. This clause delves into a comprehensive exploration of these design patterns, each tailored to solve specific problems encountered in smart contract development offering powerful architectural approaches that enable dynamic creation, efficient management, and secure execution of contract instances [97], [79], [126], [12], [56].

From foundational patterns that enhance contract modularity and upgradeability to advanced patterns that facilitate complex interactions and data management, this clause covers a wide spectrum of architectural solutions.

Key areas explored include:

1) Factory patterns for dynamic contract creation
2) Proxy patterns for upgradeable contracts
3) Access control patterns for managing permissions
4) Oracle patterns for integrating external data
5) State management patterns for efficient data handling
6) Security patterns to mitigate common vulnerabilities
7) Governance patterns for decentralized decision-making
8) Interoperability patterns for cross-chain communication

By focusing on these advanced architectural patterns and their functional integrations, this clause aims to equip developers with the knowledge to design and implement sophisticated, secure, and scalable smart contract systems. It examines each pattern's structure, use cases, benefits, and potential drawbacks, providing a comprehensive understanding of when and how to apply these patterns effectively.

It also discusses how these patterns can be combined and adapted to meet the unique requirements of various blockchain applications, from Decentralized Finance (DeFi) to supply chain management and beyond. By mastering these design patterns, developers will be better prepared to create robust smart contract systems that can effectively leverage external data, manage complex access control requirements, and adapt to the evolving landscape of blockchain technology.

7.8.2 Contract Factory Pattern

The **Contract Factory Pattern** is a design pattern in smart contract development where a specialized contract, known as a factory contract, is used to create and deploy other smart contracts. This pattern allows for the efficient deployment of multiple instances of the same contract type, each with its own unique state, while adhering to a consistent and predefined logic. The factory contract acts as a central registry, simplifying the process of tracking and interacting with the deployed contracts, and can optimize gas costs by deploying numerous child contracts with lower gas costs per deployment [105], [35], [76].

[R237] PDL systems **SHALL** support the implementation of contract factory patterns for efficient deployment of multiple similar contracts.

[D640] Contract factories **SHOULD** include version control mechanisms to manage different iterations of deployed contracts.

7.8.3 Oracle Integration Pattern

The **Oracle Integration Pattern** is a design pattern in smart contract development that enables the integration of external data sources and systems with blockchain-based smart contracts. This pattern involves using oracle services as intermediaries to fetch and verify external data, which is then used to execute smart contract functions. Oracles act as bridges between the on-chain and off-chain worlds, allowing smart contracts to access real-world data while maintaining the security and integrity of the blockchain [32], [110], [76].

[R238] The smart contract architecture **SHALL** include secure mechanisms for integrating with oracle services for external data.

[D641] Multiple oracle sources **SHOULD** be used with a consensus mechanism to enhance data reliability.

7.8.4 Model-View-Controller (MVC) Pattern

The Model-View-Controller (MVC) Pattern is a software design pattern that separates an application into three interconnected components: Model, View, and Controller. The Model represents the application's data and business logic, the View displays the data to the user, and the Controller manages user input and updates the Model and View accordingly. This separation allows for easier maintenance and modification of individual components without affecting the entire system [36], [55].

Benefits: Improves maintainability, scalability, and reusability of smart contract systems.

7.8.5 Observer Pattern

The Observer Pattern is a design pattern that defines a one-to-many dependency between objects, allowing multiple observers to be notified and updated automatically when the state of a subject object changes. This pattern enables loose coupling between objects, making it easier to modify and extend systems without affecting other components. The observer pattern consists of two main components: the Subject, which maintains a list of observers and notifies them of changes, and the Observer, which registers with the subject to receive updates [38], [100], [i.26], [17], [75], [41].

Benefits: Facilitates real-time communication and interaction between different components of a decentralized application.

7.8.6 Strategy Pattern

The Strategy Pattern is a design pattern that defines a family of algorithms, encapsulates each one, and makes them interchangeable. This pattern allows smart contracts to select a strategy at runtime, enabling dynamic changes in behaviour without altering the contract's structure. By using the strategy pattern, developers can decouple the algorithmic logic from the contract's core functionality, making it easier to add new strategies or modify existing ones without affecting the entire system [135], [50], [144].

Benefits: Enhances flexibility and maintainability of smart contract systems by allowing for dynamic changes in behaviour.

7.8.7 Decorator Pattern

The Decorator Pattern is a design pattern that allows behaviour to be added to an individual object, either statically or dynamically, without affecting the behaviour of other objects from the same class. In the context of smart contracts, the decorator pattern enables the dynamic addition of new functionalities to existing contracts without altering their core structure. This pattern is particularly useful for extending the behaviour of smart contracts in a flexible and modular way, allowing developers to add new features or modify existing ones without redeploying the entire contract [143], [133], [49].

Benefits: Improves flexibility and maintainability of smart contract systems by allowing for dynamic changes in behaviour.

7.8.8 N-Tier Pattern

The N-Tier Pattern is a software architecture pattern that separates an application into multiple layers or tiers, each with a specific responsibility. This pattern allows for the physical and logical separation of presentation, business logic, and data storage, enhancing scalability, maintainability, and flexibility. The N-Tier architecture typically includes three tiers: the presentation tier (user interface), the logic tier (business rules and processing), and the data tier (storage and data management). This separation enables developers to modify or add specific tiers without affecting the entire application, improving manageability and reducing the impact of changes [136], [99].

Benefits: Enhances scalability, maintainability, and flexibility of smart contract systems.

7.8.9 Shared Repository Pattern

The Shared Repository Pattern is a design pattern that centralizes data access logic, providing a single interface for managing data interactions between multiple components or applications. This pattern separates the data access layer from the business logic, making it easier to maintain, test, and scale systems. The shared repository acts as an intermediary between the data storage and the application, encapsulating data access operations and providing a standardized way to interact with the data [98], [i.33].

Benefits: Ensures data consistency and integrity across a decentralized network.

7.8.10 Broker Pattern

The Broker Pattern is a design pattern that enables secure communication between nodes and actors in a system by acting as an intermediary. In the context of smart contracts, the broker pattern facilitates interactions between different contracts or external systems, ensuring that messages are delivered securely and reliably. This pattern is particularly useful in decentralized applications where direct communication between contracts may not be feasible or secure [132], [142].

Benefits: Facilitates secure and resilient communication and interaction between different components of a decentralized application.

7.8.11 Pipe-Filter Pattern

The Pipe-Filter Pattern is a design pattern that represents a processing pipeline in which data flow through a series of filters, with each filter transforming the data in some way. In the context of smart contracts, this pattern can be used to process and transform data in a series of steps, ensuring that each step is executed in a specific order and that the data is transformed correctly [i.26], [134].

Benefits: Ensures the authenticity of transactions and identifies potential threats in a decentralized system.

8 Smart Contracts - Applications, Solutions, and Needs

8.1 Applications

8.1.1 Introduction

Smart contracts have revolutionized various industries by automating processes and enhancing transparency and security. They are widely used in finance for Decentralized Finance (DeFi) applications, enabling peer-to-peer lending, borrowing, and trading without intermediaries. In supply chain management, smart contracts automate the tracking of goods and verification of transactions, ensuring transparency and reducing fraud. The healthcare sector utilizes smart contracts for secure patient data management and automated insurance claims processing. Additionally, smart contracts facilitate digital identity verification and voting systems in governance, providing tamper-proof and transparent solutions. These applications demonstrate how smart contracts are transforming industries by offering automated, transparent, and secure solutions that reduce reliance on intermediaries and enhance operational efficiency. As these applications grow, the need for standardized protocols and robust security measures becomes increasingly important to ensure interoperability and trust across different platforms [148].

8.1.2 Finance

In the finance sector, smart contracts are central to Decentralized Finance (DeFi) platforms, enabling peer-to-peer lending, borrowing, and trading without intermediaries. Platforms like Uniswap and Aave use smart contracts to automate liquidity provision and interest rate adjustments based on supply and demand, ensuring transparency and reducing operational costs [151].

8.1.3 Supply Chain Management

Smart contracts streamline supply chain operations by automating the tracking of goods and verifying transactions at each stage. For example, IBM Food Trust uses blockchain technology to enhance transparency and traceability in the food supply chain, allowing stakeholders to track the journey of food products from farm to table [81]. ETSI GS PDL 022 [107] discusses the use of PDL for supply chain management with an implicit reference to use of smart contracts in response to events.

8.1.4 Healthcare

In healthcare, smart contracts facilitate secure patient data management and automate insurance claims processing. Projects like MedRec leverage blockchain to create immutable records of patient data, ensuring privacy and security while allowing authorized parties access to medical records [160].

8.1.5 Real Estate

Smart contracts automate real estate transactions by handling agreements, payments, and ownership transfers without the need for intermediaries. This reduces costs and increases transaction speed while providing a transparent record of property ownership changes [162].

8.1.6 Government Services

Governments are exploring smart contracts for digital identity verification and voting systems. Estonia's e-Residency program uses blockchain technology to provide secure digital identities for citizens, enabling them to access government services online with enhanced security and transparency [106].

8.2 Solutions

8.2.1 Issues to be solved

Smart contract solutions focus on addressing challenges related to scalability, security, and interoperability. Layer-2 solutions like rollups and state channels enhance scalability by processing transactions off-chain while maintaining security through periodic on-chain settlements. Security solutions include formal verification techniques to mathematically prove the correctness of contract logic and the use of decentralized oracle networks to securely integrate external data. Interoperability is achieved through cross-chain bridges and standardized interfaces that enable seamless communication between different blockchain networks. These solutions are crucial for building robust smart contract systems that can handle complex operations efficiently [169].

Here are some examples of smart contract solutions that address challenges related to scalability, security, and interoperability:

8.2.2 Scalability

1) **Layer-2 Solutions:** Technologies like Optimistic Rollups and zk-Rollups on Ethereum improve scalability by processing transactions off-chain while maintaining security through periodic on-chain settlements. These solutions significantly increase transaction throughput and reduce costs [20].

2) **State Channels:** State channels allow multiple transactions to occur off-chain between parties and only settle the final state on-chain, reducing the load on the main blockchain and improving scalability [114].

8.2.3 Security

1) **Formal Verification:** Tools like CertiK and Securify provide formal verification of smart contracts, mathematically proving the correctness of contract logic to prevent vulnerabilities such as reentrancy attacks [58].

2) **Decentralized Oracle Networks:** Chainlink provides secure and reliable data feeds by aggregating data from multiple sources to prevent manipulation and ensure data integrity for smart contracts [2].

8.2.4 Interoperability

1) **Cross-Chain Bridges:** Solutions like Polkadot's parachains and Cosmos' Inter-Blockchain Communication (IBC) Protocol facilitate interoperability by allowing different blockchain networks to communicate and exchange assets securely [161].

2) **Atomic Swaps:** Atomic swaps enable trustless exchanges of cryptocurrencies between different blockchains without intermediaries, enhancing interoperability across blockchain ecosystems [57].

8.2.5 Smart Contracts with QoS Monitoring

Smart contracts can be used to monitor Quality of Service (QoS) in various applications such as telecommunications or cloud services. By automatically executing predefined actions when specific QoS metrics are met or violated (e.g. latency thresholds), smart contracts ensure compliance with Service Level Agreements (SLAs). This automation reduces the need for manual intervention and enhances transparency in service delivery. Implementing QoS monitoring in smart contracts requires integrating reliable data sources through oracles to provide real-time performance metrics [3], [21].

8.3 Requirements for Building a Viable System using Smart Contracts

Building a viable system with smart contracts requires addressing key requirements such as scalability, security, interoperability, and user experience. Scalability solutions like layer-2 protocols are essential for handling high transaction volumes efficiently. Security measures should include formal verification and regular audits to prevent vulnerabilities in immutable code. Interoperability is crucial for enabling seamless interaction between different blockchain networks and external systems through standardized protocols and decentralized oracles. Additionally, user experience should be prioritized by providing intuitive interfaces and clear documentation to facilitate user interaction with smart contracts [156], [67], [68], [66], [70].

9 Governance Role in Smart Contracts

9.1 Introduction to the Role of Governance in Smart Contracts

Governance of smart contracts is crucial for ensuring that these digital agreements operate smoothly, securely, and in alignment with stakeholder interests. Governance involves setting rules and policies that dictate how smart contracts are managed, updated, and terminated. This clause explores various aspects of governance, including delegation to policies, updating mechanisms, operational decision-making, contract termination, and compliance strategies.

Similar to the role of governance PDL systems, governance of smart contracts involves establishing rules and processes for managing the lifecycle of contracts, including their creation, execution, modification, and termination. It ensures that smart contracts operate within the defined parameters and adapt to changing conditions or requirements. Effective governance enhances transparency, accountability, and trust among stakeholders by providing clear protocols for decision-making and conflict resolution. In decentralized environments, governance can be embedded within the contract code or managed through Decentralized Autonomous Organizations (DAOs), allowing for community-driven oversight and control [156].

Requirements:

[R239] Governance mechanisms **SHALL** be clearly defined within the smart contract to ensure consistent application.

[R240] Stakeholders **SHALL** have access to governance processes to participate in decision-making.

Recommendations:

[D642] Developers **SHOULD** implement transparent governance models that allow for stakeholder input.

[D643] Contracts **SHOULD** include mechanisms for dispute resolution to address conflicts arising from governance decisions.

9.2 Governing the Update of a Smart Contract

Updating smart contracts is essential for incorporating new features, fixing bugs, or adapting to regulatory changes. Due to the immutable nature of blockchain, updating requires careful planning and execution. Governance plays a critical role in managing updates by defining processes for proposing changes, obtaining stakeholder approval, and implementing modifications without disrupting existing functionalities [112].

Requirements:

[R241] Smart contracts **SHALL** implement upgrade mechanisms that allow secure updates.

[R242] Update processes **SHALL** be transparent and documented for stakeholder review.

Recommendations:

[O17] Developers **MAY** use proxy patterns to facilitate contract updates.

[D644] Smart Contracts **SHOULD** include versioning systems to track changes over time.

9.3 Governing Operational Decisions

Operational decisions involve managing day-to-day activities such as executing transactions, adjusting parameters, or responding to external events. Effective governance requires clear protocols for making these decisions to ensure consistency and accountability. Decentralized Autonomous Organizations (DAOs) often use voting mechanisms or multi-signature approvals for collective decision-making [148].

Requirements:

[R243] Smart contracts **SHALL** define protocols for making operational decisions.

[R244] Decision-making processes **SHALL** involve relevant stakeholders where applicable.

Recommendations:

[D645] Developers **SHOULD** implement voting mechanisms for collective decision-making.

[D646] Contracts **SHOULD** provide audit trails of operational decisions for transparency.

9.4 Governing the Termination of a Smart Contract

Termination involves ending a contract's operations upon fulfilment of its terms or due to external factors like regulatory changes, bug fixing or security vulnerabilities. Governance is crucial in defining termination processes to ensure orderly shutdowns and asset recovery by stakeholders. Mechanisms such as self-destruct functions or governance-controlled termination facilitate this process [108].

Requirements:

[R245] Smart contracts **SHALL** include mechanisms for orderly termination.

[R246] Termination processes **SHALL** ensure asset recovery by stakeholders.

Recommendations:

[D647] Developers **SHOULD** implement self-destruct functions with safeguards against misuse.

[D648] Contracts **SHOULD** notify stakeholders prior to termination to allow asset extraction.

9.5 General Governance Compliance Strategies for Smart Contracts

Compliance strategies ensure that smart contracts adhere to relevant legal and regulatory requirements throughout their lifecycle. This includes data protection laws like GDPR and financial regulations applicable to transactions or asset management. Governance plays a role in embedding compliance measures into contract logic and maintaining audit trails for regulatory review [140], [61], [62], [65], [63], [64], [69].

Requirements:

[R247] Smart contracts **SHALL** comply with relevant legal and regulatory requirements.

[R248] Compliance measures **SHALL** be documented and auditable by regulators.

Recommendations:

[D649] Developers **SHOULD** use privacy-preserving techniques to protect sensitive data.

[D650] Contracts **SHOULD** implement audit trails for compliance verification.

10 Gas Optimization Techniques

10.1 Introduction to Gas optimization

Gas optimization techniques are essential in smart contract development to minimize transaction costs on blockchain platforms. These methods ensure efficient operation by reducing computational resource demands and, consequently (where applicable), the gas fees users incur. Effective gas optimization not only lowers transaction costs but also enhances the scalability of decentralized applications, allowing more transactions to be processed within a given block size. As smart contracts grow in complexity, efficient gas management becomes increasingly critical to their viability and adoption. Developers employ various strategies and patterns to optimize gas usage, emphasizing the importance of cost efficiency and performance in smart contract execution [31].

Requirements:

[R249] Smart contracts **SHALL** be designed with gas efficiency in mind, minimizing unnecessary computations and storage operations.

[R250] Developers **SHALL** regularly review and optimize contract code to reduce gas consumption.

Recommendations:

[D651] Developers **SHOULD** use established design patterns and tools that facilitate gas optimization.

[D652] Smart contracts **SHOULD** be tested in various scenarios to identify potential areas for gas savings.

10.2 Gas-Efficient Design Patterns

10.2.1 Introduction Gas-Efficient Design Patterns

Gas-efficient design patterns are coding practices that help reduce the computational overhead of smart contracts. These patterns include using libraries for common functions to avoid code duplication, optimizing data structures for minimal storage costs, and employing lazy loading techniques to defer computations until necessary. By adhering to these patterns, developers can create contracts that execute with lower gas consumption [92].

Requirements:

[R251] Smart contracts **SHALL** implement design patterns that minimize redundant computations and storage operations.

[R252] Contracts **SHALL** use efficient data structures to optimize storage costs.

Recommendations:

[D653] Developers **SHOULD** leverage existing libraries and frameworks that offer optimized implementations of common functionalities.

[D654] Smart Contracts **SHOULD** be modularized to enable reuse of efficient components across different projects.

10.2.2 Using Libraries for Common Functions to Avoid Code Duplication

Utilizing libraries for common functions helps avoid code duplication, which can significantly reduce the size of the contract and associated deployment costs. By abstracting frequently used functionalities into libraries, developers can ensure consistency across multiple contracts while minimizing gas consumption during execution.

Requirements:

[R253] Smart contracts **SHALL** use libraries for implementing common functionalities to reduce code duplication.

[R254] Libraries **SHALL** be thoroughly tested and audited before integration into contracts.

Recommendations:

[D655] Developers **SHOULD** leverage existing libraries like OpenZeppelin for standardized implementations.

[D656] Smart Contracts **SHOULD** modularize code into reusable components to enhance maintainability and efficiency.

10.2.3 Optimizing Data Structures for Minimal Storage Costs

Optimizing data structures involves selecting efficient storage methods that minimize the amount of data stored on-chain, thereby reducing storage costs. Techniques such as using mappings instead of arrays or compressing data into smaller units can lead to significant gas savings.

Requirements:

[R255] Smart contracts **SHALL** implement efficient data structures to minimize on-chain storage costs.

[R256] Data structures **SHALL** be chosen based on their suitability for the contract's specific use case.

Recommendations:

[D657] Developers **SHOULD** evaluate different data structures for their space-time trade-offs before implementation.

[D658] Smart Contracts **SHOULD** use packed storage where possible to optimize memory usage.

10.2.4 Employing Lazy Loading Techniques to Defer Computations Until Necessary

Lazy loading defers computations until they are explicitly needed, reducing unnecessary processing and saving gas. This technique is particularly useful in scenarios where certain data or calculations are not required immediately or in every transaction.

Requirements:

[R257] Smart contracts **SHALL** implement lazy loading techniques where applicable to defer unnecessary computations.

[R258] Deferred computations **SHALL** be managed carefully to ensure they do not affect contract logic or security.

Recommendations:

[D659] Developers **SHOULD** identify parts of the contract that can benefit from lazy loading during the design phase.

[D660] Smart Contracts **SHOULD** include mechanisms to trigger deferred computations efficiently when needed.

10.3 Managing complex operations efficiently

10.3.1 Batching

Batching reduces gas costs by consolidating multiple operations into a single transaction, which minimizes redundant state changes and transaction overheads. This technique is especially useful for managing complex operations efficiently within a single transaction context.

Requirements:

[R259] Smart contracts **SHALL** implement batching techniques where applicable to consolidate multiple operations into single transactions.

[R260] Batching processes **SHALL** ensure atomicity and consistency of operations within a batch.

Recommendations:

[D661] Developers **SHOULD** design contracts with modularity in mind to facilitate batching of related operations.

[D662] Smart Contracts **SHOULD** provide clear documentation on how batching impacts transaction execution and outcomes.

10.3.2 Proxy Patterns

Proxy patterns enable contract upgrades without redeploying the entire contract, thus saving on deployment costs while preserving state continuity. This pattern separates the logic from the data layer, allowing updates without affecting existing user interactions or data integrity [112].

Requirements:

[R261] Contracts **SHALL** use proxy patterns for upgradeability without incurring additional deployment costs.

[R262] Proxy implementations **SHALL** ensure secure delegation of calls between proxy and logic contracts.

Recommendations:

[D663] Developers **SHOULD** implement robust access controls within proxy patterns to prevent unauthorized upgrades.

[D664] Proxy patterns **SHOULD** be used judiciously to ensure security while enabling contract upgrades.

10.4 Tools for Flexible Management of Gas Expenses

10.4.1 Gas Tokens

Gas tokens are mechanisms that allow users to manage their gas expenses by pre-purchasing gas at lower prices or earning rebates through the release of unused storage space on the blockchain. By minting gas tokens when gas prices are low and redeeming them when prices are high, users can effectively reduce their transaction costs. Gas tokens work by taking advantage of Ethereum's refund mechanism, where users receive a gas refund for freeing up storage space. This approach provides flexibility in managing gas expenses, especially during periods of network congestion when gas prices spike [151].

Requirements:

[R263] Smart contracts **SHALL** be compatible with gas token standards if they are intended to leverage such mechanisms.

[R264] Contracts **SHALL** ensure that the use of gas tokens does not introduce security vulnerabilities or affect contract functionality.

Recommendations:

[D665] Developers **SHOULD** consider implementing support for gas tokens in scenarios where significant storage operations are involved.

[D666] Contracts **SHOULD** provide clear documentation on how users can benefit from using gas tokens.

10.4.2 Relayers

Relayers facilitate transactions by covering gas costs on behalf of users, often used in meta-transactions where the user does not hold native tokens but wants to interact with a contract. This tool allows users to execute transactions without needing to own or spend Ether directly, enhancing accessibility and user experience. Relayers operate by submitting transactions on behalf of users and charging a small fee for their service. This model is particularly beneficial for Decentralized Applications (DApps) targeting non-crypto-savvy users or those who want to abstract away the complexities of managing Ether for transaction fees [31].

Requirements:

[R265] Smart contracts **SHALL** support interactions with relayers where applicable to enable user-friendly transaction models.

[R266] Contracts **SHALL** implement security measures to ensure that relayer interactions do not compromise transaction integrity or user data.

Recommendations:

[D667] Developers **SHOULD** design contracts to accommodate meta-transactions and relayer services seamlessly.

[D668] Contracts **SHOULD** provide clear guidelines on how relayers can be integrated and used effectively within the DApp ecosystem.

11 Emerging Smart Contract Standards

11.1 Intro

Emerging smart contract standards are pivotal in advancing the functionality, security, and interoperability of blockchain applications. These standards aim to address the evolving needs of decentralized systems by introducing new protocols and frameworks that enhance the capabilities of smart contracts. This clause explores advanced token standards, Non-Fungible Token (NFT) standards, and innovations in smart contract wallets and account abstraction.

11.2 Advanced ERC Token Standards

Advanced Ethereum Request for Comments (ERC) token standards [113], [146] build upon the foundational ERC-20 [46] and ERC-721 [44] standards to introduce new functionalities and improve interoperability. These standards include:

- **ERC-777** [146]: An improvement over ERC-20 [46], providing advanced features like hooks, which allow tokens to notify contracts of incoming transactions, enhancing security and flexibility.
- **ERC-1155** [113]: A multi-token standard that supports both fungible and non-fungible tokens within a single contract, optimizing gas usage and simplifying complex interactions.

Requirements:

[R267] Smart contracts **SHALL** implement advanced token standards to leverage enhanced functionalities and interoperability.

[R268] Token standards **SHALL** ensure backward compatibility with existing protocols where possible.

Recommendations:

[D669] Developers **SHOULD** evaluate the specific needs of their application to choose the most appropriate token standard.

[D670] Contracts **SHOULD** include comprehensive documentation on how they implement these standards.

11.3 Non-Fungible Token Standards

Non-Fungible Tokens (NFTs) have gained significant traction for representing unique digital assets on the blockchain. The primary standard for NFTs is:

- **ERC-721** [44]: Defines a standard interface for NFTs, enabling unique asset representation with distinct ownership and metadata.

Emerging NFT standards focus on improving scalability and interoperability:

- **ERC-2981**: Introduces royalty payment mechanisms directly into NFT transactions, allowing creators to earn royalties on secondary sales [124].

Requirements:

[R269] NFT contracts **SHALL** adhere to established standards like ERC-721 [44] to ensure compatibility across platforms.

[R270] Contracts implementing royalties **SHALL** comply with emerging standards like ERC-2981 [124].

Recommendations:

[D671] Developers **SHOULD** consider using layer-2 solutions to enhance the scalability of NFT platforms.

[D672] Contracts **SHOULD** provide clear metadata structures for NFTs to facilitate integration with marketplaces and other services.

11.4 Smart Contract Wallets and Account Abstraction

Smart contract wallets and account abstraction are innovations aimed at improving user experience and security in blockchain interactions:

- **Smart Contract Wallets**: These wallets use smart contracts to manage user funds and permissions, offering features like multi-signature approvals and recovery mechanisms [88].
- **Account Abstraction**: Allows users to interact with blockchain applications without needing native cryptocurrency for transaction fees by abstracting account operations into smart contracts [22].

Requirements:

[R271] Smart contract wallets **SHALL** implement robust security measures such as multi-signature support.

[R272] Account abstraction mechanisms **SHALL** ensure seamless user experiences without compromising security.

Recommendations:

[D673] Developers **SHOULD** integrate user-friendly interfaces for smart contract wallets to enhance accessibility.

[D674] Contracts **SHOULD** provide clear documentation on account abstraction processes to educate users on their benefits.

By adopting these emerging standards, developers can create more versatile and secure smart contract applications that meet the growing demands of blockchain ecosystems while ensuring compatibility and ease of use across platforms.

12 Regulatory and Environmental Considerations

12.1 Introduction

The increasing adoption of smart contracts necessitates a thorough understanding of the regulatory and environmental implications associated with their use. This clause explores recent legislative developments, regulatory guidance, and environmental impacts related to smart contract platforms.

12.2 Recent Legislation on Smart Contract Enforceability

Recent legislative efforts have focused on clarifying the legal status and enforceability of smart contracts. Jurisdictions are increasingly recognizing smart contracts as legally binding agreements, provided they meet certain criteria such as clear terms and mutual consent [150], [152]. For instance, the state of Arizona in the United States has enacted laws affirming that smart contracts can be used for legally enforceable agreements. These legislative developments aim to provide legal certainty and encourage broader adoption of blockchain technologies.

Requirements:

[R273] Smart contracts **SHALL** comply with applicable laws and regulations to ensure their enforceability in the country of operation.

[R274] Developers **SHALL** ensure that smart contract terms are clear and unambiguous to meet legal standards.

Recommendations:

[D675] Legal professionals **SHOULD** be consulted during the development of smart contracts to ensure compliance with relevant legislation.

[D676] Developers **SHOULD** stay informed about changes in legislation affecting smart contract enforceability.

12.3 Regulatory Guidance from Financial Authorities

Financial authorities worldwide are providing guidance on the use of smart contracts within financial systems. Regulatory bodies such as the U.S. Securities and Exchange Commission (SEC) and the European Securities and Markets Authority (ESMA) have issued guidelines to ensure that smart contracts used in financial transactions comply with existing securities laws and consumer protection standards. These guidelines focus on transparency, risk management, and the prevention of fraud [176], [51].

Requirements:

[R275] Smart contracts used in financial applications **SHALL** adhere to regulations set by relevant financial authorities.

[R276] Compliance measures **SHALL** be documented and auditable by regulators.

Recommendations:

[D677] Developers **SHOULD** implement features that enhance transparency and accountability in financial smart contracts.

[D678] Regular audits **SHOULD** be conducted to ensure ongoing compliance with financial regulations.

12.4 Energy Consumption of Smart Contract Platforms

The energy consumption associated with blockchain platforms, particularly those using proof-of-work consensus mechanisms like Ethereum, has raised environmental concerns. The high computational power required for mining contributes significantly to carbon emissions [127], [137]. However, efforts are underway to transition to more energy-efficient models.

Requirements:

[R277] Developers **SHALL** consider the environmental impact when choosing a blockchain platform for deploying smart contracts.

[R278] Energy consumption metrics **SHALL** be monitored and reported for transparency.

Recommendations:

[D679] Developers **SHOULD** prioritize platforms that implement energy-efficient consensus mechanisms.

[D680] Efforts **SHOULD** be made to offset carbon emissions through renewable energy sources or carbon credits.

12.5 Proof-of-Stake and Other Eco-Friendly Consensus Mechanisms

To address environmental concerns, many blockchain platforms are transitioning from energy-intensive proof-of-work systems to more sustainable Proof-of-Stake (PoS) models or other eco-friendly consensus mechanisms like Delegated Proof-of-Stake (DPoS) or Proof-of-Authority (PoA) [121], [82]. These alternatives significantly reduce energy consumption by eliminating the need for competitive mining processes.

Requirements:

[R279] Smart contract platforms **SHALL** implement eco-friendly consensus mechanisms where feasible.

[R280] Transition plans from proof-of-work to proof-of-stake **SHALL** be clearly documented and communicated to stakeholders.

Recommendations:

[D681] Developers **SHOULD** support initiatives aimed at enhancing the sustainability of blockchain technologies.

[D682] Platforms **SHOULD** provide incentives for validators who utilize renewable energy sources.

By considering these regulatory and environmental factors, developers can create smart contract solutions that are not only legally compliant but also environmentally sustainable, contributing positively to both technological advancement and ecological preservation.

Annex A (informative): Examples from research papers used in the present document

A.1 Void

A.1.1 Void

A.1.1.1 Examples of publications for each of the solutions listed in clause 4.3.1.4 "Solutions"

A.1.1.1.1 Proxy Patterns

Publication: "Proxy Patterns for Upgradeable Smart Contracts: A Survey" (2021) Authors: Sayeed, S., & Marco-Gisbert, H. [i.22].

- This paper surveys various proxy patterns used for upgrading smart contracts, discussing their advantages and limitations.

Publication: "Upgradeable Smart Contracts: A Survey" (2021) [i.23].

- This paper provides a comprehensive survey of upgradeable smart contract patterns, including various proxy implementations. It analyses the security implications and trade-offs of different proxy approaches.

A.1.1.1.2 Data Separation

Publication: "State Separation for Dynamic Smart Contract Upgrades" (2020) Authors: Rodler, M., Li, W., Karame, G. O., & Davi, L. [i.21].

- This paper proposes a novel approach to separate contract logic from state, enabling more flexible upgrades while preserving data integrity.

Publication: "State Separation for Flexible Upgrades in Ethereum Smart Contracts" (2020) [i.9].

- This publication proposes a novel approach to separate contract logic from state, enabling more flexible upgrade mechanisms while preserving data integrity.

A.1.1.1.3 Parameterization

Publication: "Parametric Smart Contracts: A New Paradigm for Blockchain-Based Agreement" (2022) [i.25].

- The authors introduce a framework for creating highly configurable smart contracts through parameterization, allowing for greater flexibility without compromising immutability.

A.1.1.1.4 Modular Design

Publication: "A Modular Design for Ethereum Smart Contracts Verification" (2020) Authors: Li, Y., Liao, C., & Zhang, Y. [i.16].

- This paper presents a modular approach to smart contract design and verification, improving scalability and reusability.

Publication: "Modular Smart Contracts: A Compositional Approach to Contract Development" (2023) [i.13].

- This paper presents a methodology for designing smart contracts as interconnected modules, enhancing reusability and facilitating easier upgrades of specific components.

A.1.1.1.5 Thorough Testing and Auditing

Publication: "SmartBugs: A Framework to Analyze Solidity Smart Contracts" (2020) Authors: Ferreira, J. F., Cruz, P., Durieux, T., & Abreu, R. [i.7].

- This paper introduces a comprehensive framework for testing and analysing Solidity smart contracts, aiding in the detection of vulnerabilities and bugs.

Publication: "Automated Smart Contract Testing: Techniques and Challenges" (2021) [i.8].

- The publication explores advanced techniques for comprehensive smart contract testing, including formal verification methods and automated vulnerability detection tools.

A.1.1.2 Examples of publications for each of the emerging interoperability solutions listed in clause 4.3.8.7

A.1.1.2.1 Polkadot Parachains

Publication: "Polkadot: Vision for a Heterogeneous Multi-Chain Framework" (2020) Authors: Wood, G. [i.27].

- This whitepaper outlines the vision and technical details of Polkadot, including its parachain architecture for interoperable blockchain networks.

A.1.1.2.2 Cosmos Inter-Blockchain Communication (IBC)

Publication: "Inter-Blockchain Communication Protocol: An Overview" (2021) Authors: Kwon, J., & Buchman, E. [85].

- This paper provides an in-depth overview of the Inter-Blockchain Communication protocol used in the Cosmos ecosystem for enabling communication between independent blockchains.

A.1.1.2.3 Ethereum Layer-2 Solutions

Publication: "Rollups on Trial: Optimistic vs. Zero-Knowledge" (2021) Authors: John, K., Yin, H., Ektefa, M., & Sakurai, K. [i.12].

- This paper compares and analyses two prominent Ethereum Layer-2 scaling solutions: Optimistic Rollups and Zero-Knowledge Rollups, discussing their mechanisms, advantages, and challenges.

Publication: "Optimistic Rollups: The Present and Future of Ethereum Scaling" (2022) [i.1].

- This publication explores Optimistic Rollups as a Layer-2 scaling solution for Ethereum. It discusses how these rollups interact with the main Ethereum network, providing increased transaction throughput while maintaining security guarantees.

A.1.1.2.4 Additional relevant publications:

A.1.1.2.4.1 Cross-Chain Bridges:

Publication: Zamyatin A., Al-Bassam M., Zindros D., Kokoris-Kogias E., Moreno-Sanchez P., Kiayias A. & Knottenbelt W. J. (2021): "SoK: Communication across distributed ledgers", [161].

- This systematic overview covers various cross-chain communication protocols, including bridges, providing a comprehensive analysis of their security and design principles.

A.1.1.2.4.2 Interoperability Protocols:

Publication: "Interoperability between Blockchain Systems" (2020) Authors: Belchior, R., Vasconcelos, A., Guerreiro, S., & Correia, M. [14].

- This survey paper provides a comprehensive overview of blockchain interoperability solutions, including protocols, architectures, and frameworks for enabling cross-chain interactions.

A.1.1.3 Examples of publications related to the tools and techniques listed in clause 4.3.9.6

A.1.1.3.1 Static Analysis Tools

Publication: "SmartBugs: A Framework to Analyze Solidity Smart Contracts" (2020) [i.6].

- This paper introduces SmartBugs, an execution framework for analysing Ethereum smart contracts using multiple static analysis tools. It provides a comprehensive evaluation of existing tools and their effectiveness in detecting various vulnerabilities.

Publication: "Slither: A Static Analysis Framework For Smart Contracts" (2020) Authors: Feist, J., Grieco, G., & Groce, A. [i.4].

- This paper presents Slither, a static analysis framework for Ethereum smart contracts. It can detect various vulnerabilities, optimize gas usage, and provide insights into code quality.

A.1.1.3.2 Dynamic Analysis

Publication: "ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection" (2021) [i.11].

- This publication presents ContractFuzzer, a novel fuzzing framework specifically designed for Ethereum smart contracts. It demonstrates how dynamic analysis through fuzzing can effectively uncover vulnerabilities that might be missed by static analysis alone.

Publication: "ETHPLOIT: From Fuzzing to Efficient Exploit Generation against Smart Contracts" (2020) Authors: Feng, Y., Torlak, E., & Bodík, R. [i.5].

- This paper introduces ETHPLOIT, a dynamic analysis tool that combines fuzzing with symbolic execution to generate exploits for vulnerabilities in Ethereum smart contracts.

A.1.1.3.3 Fuzzing

Publication: "EthPloit: From Fuzzing to Efficient Exploit Generation against Smart Contracts" (2022) [i.10].

- The authors introduce EthPloit, an advanced fuzzing tool that not only detects vulnerabilities but also generates exploits for Ethereum smart contracts. This work showcases the evolution of fuzzing techniques in the smart contract security domain.

Publication: "sFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts" (2020) Authors: Nguyen, T. D., Pham, L. H., Sun, J., Lin, Y., & Minh, Q. T. [i.17].

- This paper presents sFuzz, an adaptive fuzzer for Solidity smart contracts that uses a novel algorithm to generate test inputs efficiently.

A.1.1.3.4 Formal Verification Tools

Publication: "VERISOL: A Formal Verifier for Solidity Smart Contracts" (2020) [i.15].

- This paper presents VERISOL, a formal verification tool for Solidity smart contracts. It demonstrates how formal methods can be applied to prove correctness properties of smart contracts, providing a higher level of assurance than traditional testing methods.

Publication: "VeriSmart: A Highly Precise Safety Verifier for Ethereum Smart Contracts" (2020) Authors: So, S., Lee, M., Park, J., Lee, H., & Oh, H. [i.24].

- This paper introduces VeriSmart, a formal verification tool for Ethereum smart contracts that can prove the absence of vulnerabilities with high precision.

A.1.1.3.5 Security Frameworks

Example: "OpenZeppelin Contracts: An Open Framework of Reusable and Secure Smart Contracts" (2023) [i.18].

- While not a traditional academic publication, the OpenZeppelin Contracts library documentation and its associated technical papers provide a comprehensive overview of a widely-used security framework for smart contract development. The 2023 version includes significant updates and new security features.

Publication: "SmartShield: Automatic Smart Contract Protection Made Easy" (2021) [i.28].

- This paper presents SmartShield, a comprehensive security framework for smart contracts that combines multiple analysis techniques to detect and mitigate vulnerabilities automatically.

A.1.1.4 Examples of solutions for storing large amounts of data on-chain, as mentioned in clause 4.4.8.1

A.1.1.4.1 Layer-2 Solutions:

Example: "Rollups on Ethereum: A Comprehensive Survey" (2022) [i.14].

- This paper provides an overview of various Layer-2 scaling solutions, focusing on rollups and their impact on data storage scalability.

A.1.1.4.2 Sharding:

Example: "A Sharding-Based Blockchain Protocol. Proceedings of the ACM Conference on Advances in Cryptographic Technology" (2021) [i.2].

- This paper presents a Sharding-Based Blockchain Protocol using sharding and Practical Bysantine Fault Tolerance (PBFT) protocol demonstrating high efficiency and scalability.

A.1.1.4.3 Off-chain Storage with On-chain Verification:

Example: "Decentralized Storage: The Backbone of the Blockchain" (2020) [i.29].

- This paper explores the integration of decentralized storage solutions with blockchain, discussing how to maintain data integrity while reducing on-chain storage requirements.

A.1.1.4.4 Data Compression Techniques:

Example: "Efficient Data Compression for Blockchain Storage Optimization" (2023) [i.19].

- This study presents novel compression techniques specifically designed for blockchain data, aiming to reduce storage requirements without compromising data integrity.

A.1.1.4.5 State Channels:

Example: "State Channels: An Overview and State of the Art" (2021) [i.3].

- This publication provides a comprehensive review of state channel technology, including its application in reducing on-chain data storage by moving transactions off-chain.

Annex B (informative): Bibliography

- Bartoletti M.: "Smart Contracts Contracts", Frontiers in Blockchain, 2020, 3.
- Nguyen C. T., Hoang D. T., Nguyen D. N., Niyato D., Nguyen H. T. & Dutkiewicz E. (2020): "Proof-of-stake consensus mechanisms for future blockchain networks: fundamentals, applications and opportunities", IEEE™ Access, 7, 85727-85745.
- Perez D. & Livshits B. (2021): "Smart contract vulnerabilities: Vulnerable does not imply exploited", In 30th USENIX Security Symposium (USENIX Security 21) (pp. 851-868).

History

Document history		
V1.1.1	June 2025	Publication