POLITECNICO DI TORINO

Master's Degree
in Computer Engineering

Master's Degree Thesis

A Blockchain-Based DApp for Integrated Building Maintenance Management

Supervisors
Prof.ssa Valentina Gatteschi
Prof. Marco Domaneschi
Prof.ssa Valentina Villa

Candidate
Filippo Restori

Academic Year 2024-2025

Summary

In building maintenance processes there is often the presence of fragmented data, the absence of traceability, and centralized processes in building maintenance mechanisms are the major drawbacks to transparency and efficient operations. In this thesis, the challenges are overcome by presenting and developing a decentralized building maintenance management system, combining Building Information Modeling (BIM) and blockchain technology. The proposed solution is a decentralized application (DApp) for facility maintenance that enables fault reporting, asset registration, and technician coordination through Ethereum-based smart contracts. A 3D BIM model viewer (implemented with the IFC.js library and React) provides an interactive interface for users to visualize building assets and report issues in context, while smart contract logic written in Solidity manages maintenance tasks and enforces role-based access via on-chain rules. Users interact with the system using MetaMask for authentication and transaction signing, and the application is deployed on an Ethereum-compatible network (Polygon) to leverage faster confirmations and significantly lower transaction fees. The system's architecture comprises a React front end with an embedded BIM model, a set of Solidity smart contracts for core maintenance workflows (asset registry, fault logs, job assignment, status updates), and an Express.js server with a SQLite database to handle off-chain metadata and auxiliary functions such as logging gas costs for reimbursement. Performance evaluation on Ethereum Sepolia versus the Polygon testnet demonstrates that gas usage remains consistent across networks, but Polygon dramatically improves cost and speed: the same transactions that cost about $0.88 and take 13-15 seconds to fully confirm on Sepolia cost roughly $0.02 on Polygon with confirmations in a few seconds. Preliminary usability testing with sample end-users (occupant, technician, administrator roles) confirms the DApp's effectiveness: each role could successfully use the role-specific features (e.g. occupants submitting fault reports, technicians claiming and resolving tasks, admins overseeing payments and records), and access control was correctly enforced, while user feedback highlighted minor interface improvements needed (such as clearer guidance when switching networks in MetaMask). Overall, by showing that BIM and blockchain are indeed compatible and that the former can greatly benefit the latter, this work will contribute a secure and transparent framework to the smart building technologies and offer a basis of further improvements, like the integration of predictive maintenance or alignment with the upcoming standards of digital building logbooks.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The management of building maintenance has been an *intensive labor process* in the past because it involved often *fragmented documentation*, *repetitive paper trails*, and *slow communication* between different stakeholders. Managers, technicians, and building owners often face challenges in tracking the status of assets, scheduling repairs, and verifying completed tasks in a timely manner. These problems are especially present in larger and complex buildings, where even small errors can lead to higher costs and inefficiency. In our era of rapid digitization, *manual approaches are obsolete*, so there is a need of more **transparent** and **automated systems**.

Recent advances in *Building Information Modeling (BIM)* and *blockchain* technology presented new opportunities to improve building maintenance processes. BIM technology creates a *digital representation* that contains facility design along with operational characteristics, allowing stakeholders to visualize and manage assets with greater precision. Meanwhile, blockchain can also be used as a *secure technology, distributed ledger* for recording transactions and verifying data integrity. By merging BIM with blockchain is it possible to develop **decentralized applications (DApps)** to store and track events in an **immutable** way in the ledger, ensuring fault reports, repairs, and maintenance actions are **tamper-proof** and transparently logged.

However, introducing blockchain into a maintenance scenario isn't that simple. One known challenge that the system will face is the cost of on-chain transaction, commonly referred to as *gas fees*. In a standard maintenance flow, this would lead to a creation of twos on-chain submission, *doubling the operational costs*. Small-scale projects along side those with limited budgets may avoid implementing the system because of the additional burden it imposes. Responding to this, a gas fees **reimbursement system** can be implemented off-chain to track and log actual cost incurred by the user, avoiding repetitive charges and simplifying the entire process. This model demonstrate how decentralized solution can be made more efficient.

Another focus of this thesis is an *intuitive system* for user to interact. For decades, building maintenance software has suffered from confusing interfaces. The use of *React* and *Three.js* frameworks enables the development of an easy to use **3D space** which offers stakeholders a clear method to examine and work with building components. In this way, an otherwise abstract concept of a "faulty lamp on the third floor" is represented in an interactive model, considerably improving usability and situational awareness. Importantly, **role-based** access control in the DApp ensures that **only authorized actors** can execute certain actions, reducing both confusion and security risks.

Given these challenges and opportunities, this thesis aim to **design, implement and evaluate** a blockchain based decentralized application for building maintenance management that integrates BIM visualization with blockchain. Through a combination of smart contracts (written in *Solidity*), *off-chain data management* (handled via an Express server and SQLite database), and *3D model integration* (using Three.js within a React front end), the proposed system aims to improve **transparency**, **reduce overhead costs**, and **optimize coordination** among all stakeholders involved in maintenance tasks. The final evaluation will compare this blockchain-centric approach with conventional maintenance workflows, focusing on metrics such as **cost effectiveness**, **transaction speed**, and **user satisfaction**. The hope is that these insights will establish a realistic use of blockchain's strengths in an industry not typically associated with decentralized technologies, all while keeping day-to-day operational costs in check.

1.1 Background: Maintenance Management in Smart Buildings and Its Challenges

Maintenance management includes all activities involved in keeping building systems operational, from routine inspections and repairs to major overhauls. Efficient maintenance ensures that systems (e.g. HVAC, lamp, elevator) run optimally, electrical and plumbing systems remain safe, and structural components are preserved. Practical applications expose *facility managers* to multiple *issues* during the coordination of maintenance work. One major issue is **data fragmentation**. Information needed for maintenance is often *spread across disparate documents and systems*, including original building plans, operation manuals, maintenance logs, and vendor contracts. These pieces of information are rarely integrated, as a result, a manager might have to consult multiple sources to understand what assets exist in a building, their maintenance history, and their current condition. The *handover from construction* to operations is a known point of data loss, an industry analysis found that of *all data generated during design and construction, 95% is not used effectively during operations, essentially going to waste after handover* [Mergenschroer and Lipsey [2024]]. This implies that building owners and facility managers often encounter difficulties accessing complete data about their properties at the beginning which creates circumstances that lead to inefficient maintenance practices.

The fragmentation of maintenance data is closely linked to **inefficiencies** and **mistakes** in *facility management.* When information about building equipment and past maintenance is not readily available or is inconsistent, maintenance personnel may *spend excessive time searching for data* or *reinventing solutions* to known problems. Inefficient workflows are common, for instance, imagine a maintenance technician who discovers a faulty sensor in a smart building. If the building's documentation is not centralized, the technician might not easily find the sensor's specifications or warranty information, causing delays. Such delays compound across many tasks. Recent industry surveys underscore the severity of the *issue*: **62% of facilities managers report problems with data centralization, integration, and communication**, which indicates that a majority struggle with siloed systems that don't "talk" to each other. Miscommunication between stakeholders (managers, technicians, contractors, etc.) is another symptom of this fragmentation. Nearly **48% of delays in work order execution are attributed to miscommunication among parties** [Infraspeak [2025]]. In other words, almost *half of maintenance delays* could potentially be *avoided* with better information flow and coordination.

Another challenge in current maintenance management is the **insufficient trasparency** and **lack of accountability** measures. Building maintenance often involves **multiple stakeholders**: the property owner, the facility management team, external service contractors, equipment suppliers, and sometimes regulatory bodies. Each stakeholder may maintain their own records of maintenance activities. These records are typically private and easily modified, which can lead to disputes or uncertainties. For example, a building owner might question whether a service contractor actually performed all the preventive maintenance tasks they were contracted to do last year, if the records are paper-based or kept in an internal database, they could be altered after the fact or may not be easily shareable for verification. This opaqueness can erode trust. It also makes it difficult to enforce service level agreements (SLAs) or compliance with safety regulations, since there is no immutable **"source of truth"** about what maintenance has been done and when. The need for transparency has been highlighted in safety-critical contexts. For instance, after incidents like the Grenfell Tower fire (2017) regulators have stressed having a **"Golden Thread" of information**, a complete, trustworthy record of all building information and changes over time. In maintenance terms, this means having a *tamper-proof log of inspections*, *repairs*, and *system performance data.* Transparency becomes essential to achieve accountability because hidden issues along with missed performance opportunities stem from unreliable information.

Compounding these issues is the sheer scale and complexity of modern smart buildings. **Smart buildings are equipped with numerous IoT sensors** and **automated systems** that generate real-time data on everything from temperature and air quality to elevator performance and room occupancy. In theory, this wealth of data should enable **data-driven maintenance**, for example, sensors could detect anomalies in equipment operation and trigger maintenance requests automatically. In practice, however, if the data streams from IoT devices are not integrated into maintenance management platforms, they risk becoming yet another silo (isolated repository of data or functionality that doesn't communicate or integrate with other systems). Many existing Computerized Maintenance Management Systems (CMMS) or other facility management software in use were not designed with IoT and advanced analytics in mind, or they operate in isolation from the building's BIM (if one exists). The result is that the opportunity to shift from reactive maintenance to proactive or predictive maintenance is often missed. Even when data is available, it may not be trusted or utilized across organizations. A maintenance manager might hesitate to rely on data they cannot verify or easily share with others, and a building owner might be reluctant to invest in IoT upgrades without assurance of data integrity and clear ROI.

Most maintenance issues arise from **fragmented data systems**, **inefficient work processes** and **unclear organizational information** leading to poor maintenance results and resource misallocation. It becomes evident through research that **93% of organizations identify their maintenance processes as inefficient** [**Infraspeak** [**2024**]]. In buildings, inefficient maintenance can lead to higher life-cycle costs, increased downtime of critical systems, discomfort for occupants, and even safety risks. When equipment histories are incomplete or trust in data is low, preventive maintenance might be skipped or improperly done, leading to breakdowns. When communication is lacking, two different teams might unknowingly service the same issue, or worse, assume each other has handled it and leave an issue unresolved. Clearly, there is a pressing need to **modernize building maintenance management**. To address the current data fragmentation the perfect *solution* should *link separated data systems while simultaneously establishing automated document management and trackable maintenance documentation.* This is where emerging digital technologies like BIM and blockchain become highly relevant. They offer the ingredients to tackle the very pain points outlined above, potentially enabling a new paradigm of maintenance management fitting the era of smart buildings.

1.2 BIM and Blockchain Technologies: Relevance to the Problem

Building Information Modeling **(BIM)** and **blockchain technology** have each revolutionized their respective domains, BIM in the architecture, engineering, and construction (AEC) industry, and blockchain initially in finance (through cryptocurrencies) but now across many fields requiring secure data sharing. While they originate from different worlds, both technologies deal fundamentally with **information management**, and together they hold particular promise for facility maintenance in smart buildings.

BIM is essentially a *digital foundation for building information.* It involves creating comprehensive 3D models of buildings enriched with data about every element, structural components, mechanical systems, spaces, and even operational details. A BIM model is often described as a centralized repository of building data or a digital twin of the physical asset. Unlike traditional blueprints or CAD drawings, a BIM model can contain functional information such as the specifications of a unit, the maintenance schedule for a generator, and the warranty details of a window system, all linked to their location in the 3D space. By its very nature, BIM is *designed to integrate data that would otherwise be scattered.* As one source puts it, "one of the standout advantages of BIM is its ability to **centralize** and **organize large volumes of building data** in a highly accessible way," whereas in traditional facility management, important information is often scattered and leads to inefficiencies and mistakes [bimcommunity [2025]]. In a BIM-centric approach,

all stakeholders work off the same dataset, which greatly reduces inconsistencies. For maintenance management, this means a technician or manager using a BIM-based system could click on an asset (say a pump or a fire alarm device) within a digital model and immediately retrieve its history, manuals, and pending work orders. BIM thus offers a solution to the problem of fragmented data: it strives to **ensure that every piece of information about the building is "under one roof" digitally**.

The use of BIM in the **operations and maintenance phase** of a building's life (often called BIM for Facilities Management or BIM for FM) is an area of growing interest. Traditionally, BIM has been used mainly in design and construction. However, as noted earlier, there is often a drop-off in data utilization after handover. For example, owners might receive a BIM model at project close-out, but it may not be kept up to date or actively used in day-to-day operations. This represents a missed opportunity. The industry is aware of this gap: owners and facility managers often express that they **struggle with poor and fragmented data** despite having plenty of data available [[Mergenschroer and Lipsey [2024]]. By actively using BIM during the maintenance phase, facility managers can have **complete** and **consistent** information, preventing the loss of knowledge that typically occurs at handover. Furthermore, BIM enables better visualization and planning of maintenance activities. For instance, scheduling a complex repair in a hospital or a high-rise can be improved by visualizing the affected areas and systems in the BIM model, coordinating access, and simulating the work to avoid clashes with other ongoing tasks. The maintenance management problem benefits from BIM because it functions as an **information integration platform** which addresses the current practice deficiencies of data silos and inefficiencies.

Blockchain technology, in contrast, **addresses the dimension of data integrity, transparency, and decentralization**. A blockchain is a distributed ledger maintained by a network of computers (nodes) that collectively verify and record transactions in blocks, which are cryptographically linked (hence forming a chain). Once information is recorded on a blockchain and confirmed, it becomes extremely difficult to alter retroactively. This immutability, combined with the fact that the ledger is shared among multiple parties, means that blockchain works as an authoritative trusth which all parties agree on. The implementation of *blockchain* in maintenance management **creates secure storage for maintenance records along with sensor readings and work orders** because the system prevents any unauthorized alterations. If multiple stakeholders (owner, maintenance contractor, equipment vendor, etc.) are part of a blockchain network for a building, they can all **access the same verified records in real time** without relying on a single central authority. *Blockchain effectively provides a "distributed, single source of shared truth," functioning as the top record system for transactions [intellis].* By integrating this capability it establishes digital trust between parties which might not fully trust each other's internal records [intellis]. Handing facility managers this capability allows them to verify that maintenance activities occurred at specified times alongside unalterable sensor readings which cannot be altered for hiding negligence or errors.

The **transparency and security features of blockchain** directly tackle the earlier mentioned issue of lack of accountability. With a blockchain-based maintenance log, **each maintenance activity** (like an **inspection**, a **repair**, or a **part replacement**) could be **recorded as a transaction**. Each transaction could include details like who performed the work, when, on which component (perhaps referencing an ID from the BIM model), and what the outcome was. Because these transactions are confirmed by the network and cryptographically secured, once they are in the ledger they become a permanent part of the building's history. If an auditor or regulator wants to verify compliance, they can be given access to the blockchain records. For example, a new regulation might require that **critical safety systems have certified maintenance every year; using blockchain, an inspector could immediately see the verifiable timestamped records of those maintenance events**. Buildings can maintain a *tamper-proof blockchain ledger for tracking inspection reports and maintenance records to make owners and managers accountable according to an industry perspective [Oulton [2023]]*. In essence, blockchain creates a trusted environment for collaboration: even if multiple companies are involved in maintaining a building, none can unilaterally manipulate the records to their advantage, and all can trust the authenticity of the data. Additionally, blockchain can enable **smart contracts**, which are self-executing agreements coded on the blockchain. In maintenance, smart contracts might be used to automatically trigger payments when a job is completed and verified, or to send alerts if scheduled maintenance does not occur by a deadline, thereby automating compliance with service agreements.

By integrating blockchain, maintenance management can also be extended beyond a single organization's boundary. Consider a scenario with **equipment manufacturers** providing warranty service. If maintenance logs are on a blockchain, a manufacturer could verify whether required maintenance was performed (to honor a warranty claim) without needing to trust potentially biased reports from the owner. Or tenants in a smart building could be given confidence about indoor air quality maintenance by viewing an immutable log of filter replacements. These are new possibilities that a conventional centralized database could not offer with the same level of credibility. The **data-integrity protection** combined with **visibility** for authorized parties offered by blockchain technology complements BIM's role of capturing and structuring the data.

The synergy of BIM and blockchain in the maintenance context is particularly powerful. BIM solves the "where and what" – it knows the building's components, their properties, and it can store rich information about them. Blockchain solves the "when and who" – it provides a reliable record of events (like maintenance tasks or sensor triggers) and who performed or initiated them, in chronological order. When combined, one can envision a system where **the BIM model serves as the user interface and data environment**, while **blockchain runs under the hood to notarize every piece of maintenance data**. For example, a maintenance technician could navigate the BIM model of a smart building, click on an air handling unit that needs servicing, update the maintenance performed in the BIM-linked system, and that update would simultaneously be recorded to the blockchain ledger. Later, anyone with access can see in the BIM model that the unit was serviced on a certain date, and they can trust that record because the blockchain entry confirms it. If a dispute arises about whether a task was done or whether a sensor reading was correctly logged, the blockchain provides an audit trail. In technical terms, BIM data could includes references to blockchain transactions through hashes or IDs thus uniting detailed building information with **trustable event logs**.

By leveraging blockchain, BIM's data becomes more **credible and usable among a wider network of stakeholders**. Conversely, by leveraging BIM, the data on the blockchain is given context and structure (instead of being isolated transactions, they are tied to real-world building elements and systems). This integrated approach can address the **existing gaps** in maintenance management: it can ensure *data completeness and accessibility* (through BIM) while also ensuring *data reliability and transparency* (through blockchain). Such integration serves the requirements of **smart buildings** which need big data management from BIM and trusted automation delivered by blockchain smart contracts. In summary, **BIM and blockchain together provide a complementary solution** to the challenges outlined in Section 1.1. This thesis investigates the integration assessment to prove how a **blockchain-maintained management system which connects with BIM** produces substantial improvements in smart building maintenance efficiency and visibility.

1.3 Problem Statement and Research Motivation

Despite benefits that BIM and blockchain promise, their combined application in real-world building maintenance management remains nascent. Most facilities today do not use BIM for daily maintenance work, and very few (if any) have incorporated blockchain for maintenance record-keeping. The **core problem** addressed in this thesis can be stated as follows:

Building maintenance management in smart buildings suffers from fragmented information and lack of trust among stakeholders, leading to inefficiencies and suboptimal decisions. There is currently no widely adopted solution that integrates a building's digital data (BIM) with a secure, decentralized platform (blockchain) to manage maintenance activities. This gap results in missed opportunities for automation, increased transparency, and collaboration in the maintenance process.

In simpler terms, **facility managers lack a unified and trustworthy system** to manage maintenance in complex, sensor-rich buildings. Traditional maintenance software may track work orders, but they often do not integrate with the building's BIM data (if it exists) and usually operate as centralized systems that stakeholders outside the organization cannot access. On the other hand, while blockchain has been proposed in various research works as a way to create tamper-proof logs, **there is a gap in how to practically connect those logs to the detailed technical data of the building and its equipment**. Without that connection, a blockchain by itself would be of limited use in maintenance and it could become just an isolated ledger of transactions, lacking the rich context needed to plan and execute maintenance tasks effectively. The *motivation* for this research is to bridge that gap by combining the contextual power of BIM with the trust mechanism of blockchain into a single system tailored for maintenance management.

The motivation is further strengthened by observing the **state of industry and research**: Owners and facility management firms are increasingly interested in digital twins and smart maintenance platforms, yet they struggle with interoperability and data silos. Meanwhile, the concept of using blockchain in the construction and real estate sector is gaining traction for things like supply chain transparency and contract management, but its application in the operations phase (post-construction) is still limited. This thesis argues that **integrating BIM with a blockchain-based decentralized application** can address key points such as data loss at handover, siloed maintenance databases, and disputes over service delivery. By providing a *proof-of-concept implementation* and analysis of such an integrated system, the work aims to show a viable path forward for industry practitioners and add to the academic knowledge base on novel facility management solutions.

In addition, modern buildings increasingly demand **real-time and resilient maintenance management**. Imagine a smart building where thousands of sensor readings and maintenance events occur daily, a centralized system might become a bottleneck or single point of failure, whereas a blockchain-backed decentralized system could inherently be more resilient (since multiple nodes maintain the data). Moreover, as **cybersecurity** and **data ownership** become concerns (who owns the data coming from a building and how it's used), a blockchain approach can allow controlled sharing of maintenance data without giving one party unilateral control. All these factors contribute to why this research is both timely and important. In summary, the problem this thesis addresses is the lack of a cohesive, secure information framework for maintenance management, and the motivation is to harness BIM and blockchain together to fill this void and improving efficiency, accountability, and ultimately prolonging the life and performance of smart building assets.

1.4 Research Goals, Scope, and Assumptions

Given the problem statement and motivation above, the **primary goal of this thesis** is to *develop* and *evaluate* a **blockchain-based maintenance management system integrated with BIM for smart buildings**. This high-level goal can be broken down into a few concrete objectives:

- **Design a System Architecture**: The thesis will propose an architecture for how BIM and blockchain can work together in the context of maintenance management. This includes defining how building data and maintenance events flow between a BIM environment (which could be a BIM model and a database for building information) and a blockchain network. Key questions to address are what information to store on-chain vs off-chain, how to link blockchain transactions with BIM elements, and how users (e.g., facility managers, technicians, users) will interact with the system.
- **Implement a Decentralized Application (DApp) Prototype**: To demonstrate

the feasibility of the architecture, a prototype decentralized application will be built. This application is designed to allow users to record maintenance activities via blockchain (using smart contracts) while retrieving and updating data in the BIM model. The implementation will utilize an existing blockchain platform (the candidates are Polygon and Ethereum), a BIM software and database. The prototype will operate in a **simulated smart building environment**, meaning that IoT sensor inputs and building assets will be represented in a controlled setting (not a live deployment in an actual building). This approach is chosen to focus on the technical integration and to allow testing various scenarios without the unpredictability of a real building.

- **Demonstrate and Evaluate the System**: The thesis will simulate typical maintenance management scenarios to test how the integrated BIM-blockchain system performs. For example, scenarios may include a routine maintenance task being logged, an IoT sensor automatically flagging an anomaly and creating a maintenance request, or a technician updating a task completion which triggers a smart contract for payment. The evaluation will consider **functional criteria** (does the system improve data accessibility? can multiple parties successfully share information?) as well as **performance criteria** (is the system responsive and scalable enough for practical use?). It will also evaluate **qualitative benefits** such as transparency (e.g., demonstrating that once a task is logged on the blockchain, it is visible and verifiable by all authorized stakeholders via the BIM interface).
- **Address Security and Privacy Considerations**: The research will define what **assumptions** are made (for instance, assuming the blockchain is secure and that participants' identities are managed properly) and what potential vulnerabilities need to be mitigated. Issues like data privacy will be considered, maintenance logs can contain sensitive information about a building, so the system must ensure that only authorized parties can access certain data, even though the ledger is distributed. The scope includes proposing measures such as permissioned blockchain networks or encryption techniques if needed to protect sensitive maintenance information.

The **scope** of the thesis is focused on the technical implementation and demonstration of the BIM-blockchain integrated system for maintenance management. It is important to clarify what is **within scope and what is outside**. This research will focus on a **proof-of-concept** system rather than a full production ready solution. That means some simplifications are made: for example, IoT sensor data will be **simulated** rather than collected from a physical building's sensors. This assumption allows us to test how sensor which triggers maintenance events can be handled on blockchain without needing to deploy hardware. Similarly, the BIM model used in the prototype might be a subset of a building (or a hypothetical building) containing enough complexity (some assets to test the system) to illustrate the concept, rather than an entire real building with all details.

Another boundary of the scope is that the thesis will concentrate on the **information management** and **software** architecture aspects, rather than on mechanical or structural engineering aspects of maintenance. In other words, the value of integrating BIM and blockchain will be analyzed in terms of data management (efficiency, transparency, collaboration), not in terms of, say, improving the physical process of replacing an lamp or HVAC unit. Likewise, economic analysis (like cost benefit over a building's life) is not the primary focus, although the thesis will qualitatively discuss expected benefits. The assumption is that if the system improves information flow and trust, it will indirectly lead to cost savings as evidenced by industry reports; quantifying those savings precisely is beyond the scope of this work.

The system is also scoped to a **certain scale**, for instance, the prototype might assume a single building managed on one blockchain network. Scaling to a city wide platform or integration with smart city infrastructure is left for future exploration. Additionally, while blockchain enables decentralization, in a facility management context we usually have a consortium of known parties (owner, contractors, etc.). The project will assume a **permissioned blockchain model** where participants are vetted (as opposed to completely open public blockchain), since this reflects real-world scenarios of a private building's stakeholders. This assumption simplifies identity management and aligns with likely deployment models if such a system were adopted in industry.

It will not deploy in a real operational building, nor will it address every possible feature of a commercial maintenance management system (such as complex resource scheduling algorithms, integration with financial systems, etc.). However, it will address core features needed to prove the concept: recording maintenance tasks immutably, querying asset information from BIM, multi-user access, and basic automation via smart contracts. The security analysis will be inside the scope to ensure the proposed solution is robust against common threats (data tampering, unauthorized access), reinforcing the trust objective of the system.

1.5 Thesis Outline

This thesis is organized into seven chapters, which build on each other to cover the background, development, and implications of the proposed BIM-blockchain integrated maintenance management system:

- **Chapter 2: Literature Review**, The next chapter surveys relevant literature and existing work in the domains of building maintenance management, BIM in facilities management, and blockchain applications in construction and facility operations. It identifies what technologies and methodologies have been previously explored (for example, past attempts to use BIM for maintenance, or case studies of blockchain in asset management) and highlights the gap that this thesis aims to fill. The literature review establishes the theoretical foundation and justifies why combining BIM and blockchain is a novel and worthwhile approach. It will also cover detailed definitions for BIM, blockchain, smart contracts and digital twins among other concepts as foundational knowledge before moving into later chapters.

- **Chapter 3: Proposed Architecture**, In this chapter, the thesis presents the design of the decentralized application framework that integrates BIM and blockchain for maintenance management. The architecture is described in detail, including system components, their interactions, and data flows. For instance, the chapter will show how a maintenance request moves from an IoT sensor or user input into the blockchain, and how the BIM model gets updated with information from the blockchain. The use of smart contracts to automate certain processes (like verification of task completion) will be explained. Diagrams and models (such as UML diagrams or system architecture diagrams) are provided to illustrate the proposed system. Chapter 3 essentially serves as the blueprint for the implementation.

- **Chapter 4: Implementation**, This chapter describes the development of the prototype system based on the proposed architecture. It discusses the tools, platforms, and programming languages used. The implementation chapter walks through how key features were realized: how we achieved linking a BIM element to a blockchain record, how users interact with the system (perhaps via a web interface), and how we simulated the smart building environment. Challenges encountered during development and how they were overcome are also documented. By the end of Chapter 4, the reader should have a clear understanding of how the conceptual design was turned into a working prototype.

- **Chapter 5: Evaluation**, Once the system is built, it needs to be evaluated to validate that it meets the goals. Chapter 5 describes the scenarios and test cases used to evaluate the prototype. The chapter then discusses the results: Did the system improve the speed of information retrieval? Is the data consistency maintained across BIM and blockchain? How do stakeholders benefit in terms of transparency? This chapter may include performance metrics (like transaction times on the blockchain, or the latency of retrieving data) and discuss whether these are within acceptable bounds for practical use. It also critically examines any limitations observed, for instance, if certain operations were too slow or if some types of data were too large to store on the blockchain, etc. The evaluation ties back to the research objectives, assessing to what extent each objective was achieved.

- **Chapter 6: Conclusion and Future Work**, The concluding chapter provides a summary of thesis contributions together with its documented results. The integration of BIM and blockchain shows promise in finding solutions for the initial problem of fragmented and untrusted maintenance data. The chapter highlights the key benefits demonstrated, such as improved data accessibility, enhanced transparency, and potential for more proactive maintenance workflows. The current work's limitations include prototype scale restrictions and any assumptions which were necessary while future research potentials are explored for development. Future work might include testing the system in a live building, extending it to incorporate machine learning for predictive analytics, or exploring integration with other systems (like building management systems or facility management enterprise software). The thesis ends with closing thoughts on how decentralized, BIM-informed maintenance management could fit into the broader evolution of smart cities and digital building lifecycle management.

In summary, Chapter 1 has set the stage by explaining the **context, problems, and proposed direction** of using a blockchain-based decentralized application integrated with BIM for smart building maintenance. The subsequent chapters will dive into the details, from reviewing existing knowledge to designing and implementing the solution, and finally evaluating its effectiveness and security. These chapters unite to present an extensive study of digital technologies that will solve conventional building maintenance management difficulties leading to better sustainable building management.

Chapter 2

State of the Art and Literature Review

2.1 Introduction to the Chapter

This chapter reviews the state of the art in applying blockchain and related technologies in the AEC[1]/FM[2] domain, with a particular focus on integrated building maintenance management. First, some concepts and terminologies are explained to establish a common understanding of key technologies. Next, the literature is reviewed in thematic clusters reflecting major research trends: **(1)** *blockchain and IoT for maintenance*, **(2)** *smart contracts in construction automation*, **(3)** *BIM and blockchain integration*, **(4)** *blockchain for data and document management*, and **(5)** *blockchain for smart infrastructure and cities*. For every cluster, some studies are presented to point out the current trends, findings, and limitations. Finally, a critical analysis brings together these findings to show overlaps, limitations, and gaps in the existing literature. This chapter ends by specifying the research gap addressed by this thesis and the lack of a fully integrated decentralized application (DApp) that uses a 3D BIM model with blockchain-based smart contract logic for building operations and maintenance and outlines how the proposed work will contribute to filling this gap.

2.2 Foundational Definitions

To contextualize the literature review, this section provides definitions of the foundational concepts used for this research.

[1]Architecture, Engineering, Construction

[2]Facility Management

2.2.1 Blockchain

Blockchain is a tamper-resistant and decentralized digital ledger technology in which transactions are recorded in sequential blocks that are cryptographically linked, forming a chain across a distributed network of computers (nodes). There isn't a central authority controls this ledger, instead, consensus mechanisms (such as proof-of-work or proof-of-stake) allow the network of nodes to collectively verify and agree on new transactions, ensuring that once data is added it is extremely difficult to alter or remove. This design provides a high degree of **immutability** and **transparency**. All participants maintain a copy of the ledger, and any attempt to tamper with past records would be evident and rejected by the consensus of honest nodes. By establishing a single source of truth shared among parties who may not fully trust each other, blockchain enables secure, auditable record-keeping and value transfer without relying on intermediaries. The first implementation was in Bitcoin in 2008 [Nakamoto and Bitcoin [2008]], blockchain systems now they look for a wide range of applications beyond cryptocurrency, for example for supply chain management or digital identity, due to their ability to uphold data integrity and decentralized trust [Yaga et al. [2018]].

2.2.2 Ethereum

Ethereum is a decentralized blockchain platform introduced by Vitalik Buterin that extends the concept of blockchain with a built-in **Turing-complete programming language**, enabling the creation and execution of smart contracts and decentralized applications on its network [Buterin et al. [2013]]. Launched in 2015, Ethereum's architecture goes beyond simple transactions (as in Bitcoin) by incorporating an **Ethereum Virtual Machine (EVM)** on every node, which runs code expressed in smart contract languages (like Solidity) and allows users to implement arbitrary logic on the blockchain [Wood et al. [2014]]. Ether (ETH) is the platform's native cryptocurrency, used not only as a digital currency but also to pay **gas** fees for contract execution and network utilization, providing an economic incentive for miners (or validators, in Ethereum's current proof-of-stake system) to secure the network and process computations [Buterin et al. [2013]]. Ethereum's innovation lies in making the blockchain a general-purpose, programmable infrastructure: it guarantees that submitted code (smart contracts) will execute in a trustless manner (without central servers or authorities) and yield the same result on every node, which has led to a rich ecosystem of financial services, games, and other DApps running on the Ethereum network [Buterin et al. [2013]].

Ethereum Sharding

Ethereum sharding is a planned scalability mechanism that will partition the Ethereum blockchain's workload into multiple parallel chains (called **shards**) in order to dramatically increase throughput and capacity [Buterin]. Instead of requiring every node in the network to process every single transaction (the current design, which can become a bottleneck), sharding divides the network into groups of nodes each responsible for validating transactions for only a specific shard, allowing many transactions to be processed concurrently across the different shards [Bez et al. [2019]]. A central **beacon chain** (coordinating chain) will manage and secure the system by coordinating validators, maintaining a global consensus, and facilitating cross-shard communication. For example, ensuring that data and tokens can move from one shard to another in a consistent manner [Bez et al. [2019]]. In Ethereum's roadmap (previously referred to as "Ethereum 2.0"), sharding works hand-in-hand with the proof-of-stake consensus to preserve security: validators are randomly assigned to shards and to the beacon chain so that an attack on any single shard would require controlling a significant portion of the overall network. The sharding in Ethereum is expected to **increase transactions per second** that the network can handle as Ethereum would scale to serve the global user demand without losing its decentralization [Bez et al. [2019]].

2.2.3 Mainnet vs Testnet

In the Ethereum context, mainnet refers to the primary public Ethereum blockchain where actual transactions are broadcast, validated, and recorded with real economic value at stake, whereas a testnet (test network) is an alternate instance of the Ethereum network used purely for experimentation and development [Antonopoulos and Wood [2018]]. On the mainnet, Ether has real monetary value and smart contracts deployed there directly interact with live assets and users; by contrast, a testnet (such as Goerli or Sepolia) uses valueless tokens and mimics Ethereum's functionality in a sandbox environment so that developers can safely debug smart contracts or protocol upgrades without risking funds or impacting the main ledger [Antonopoulos and Wood [2018]]. Testnets closely resemble the mainnet in terms of software, consensus rules, and features, providing a realistic environment to identify issues and ensure performance under network conditions before deployment to mainnet. This separation is essential for the development pipeline of Ethereum (and also most of the blockchains), because It enables client developers and smart contract creators to guarantee reliability and security by solving problems on the testnet, and then transferring their code only to the Ethereum mainnet after they have tested their code thoroughly [Antonopoulos and Wood [2018]].

2.2.4 Polygon

Polygon is a blockchain platform and layer-2 scaling solution that operates alongside Ethereum to provide faster and cheaper transactions through the use of sidechains and other scaling techniques [Bjelic et al. [2022]]. Essentially, Polygon establishes a **parallel network** that is compatible with the Ethereum Virtual Machine, where users can run smart contracts and DApps with significantly lower fees and latency than on the Ethereum mainnet; periodic checkpoints (cryptographic proofs of Polygon's sidechain blocks) are then submitted to the Ethereum mainnet to **anchor** Polygon's security to Ethereum's robust consensus and to ensure state finality [Bjelic et al. [2022]]. Polygon's architecture is often described as an "Internet of Blockchains" for Ethereum: it supports multiple sidechains and rollup solutions, all connecting back to Ethereum, thereby expanding Ethereum's ecosystem into a multi-chain system while still leveraging the existing developer tools, wallets, and standards of Ethereum (Bjelic et al., 2021). Using a proof-of-stake consensus mechanism on its sidechain network, Polygon achieves block times on the order of a couple of seconds and can handle a far greater transaction throughput, which makes it well-suited for decentralized applications (like games, NFTs, or DeFi platforms) that require high frequency interactions without incurring prohibitive gas costs (Polygon, 2021). By adopting Polygon, developers and users benefit from the scalability and user experience improvements of a dedicated chain (or layer) for their application, while still retaining interoperability with Ethereum. For example, assets can be moved between Polygon and Ethereum, and smart contract code can be deployed on Polygon with minimal modifications since it's EVM-compatible.

2.2.5 Oracles

In blockchain systems, an *oracle* is a trusted off-chain data provider or middleware service that supplies a blockchain (and its smart contracts) with external information that the blockchain itself cannot access directly [Beniiche [2020]]. Blockchains are inherently isolated networks for security reasons (each node only has access to the data on the ledger and the transactions it processes), so when a smart contract needs real-world input(such as a price feed, weather condition, or the result of an event) it relies on an oracle to retrieve that data from an external source and deliver it onto the blockchain in a verifiable manner. Importantly, the oracle is not the data source itself but acts as a **bridge**: it queries and aggregates information from one or multiple sources, possibly adds a layer of verification or **authentication** (to ensure the data is trustworthy), and then feeds the formatted data into the smart contract's environment [Beniiche [2020]]. Oracles can be **software-based** (pulling data from online APIs), **hardware-based** (reading data from IoT sensors or devices in the physical world), inbound (bringing off-chain data in) or outbound (sending data or triggering action outside the chain), and can be centralized or decentralized networks of oracles. For example, a decentralized price oracle might take the median price of an asset from 10 different exchanges to update a DeFi contract. By using oracles, blockchains expand their usefulness allowing smart contracts to interact with real-world events events and with traditional systems, but they also introduce a trust consideration known as the "oracle problem," which is why modern designs often employ multiple oracles and cryptographic techniques to minimize reliance on any single data source.

2.2.6 Smart Contracts

Smart contracts are **self-executing** pieces of code that are stored on a blockchain and automatically enforce the terms of an agreement once predefined conditions are met ([Szabo [1997]]; [Yaga et al. [2019]]). The concept was first defined by Nick Szabo (1997) as "a computerized transaction protocol that executes the terms of a contract," highlighting the goal of reducing the need for trusted intermediaries in business agreements. In practical terms, a smart contract on a blockchain like Ethereum is a program consisting of functions (code logic) and state data that resides at a specific address on the blockchain, and which gets executed by the network's nodes as part of the blockchain's consensus process [Yaga et al. [2019]]. When a user triggers a smart contract (typically by sending a transaction calling one of its functions), every node in the network runs the contract's code and, if the conditions encoded in the contract are satisfied, the contract carries out the prescribed actions—such as transferring funds to a beneficiary, updating a record, or even creating a new contract—thereby **automatically** enforcing the agreement's outcome. Because the contract's code and execution results are recorded on the blockchain, they are tamper-evident and transparent: no party can unilaterally alter the contract or falsify its outcome without the consensus of the network, which means all parties can trust the process even in the absence of a central authority [Yaga et al. [2019]]. Smart contracts made it possible to create a vast tyoe of decentralized applications, starting from a simple token swaps to sophisticated financial products and organizational governance rules, by providing a reliable and programmable way to handle digital assets and logic on a blockchain ([Szabo [1997]]; [Yaga et al. [2019]]).

2.2.7 Decentralized Application

Decentralized applications (DApps) are software applications that run on a decentralized network, typically a blockchain or peer-to-peer network, rather than on a single centralized server (Logan, 2023). A DApp usually consists of back-end code (smart contracts) that execute on the blockchain and a front-end user interface that can be very similar to a conventional web or mobile app. Because the core logic resides on the blockchain, DApps inherit properties of blockchain systems: they are generally open-source, operate without a central authority's control, and data/transactions on the back-end are transparent and verifiable by the community (Johnston et al., 2014). For users, this means the application's behavior is predictable and not easily alterable by a single party—once a DApp is deployed, no company or individual can unilaterally shut it down or censor its function as long as the underlying blockchain continues to run. DApps exist for a wide array of purposes: for example, cryptocurrency wallets, decentralized finance platforms, games, social media, and marketplaces can all be implemented as DApps, leveraging blockchain-based trust (ensuring that rules are enforced by code) and **tokenization** (using digital tokens to incentivize users or represent assets) (Logan, 2023). While DApps can offer enhanced security and user autonomy (e.g., users keep custody of their data or assets), they also face challenges in scalability, user experience, and maintenance, since updates or bug fixes may require consensus from the network's community. Nonetheless, DApps represent a fundamental shift in application design by combining user-facing functionality with the decentralized, resilient back-end of blockchain networks (Johnston et al., 2014).

2.2.8 Building Information Modeling

Building Information Modeling (BIM) refers to both a technology and a collaborative process for managing building project data, whereby a **digital representation** of a facility's physical and functional characteristics is created and maintained as a shared knowledge resource for the building's life cycle (National Institute of Building Sciences, 2015). In practice, a *Building Information Model* is a rich 3D digital model of the structure that embeds not only geometric information (the building's design and spatial layout) but also a multitude of properties and attributes about building components – for instance, materials, structural loads, costs, manufacturer details, and maintenance schedules (Eastman et al., 2011). All stakeholders (architects, engineers, contractors, owners, facility managers, etc.) can contribute to and extract information from this single BIM model, which **streamlines communication** and reduces information loss as a project progresses from design to construction to operation (National Institute of Building Sciences, 2015). By enabling conflict detection (e.g., spotting a plumbing route clashing with a beam in the virtual model), simulation (such as energy performance or lighting analysis before the building is built), and efficient documentation (automatic generation of plans, sections, schedules from the model), BIM significantly improves decision-making and coordination compared to traditional isolated drawings and documents (Azhar, 2011). Ultimately, BIM is about leveraging a consistent, interoperable digital dataset of a facility to achieve better outcomes: reducing errors and rework, improving cost and time efficiency, and providing owners with a detailed digital twin of their asset for facilities management and future renovations (National Institute of Building Sciences, 2015).

2.2.9 Internet of Things

The *Internet of Things* (IoT) is a computing paradigm in which everyday physical objects and devices are embedded with sensors, software, and network connectivity so that they can collect data about their environment and communicate that data over the internet [Atzori et al. [2010]]. The underlying idea is a world where a myriad of "smart" objects from appliances, vehicles, and wearable devices to industrial equipment and infrastructure sensors are **interconnected**, continuously sharing information and working with the purpose to provide new services or automation. IoT devices typically have unique identifiers (like IP addresses) and use wireless networks to transmit data they sense (e.g., temperature, location, usage metrics) to cloud platforms or directly to other devices, often without human intervention. By connecting the physical and the digital worlds, IoT enables real-time monitoring, control, and analytics across diverse domains for example:

- **in a smart home:** IoT thermostats, lighting, and security systems can adjust settings autonomously for comfort and energy efficiency
- **in industrial settings:** IoT sensors on machinery support predictive maintenance by flagging performance anomalies
- **in healthcare:** wearable IoT devices can track patient vitals and alert doctors of issues remotely.

The IoT ecosystem as a whole encompasses not just the "things" themselves, but also the data networks, the cloud or edge computing systems that aggregate and analyze device data, and the applications that turn raw data into actionable insights. As IoT grows to billions of connected devices, important considerations include interoperability (common standards so different manufacturers' devices can work together), data management of the vast information generated, and security and privacy measures to protect sensitive data and prevent unauthorized control of devices [Atzori et al. [2010]].

2.2.10 Digital Twin

A *digital twin* is a virtual replica or model of a physical object, system, or process that runs in parallel to the real-world entity, continuously receiving data updates and reflecting its current state and behavior [Sean Olcott]. A digital twin is dynamically linked to its physical counterpart via sensors and data feeds, meaning changes or events in the real world (for example, a machine's temperature, a building's energy usage, or a patient's health metrics) are captured and sent to the digital twin in real time, keeping the digital model in sync with reality. With this live and connection with the real world, the digital twin can be used to **simulate and predict** outcomes without risking the actual asset, engineers and operators can run scenarios, test modifications, or forecast failures on the digital twin, gaining insights that inform decisions in the physical domain. Originally pioneered in manufacturing and aerospace (e.g., NASA using digital twins to simulate conditions for spacecraft and aircraft), the approach is now applied in many fields: in construction and facility management, for instance, a digital twin of a building (often derived from a BIM model augmented with IoT sensor data) enables monitoring of real-time performance and proactive maintenance, also in smart cities, digital twins of infrastructure help optimize traffic flow or utility distribution through simulation. By providing a holistic virtual environment that mirrors the life cycle of physical assets, digital twins help organizations achieve **predictive maintenance**, improved design and optimization, and risk reduction, as decisions can be tested virtually before implementation [Sean Olcott]. The fidelity and usefulness of a digital twin depend on the quality and timeliness of data it receives, and as such, advances in IoT, data analytics, and AI are continually enhancing digital twin capabilities for more accurate representation and intelligent insights.

2.2.11 Computerized Maintenance Management System

A *Computerized Maintenance Management System* (CMMS) is a software solution that supports organizations in planning, tracking, and optimizing maintenance activities by centralizing maintenance information in a digital database [IBM]. Core functions of a CMMS include an **asset registry** (recording details of equipment and facilities, such as make, model, location, and maintenance history), **work order management** (scheduling and assigning preventive maintenance tasks and repair jobs, and monitoring their completion), and **inventory management** for spare parts and supplies (ensuring that necessary parts are available for maintenance work and tracking their usage) [IBM]. By moving these processes from paper or disparate systems into one integrated platform, a CMMS enables maintenance teams to receive automatic reminders for routine inspections, log issues and resolutions, and analyze equipment performance over time. This leads to benefits like reduced downtime (through timely preventive maintenance and faster response to breakdowns), extended asset life (through better upkeep), and improved compliance with regulatory standards (by keeping auditable records of maintenance and safety checks). Modern CMMS software often provides a user-friendly interface (including mobile access for technicians in the field) and generates reports or dashboards – for example, showing metrics such as mean time between failures (MTBF) or maintenance backlogs – that help managers identify trends and make informed decisions about resource allocation and asset replacement [IBM]. In the end, a CMMS is the information backbone of maintenance management, which improves the efficiency and consistency of maintenance activities in industries such as manufacturing plants, hospitals, campus facilities, and transportation fleets [IBM].

2.3 Literature Review

With these key concepts defined, the following sections review the literature by thematic clusters. Each cluster corresponds to a specific intersection of technologies and application areas in AEC/FM, highlighting how scholars have been combining blockchain, BIM, IoT, and smart contracts to innovate in building maintenance and related fields.

As shown in Figure 2.1, the reviewed literature was grouped into five clustersas previously said: **(i)** *Blockchain + IoT for Real-Time Maintenance*, (ii) *Smart Contracts for Supply Chain and Automation*, **(iii)** *Smart Building and Digital Twin Integration*, **(iv)** *Blockchain for Document and Data Management*, and **(v)** *Blockchain for Smart City Infrastructure*. These clusters reflect recurring research directions and guided the structure of the literature review in the sections that follow.

2.3.1 Cluster 1: Blockchain and IoT for Maintenance

One prominent stream of research explores the integration of blockchain with the Internet of Things (IoT) to enable real-time monitoring and maintenance of buildings and infrastructure. The advent of IoT in construction around 2014 and blockchain in 2017 opened the door to combining these technologies, and studies have found their integration promising for addressing challenges in **real-time data collection**, **remote monitoring**, and **automated maintenance workflows**. In essence, IoT sensors provide timely and rich data from the physical environment, while blockchain offers a secure and decentralized way to transmit, store, and trigger actions based on that data. Research in this cluster has proposed various frameworks to leverage this synergy for maintenance management.

Real-time Maintenance Tracking with Sharded Blockchains: [Wang et al. [2023]] present a framework for building operations and maintenance that uses *blockchain sharding* to improve scalability in processing IoT-driven maintenance transactions. The proposed approach, **supported by IoT devices**, allows maintenance events (e.g., sensor-detected faults) to be recorded on different shards, reducing latency and improving the throughput of the system. The authors demonstrate that as the complexity and size of the maintenance management system grows, sharding can significantly benefit performance. In a maintenance context, this means even a large volume of sensor alerts and work orders can be handled on-chain without bottlenecking the entire network. The use of sharded blockchain, therefore, addresses one of the key concerns of applying blockchain in IoT heavy scenarios – the limited transaction processing rate, which is critical if every sensor reading or maintenance action is logged as a blockchain transaction.

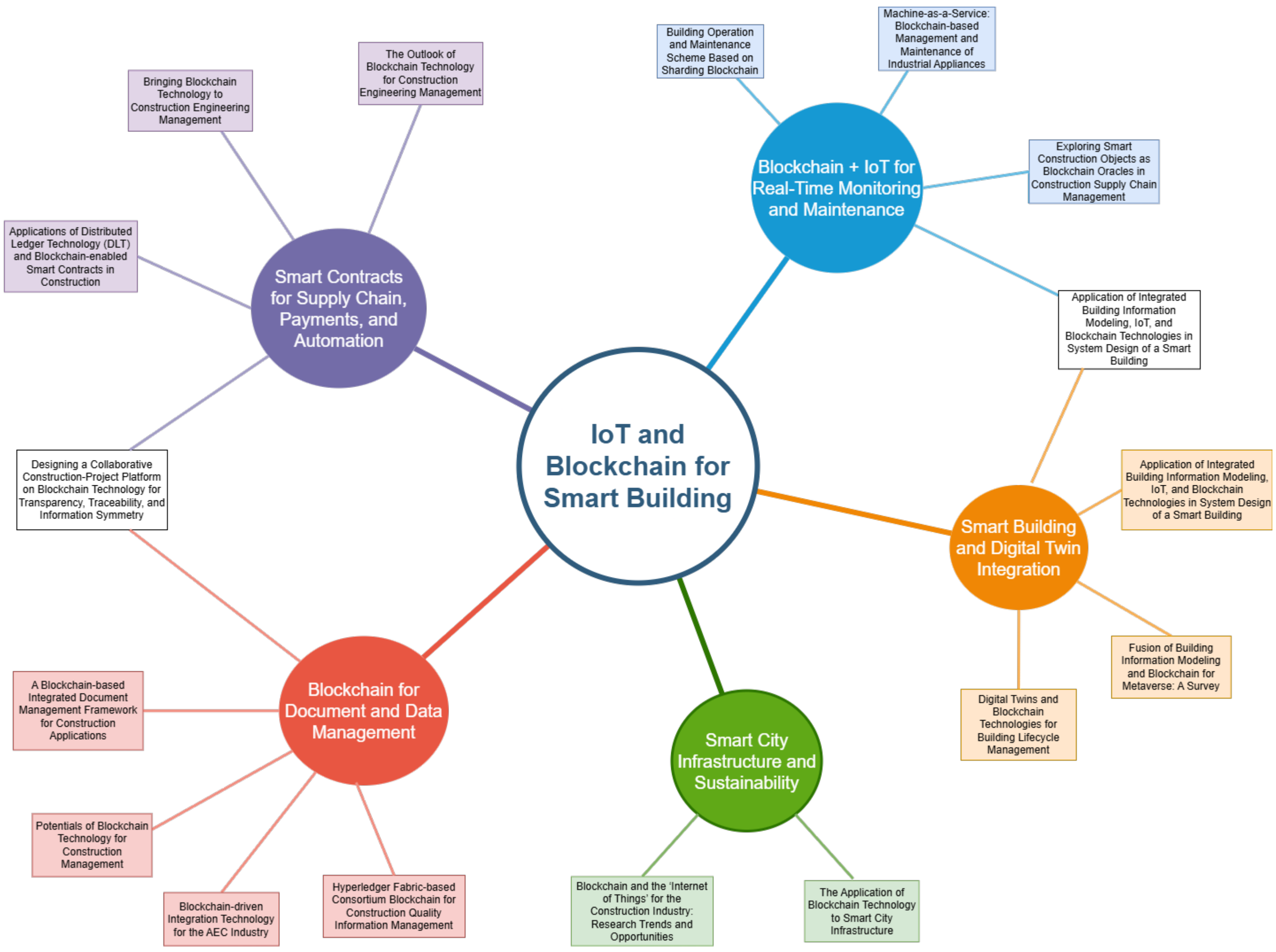

IoT and Blockchain for Smart Building
Smart Contracts for Supply Chain, Payments, and Automation
Bringing Blockchain Technology to Construction Engineering Management
The Outlook of Blockchain Technology for Construction Engineering Management
Applications of Distributed Ledger Technology (DLT) and Blockchain-enabled Smart Contracts in Construction
Designing a Collaborative Construction-Project Platform on Blockchain Technology for Transparency, Traceability, and Information Symmetry
Blockchain + IoT for Real-Time Monitoring and Maintenance
Building Operation and Maintenance Scheme Based on Sharding Blockchain
Machine-as-a-Service: Blockchain-based Management and Maintenance of Industrial Appliances
Exploring Smart Construction Objects as Blockchain Oracles in Construction Supply Chain Management
Application of Integrated Building Information Modeling, IoT, and Blockchain Technologies in System Design of a Smart Building
Smart Building and Digital Twin Integration
Application of Integrated Building Information Modeling, IoT, and Blockchain Technologies in System Design of a Smart Building
Fusion of Building Information Modeling and Blockchain for Metaverse: A Survey
Digital Twins and Blockchain Technologies for Building Lifecycle Management
Smart City Infrastructure and Sustainability
Blockchain and the 'Internet of Things' for the Construction Industry: Research Trends and Opportunities
The Application of Blockchain Technology to Smart City Infrastructure
Blockchain for Document and Data Management
A Blockchain-based Integrated Document Management Framework for Construction Applications
Potentials of Blockchain Technology for Construction Management
Blockchain-driven Integration Technology for the AEC Industry
Hyperledger Fabric-based Consortium Blockchain for Construction Quality Information Management

Figure 2.1. Thematic clustering of reviewed literature in the domain of blockchain, IoT, and BIM for smart building maintenance

Smart Construction Objects (SCOs) as Blockchain Oracles: A notable challenge in merging IoT with blockchain is how to feed real-world data into smart contracts in a trustworthy manner. [Li and Kassem [2021]] tackle this by introducing *Smart Construction Objects (SCOs)* as **blockchain oracles** in a construction supply chain setting. In their framework, SCOs are physical components (devices) of construction (or attached to assets) that are equipped with sensors and unique identifiers (e.g. QR codes), which are able to sense the condition as well as to digitally sign or confirm data. These SCOs relay valid data to the blockchain, which, in turn, become trusted data feeds for smart contracts. For example, an SCO attached to a building component could detect an anomaly in the case of this thesis (e.g. a temperature spike indicating an overheating light fixture) and automatically trigger a maintenance smart contract once that data is recorded on the blockchain. instead, they demonstrated this concept in a logistics and materials tracking scenario, but it is equally applicable to facility maintenance. The implication is that *authenticated IoT data* (via SCOs) can autonomously initiate maintenance processes on-chain – a crucial capability for any decentralized maintenance management system. By ensuring the data's reliability, SCO-based oracles increase stakeholders' trust that automated maintenance triggers (such as work order creation or emergency shutdowns) are based on genuine, tamper-proof sensor readings.

Machine-as-a-Service (MaaS) for Industrial Maintenance: Another illustration of blockchain–IoT integration is provided by [Tran et al. [2023]], who developed a *Machine-as-a-Service (MaaS)* platform for managing industrial machinery maintenance using blockchain, IoT, and decentralized storage (IPFS). In their scenario, each machine in a factory has IoT sensors that detect operational status and failures. When a sensor flags an issue, it sends a notification to a blockchain network dedicated to maintenance. The blockchain logs this event as a maintenance request (a transaction on the ledger) which is visible to all authorized maintenance service providers (who operate as nodes on the network). Smart contracts coordinate the response: for instance, the first available maintenance provider to acknowledge the request on-chain is assigned the task (a "first-come, first-served" assignment logic encoded in the contract). The maintenance provider then services the machine and logs the completion on the blockchain, which could automatically trigger payment via cryptocurrency. The inclusion of IPFS (InterPlanetary File System) in the architecture allows large data (e.g. machine logs, repair documents, sensor datasets) to be stored in a decentralized off-chain manner while the blockchain stores references and assures integrity. This case study is compelling because it demonstrates a working prototype that automates the maintenance workflow end-to-end: *fault detection → blockchain event → service dispatch → confirmation → payment*, all mediated by IoT and smart contracts.

In conclusion, the research in Cluster 1 highlights the viability and advantages of integrating the IoT sensing with the blockchain networks for the maintenance management. The key contributions include frameworks for improving blockchain's performance in IoT settings (through sharding), methods to trustworthily connect IoT data to smart contracts (using SCO oracles), and prototypes of autonomous maintenance dispatch systems (MaaS). Collectively, these works show that blockchain can provide a reliable, tamper-proof log of real-time asset conditions and maintenance actions, while IoT provides the data stream to populate that log and trigger automated responses. The increased transparency and timeliness can enable *predictive and preventive maintenance* strategies that were difficult to coordinate in traditional systems. However, most studies in this cluster focus on specific technical elements (scalability, oracle design, etc.) in isolation; a fully integrated solution that marries real-time BIM models with IoT inputs and blockchain logging is still in an nascent stage, which we will revisit in the critical analysis.

2.3.2 Cluster 2: Smart Contracts in Construction Automation

The second cluster of literature focus on the use of blockchain smart contracts to automate workflows in construction and facilities management, including supply chain transactions, payment processes, and multi-party task coordination. The construction industry is notorious for its fragmented supply chains and heavy reliance on intermediaries for contract management and payments. Smart contracts provide a potential solution by embedding in code business logic (agreements, rules, conditions) that self-executes on a blockchain, eliminating the need for central oversight and opportunities for dispute. Research in this cluster demonstrates how smart contracts can streamline processes like progress payments, procurement, and contractor selection in construction projects.

Automating Payments with Smart Contracts: A representative work by [Li and Kassem [2021]] explores applications of blockchain and smart contracts in construction, including a prototype system for *automated milestone payments*. In traditional construction projects, payments to contractors or suppliers are often tied to milestones or deliverables and require verification (inspections, certificates) and manual approval workflows. [Li and Kassem [2021]] implemented a smart contract that holds project funds in escrow and automatically releases payment when a milestone completion is confirmed on the blockchain. For example, once an inspector logs on the blockchain that "Phase X is complete" (perhaps by digitally signing a transaction), the smart contract triggers a payment to the contractor's address. This ensures payments occur *only after* the required work is completed. The blockchain record provides a transparent and immutable trail of who certified completion and when, reducing delays in approval and opportunities for disputes over whether a milestone was met. The authors reported that such a system can significantly reduce the administrative overhead and latency associated with processing invoices, as well as build trust that payments will not be released prematurely.

Collaborative Project Platforms and Workflow Automation: Another important study is by [Udokwu et al. [2021]], who designed a **collaborative construction project platform** on blockchain to improve information symmetry and trust among all project participants. Their platform uses smart contracts to automate workflow steps and share data securely in real-time between stakeholders such as owners, contractors, subcontractors, and inspectors. The goal is to ensure that *all parties see a single source of truth* for project information (contracts, change orders, schedules, etc.) and that key events (approvals, completions, deliveries) are recorded immutably. [Udokwu et al. [2021] specifically address **transparency, traceability, and information symmetry**, noting that when all data (documents, approvals, transactions) are stored on a blockchain ledger accessible to participants, it greatly increases trust and reduces coordination frictions. In their prototype, for instance, a smart contract could automatically route a work order to the appropriate subcontractor when approved by the general contractor, and log this event for everyone to see. They found that such platforms can *automate workflow* (reducing manual communication and data reconciliation) and enhance inter-party trust since no single entity can manipulate the records.

Automating Contractor Selection and Procurement: Researchers have also explored how smart contracts could autonomously handle procurement decisions and contractor selections. [Boonpheng et al. [2020]] discuss theoretical approaches for smart contracts to run *auction-like processes* to select suppliers or maintenance contractors based on predefined criteria such as lowest cost or highest reputation rating. In a conventional scenario, choosing a vendor for a job (like replacing a set of light fixtures) might involve soliciting bids and negotiating, often through a centralized system or manual process. In a blockchain scenario, one could encode a "tender" smart contract: when a maintenance need arises, the contract accepts bids from registered contractors (each bid perhaps submitted as a transaction) and automatically awards the job to the best bidder according to the criteria coded (for example, the lowest bid from a contractor with at least a 4-star rating). This paper highlight that such a system would not only speed up the procurement process but also leave a clear audit trail of how the decision was made (all bids are on-chain and time-stamped). They note this approach could reduce bias and enable more open competition.

Benefits and Challenges Identified: Across the cluster 2 literature, the demonstrated benefits of smart contracts in construction automation include increased efficiency (process steps that used to take days or weeks can be executed in seconds on-chain), improved transparency and accountability (every transaction and decision is recorded for stakeholders to review, reducing disputes), and reduced need for intermediaries (which can lower costs and complexity). Likewise, automated workflows can impose compliance to agreed procedures (no step is skipped since smart contract won't proceed without inputs). These works, however, recognize challenges and limitation. One issue is *integration with existing processes* and legacy systems and construction firms might have established project management software or practices that are not easily replaced by blockchain systems. Another challenge is the need for clear standards and legal frameworks: while a smart contract can execute a payment, the legal enforceability of such an automated release, or handling of exceptions (e.g., what if a milestone's quality is disputed?), require careful consideration outside the code. Scalability and user friendliness of these solutions are also noted concerns – early prototypes work in small trials, but industry-wide adoption would demand robust performance and intuitive interfaces (so that non-technical stakeholders can comfortably use the DApp). Lastly, *standardization* across platforms is mentioned as an open issue: if different projects use different blockchain networks or standards, it hinders interoperability and broad adoption. These challenges represent important gaps that subsequent research (including this thesis) aims to address by designing architectures that can bridge new and old systems and by evaluating the practical aspects of deploying such solutions in real maintenance scenarios.

2.3.3 Cluster 3: BIM and Blockchain Integration

The third cluster of literature focuses on integrating blockchain with Building Information Modeling (BIM) and *digital twin* concepts, specifically targeting improvements in data management over the building lifecycle, enhanced collaboration, and traceability of changes in smart building environments. As BIM provides a centralized digital model of a building, linking it with blockchain can create a trusted record of all modifications and transactions related to the building, from construction through operations and maintenance. This cluster examines how such integration can enable smarter buildings through secure data sharing and real-time synchronization between the physical and digital realms.

Enhancing Smart Buildings with Blockchain and Digital Twins: [Adu-Amankwa et al. [2023]] investigate the synergy between blockchain and digital twins for building lifecycle management. They argue that while digital twin technology can give a real-time reflection of a building's state (through BIM models updated with live IoT sensor data), blockchain can serve as a reliable backbone to log all changes to the digital twin and ensure the integrity of the data. Every time the building's digital twin is updated, either with an activity (maintenance or sensor reading), or change (renovation), or any other update, this information can be captured as a blockchain transaction, creating an immutable audit trail during the building's lifetime. This approach yields multiple benefits: it enables *predictive maintenance* by combining real-time monitoring (IoT + digital twin) with the assurance that all sensor data and predictive analytics outputs are trustworthy (thanks to blockchain). It also guarantees that any modification in the building (for example, replacing a component or updating a system setting) is transparently documented. As a result, stakeholders can always verify the history of an asset or system via the tamper-proof ledger. [Adu-Amankwa et al. [2023]] show that such an integrated system could lead to more accurate maintenance (since decisions are based on comprehensive, reliable data) and better accountability. If something goes wrong, one can trace back through blockchain records to see when and how a related change was made.

BIM–Blockchain Fusion in the Metaverse Context: [Huang et al. [2022]] take a forward-looking and more theoretical perspective by exploring the fusion of BIM and blockchain in the context of the *metaverse*, which they frame as a convergence of virtual and physical realities. In their survey, they discuss how integrating BIM models with blockchain could enable rich **virtual simulations** and collaborative digital environments for building design and management. For example, a BIM model secured by blockchain could be placed in a shared virtual space (metaverse) where multiple stakeholders (architects, engineers, facility managers) interact with it simultaneously, confident that any changes made in that environment are verifiable and traceable. Although much of their work is conceptual, they highlight use cases relevant to maintenance: virtual reality simulations for training or planning maintenance tasks, using blockchain tokens to represent building assets or maintenance tickets in a metaverse platform, and ensuring that whatever occurs in the virtual model (e.g., a simulated sensor trigger or a maintenance procedure tested virtually) can be synchronized and validated against the real building's data. An insight from [Huang et al. [2022]] work is that blockchain can facilitate *cross-platform interoperability*. If a BIM model exists in a metaverse or any digital twin platform, a blockchain can act as the neutral ground where every system records approved changes or exchanges data. This is particularly useful when envisioning future smart cities where many digital systems need to coordinate. While this survey is theoretical, it provided numerous ideas and underscored that BIM–blockchain integration is gaining interest as a means to manage increasingly complex building data environments.

Integration Frameworks and Key Findings: Overall, studies in Cluster 3 demonstrate that combining BIM/digital twins with blockchain and IoT yields *smart building systems* with enhanced reliability and traceability. Key findings include: **(1) Enhanced Change Management:** Blockchain's immutable logs can record all changes to a BIM model or digital twin, making a reliable version history for the facility. This is important in maintenance management, and knowing what was done, who did it, and when (and in knowing that it isn't altered later) is key to accountability and quality control. **(2) Cross-Stakeholder Collaboration:** A blockchain-integrated BIM allows different stakeholders (owners, maintenance contractors, inspectors, etc.) to confidently collaborate on the same model, since any data they contribute is secured and origin-stamped. It reduces fear of data tampering and enables a more decentralized contribution model for updating facility information. **(3) Potential for Predictive Maintenance:** As noted, the fusion of real-time sensor data (IoT) with BIM (as a digital twin) and blockchain (as a secure database) sets the stage for predictive maintenance – the system can predict faults and automatically log and perhaps even address them with smart contracts, all while the BIM model reflects the latest status of the building. The literature provides evidence that such setups can work in pilot scenarios, although fully operational industry examples are still limited.

It is worth noting that many works in this cluster, while optimistic, are in early stages (conceptual frameworks, surveys, or small-scale prototypes). They clearly demonstrate *the potential* of BIM–blockchain integration, but also hint at the complexity of bringing these together. For example, ensuring real-time synchronization between BIM databases and blockchain transactions is non-trivial (BIM tools are not naturally built to interface with blockchain networks). Additionally, storing detailed BIM data on-chain is impractical due to size, so careful system architecture (on-chain vs off-chain division) is required. These considerations inform the architecture proposed in this thesis and will be discussed in the next chapter. Nonetheless, the state-of-art suggests that bridging BIM and blockchain is a promising path to smarter, more maintainable buildings, by combining the strengths of each technology: BIM's rich information model and blockchain's trust and automation capabilities.

2.3.4 Cluster 4: Blockchain for Data and Document Management

The fourth cluster focuses on how blockchain can be used in document and data management in construction and facility management, security of sharing, version control, and traceability. Construction projects and facility operations involve a good amount of documentation, like contracts, drawings, specifications, maintenance logs, compliance certificates, etc. Historically, these things are stored in separate systems or even on paper, and this causes problems in consistency and access. Blockchain's core features (immutability, distributed access, cryptographic security) make it a possible solution for solving critical data management tasks where trust and coherence are paramount. This cluster's literature focuses on frameworks that leverage blockchain as a backbone for document management and information tracking.

Blockchain-Based Document Management Frameworks: [Das et al. [2022b]] propose an integrated document management framework for construction that uses blockchain and smart contracts to handle the flow of project documents. In their system, each document (for example, a blueprint, RFI, or maintenance report) is registered on the blockchain with a unique hash, and transactions are used to record actions on documents (creation, approval, revision, etc.). A smart contract ensures that only authorized parties can submit revisions and that the latest version of each document is easily verifiable. This effectively creates a secure, tamper-proof version control system. For instance, if a maintenance manual is updated or a new inspection report is issued, the blockchain log would show exactly which version is current and who approved it. The authors illustrate how this could prevent common issues like lost paperwork or conflicting document versions. In a maintenance scenario, such a framework could track work orders and service records: each maintenance request and its outcome report could be a document whose lifecycle (issued, assigned, completed, verified) is managed by blockchain, guaranteeing that no entry can be altered or deleted retroactively. The transparency and integrity provided can increase confidence in the data, for example a facility manager can trust that a "completed work order" record is legitimate and final, since it's sealed on the blockchain.

Immutable Logs for Project and Asset Data: [Turk and Klinc [2017]] early on recognized blockchain's potential for **information traceability** in construction management. They talked about how an immutable ledger could track project events and changes, and that the changes in plans and schedule are duly documented. In their perspective, even if their focus was broad, one can see the application to maintenance: imagine an immutable maintenance log for an asset (say an elevator) – every service action, part replacement, or inspection is appended to that asset's log on the blockchain. Years of records could never be accidentally lost or intentionally doctored; stakeholders (owners, insurers, inspectors) could retrieve the full maintenance history with confidence in its fidelity. [Turk and Klinc [2017]] highlighted that such transparency can reduce informational disputes and improve regulatory compliance (auditors could instantly verify whether required maintenance was performed). They also pointed out the usefulness in monitoring modifications: if a change is made to a design or a maintenance procedure, the blockchain could notify relevant parties and serve as evidence of who authorized the change. While their work was conceptual, later studies (like [Das et al. [2022b]]) have built on those ideas to implement actual frameworks.

Secure Data Integration for IoT and BIM: [Zhong et al. [2023]] present a study on "blockchain-driven integration technology for the AEC industry," focusing on data from IoT sensors and BIM models. They position blockchain as a *secure data layer* that sits between IoT/BIM systems and end-user applications. In their approach, data produced by IoT devices (environmental readings, occupancy, equipment status) and data from BIM-based systems (asset info, spatial layouts) are time-stamped and stored or referenced in a blockchain, which acts as a unified, trustworthy data exchange platform. This ensures that whether data is coming from a sensor or a BIM database, when it's retrieved through the blockchain layer, it carries a proof of authenticity and hasn't been tampered with. The efficiency of the system is maintained by not storing the raw data on-chain (due to volume) but by storing secure hashes or indices and using off-chain storage for bulk data. For example, a temperature sensor's readings might be stored in a cloud database, but each significant reading or threshold-crossing event would be hashed and recorded in a blockchain transaction along with a reference to the data. Similarly, a BIM change (like updating an asset's property) might be logged on-chain. [Zhong et al. [2023]] found that this approach creates an **ecosystem of data** that is more reliable and readily shareable among systems. In practical terms, for building maintenance, this means one could build dashboards or analytics that draw from both BIM and live sensor data via blockchain queries.

Common Themes: The papers in Cluster 4 address some of the most important pain points regarding data fragmentation and security. They show how blockchain is able to *standardize handling of the data* from different sources and stakeholders through its provision of a single source of truth. Important themes include:

- **Data Fragmentation:** Traditionally, maintenance data might be siloed (some in a CMMS, some in spreadsheets, some in email threads). Blockchain-based systems encourage integration, as data from different subsystems (sensors, BIM, maintenance logs) can all register on the same ledger. This helps break down silos and ensure everyone is looking at consistent information.

- **Data Security and Integrity:** By leveraging cryptographic hashes and decentralized consensus, these systems protect data from unauthorized changes. This is especially important for compliance documents or safety records in maintenance and any attempt to fudge a record (like skipping a safety check) would be evident if the blockchain entry is missing or conflicting.

- **Auditability:** Maintenance and operation processes can be reviewed after the fact by examining blockchain logs. For example, an investigation into a critical equipment failure can trace all related maintenance actions on that equipment via its blockchain history, revealing if any required maintenance was missed. This auditability is a form of accountability that can drive better performance from maintenance teams.

- **Interoperability:** A subtle but powerful aspect is that blockchain doesn't replace existing systems, but rather links them. The document management framework can tie into existing document repositories, the IoT integration can tie into existing sensor networks, etc., but through blockchain they become more interoperable (everyone uses the blockchain as the reference for data exchanges). This means upgrades can be incremental..

In summary, cluster 4 shows that using blockchain for data/document management in AEC/FM moves the industry toward more **transparent, secure, and collaborative information management**. These approaches align with the needs of integrated maintenance management: maintenance is fundamentally a data management challenge (keeping track of what needs to be done, what has been done, and by whom, with what outcome), and blockchain-based solutions are proving to be effective tools to manage such information at scale and with high integrity.

2.3.5 Cluster 5: Blockchain for Smart Infrastructure and Cities

The fifth cluster broadens the scope to smart infrastructure and smart cities, examining how blockchain (often combined with IoT) can support urban-scale maintenance and resource management. While not focused solely on building maintenance, these works provide insights into large-scale, distributed maintenance problems (like city infrastructure upkeep) and sustainability initiatives (like energy grids), which parallel challenges in building maintenance and operations. The cluster underscores how blockchain can facilitate secure data sharing and coordination in complex urban ecosystems, and these ideas can trickle down to inform building-scale systems.

Smart City Maintenance and Resource Management: [Alnahari and Ariaratnam [2022]] explore blockchain applications in smart city infrastructure, particularly for managing utilities like energy and water. They propose ideas such as blockchain-enabled peer-to-peer energy exchanges between city buildings and using blockchain to track the maintenance of public infrastructure assets (streetlights, pipelines, etc.). A key point is that smart cities involve myriad connected devices and stakeholders (city authorities, private citizens, utility companies), so a high level of transparency and trust is needed in data and transactions. Blockchain, coupled with IoT, is posited as a solution to ensure data from distributed sensor networks is collected and used in a trustworthy way for decision-making. For example, a sensor on a public water pipe detects a leak; a blockchain-based maintenance system could automatically create a record of that event and perhaps issue a repair token or contract to a maintenance crew, visible to both the water department and city auditors. [Alnahari and Ariaratnam [2022]] highlight that while their approach was more focused on resource transactions (like energy trading).

Improving Traceability in Construction Supply Chains: [Elghaish et al. [2021]] present a study on blockchain and IoT for the construction industry that, among other things, addresses resource tracking and automation in construction supply chains. While not explicitly about "maintenance", their findings on traceability and automated management of construction components have clear parallels. They describe how blockchain and IoT can together ensure that every component (like a pre-fabricated wall panel or a critical piece of equipment) is tracked from manufacturing to installation, with all handoffs recorded on-chain. This has implications for maintenance because knowing the provenance and history of building components is valuable when servicing them, e.g., if a certain batch of materials was defective, a blockchain record could quickly identify all buildings containing those materials. Additionally, [Elghaish et al. [2021]] discuss automation of processes using smart contracts, such as auto-ordering a replacement when a stock level goes low or triggering maintenance workflows when IoT sensors report anomalies. They note that even though their work was focused on active construction management, the same principles apply to operational phase: the combination of blockchain and IoT can provide *end-to-end visibility* of assets and orchestrate actions without manual oversight.

Relevance and Adaptation to Building Maintenance: The studies in cluster 5 are somewhat tangential to building maintenance in that they often consider either a broader scale (city infrastructure) or adjacent processes (supply chain, resource sharing). However, the reason to include them is that they introduce innovative ideas and architectures that can be adapted to maintenance DApps. They showcase the versatility of blockchain beyond single buildings – proving its value in *distributed, large-network environments*. For example, if a smart city platform can manage hundreds of thousands of IoT devices via blockchain, then a building with a few hundred sensors is quite feasible. They also focus on **sustainability and efficiency**, which goes in line with the idea that a maintenance system should not only repair, but also optimize (reduce energy consumption by maintaining equipment better). The key takeaways applied to our context are: **a)** blockchain can handle coordinating maintenance across multiple sites or organizations (important if facility management is outsourced or if one service provider handles many buildings), and **b)** the transparency blockchain offers is beneficial at any scale (from a building to a city) for accountability and performance tracking.

To summarize cluster 5: Blockchain and IoT in smart infrastructure contexts have been used to increase transparency, data trust, and efficiency in managing public assets and resources. These works, while not directly solving building maintenance, provide proof that the technology stack we consider (blockchain + IoT + smart contracts, possibly plus BIM/digital twin) can work in even the most complex, large-scale settings. They also inspire some features (like peer-to-peer service models or automated cross-organization workflows) that could be innovative if applied to building maintenance. For instance, one could envision a future where buildings in a city share a blockchain network for maintenance, and a maintenance contractor can service any building, logging their work on the same ledger, leading to a city-wide maintenance marketplace, analogously to the P2P energy trading concept.

2.4 Critical Analysis of Literature

To synthesize the literature reviewed from Cluster 1 to Custer 5, Table 2.1 categorizes each analyzed paper according to key technological dimensions: Blockchain (BC), Smart Contracts (SC), IoT, Smart Buildings (SB), Maintenance Tasks (MT), Construction Operations (COT), Traceability/Fidelity (T/F), and whether the paper presented a Prototype or Idea. This matrix allows for cross-comparison of contributions and reveals which technologies have been most frequently combined in the existing literature.

Reference	**Year**	**Type Pub**	**BC**	**SC**	**IoT**	**SB**	**MT**	**COT**	**T/F**	**P/I**
[Lu et al. [2021]	2021	Journal	✓	✓	✓	✓	✓	✓	✓	Prototype
[Das et al. [2022b]	2022	Journal	✓	✓	×	×	✓	✓	×	Prototype
[Adu-Amankwa et al. [2023]	2023	Journal	✓	✓	✓	✓	✓	✓	✓	Prototype
[Wang et al. [2023]	2023	Journal	✓	✓	✓	✓	✓	✓	×	Idea
[Li and Kassem [2021]	2021	Journal	✓	✓	✓	✓	✓	✓	✓	Idea
[Tran et al. [2023]	2023	Journal	✓	✓	×	×	✓	✓	✓	Prototype
[Lokshina et al. [2019]	2019	Conference	✓	✓	✓	✓	✓	✓	×	Prototype
[Elghaish et al. [2021]	2021	Journal	✓	✓	✓	✓	✓	×	✓	idea
[Mahmudnia et al. [2022]	2022	Journal	✓	✓	×	×	✓	✓	✓	Idea
[Zhong et al. [2023]	2020	Journal	✓	✓	✓	×	✓	✓	×	Prototype
[Turk and Klinc [2017]	2017	Conference	✓	✓	×	✓	✓	✓	×	Idea
[Kim et al. [2020]	2020	Journal	✓	✓	✓	×	✓	✓	×	Idea
[Wang et al. [2017]	2017	Journal	✓	✓	✓	✓	✓	×	✓	Idea
[Boonpheng et al. [2020]	2020	Journal	✓	✓	×	×	✓	✓	×	Idea
[Alnahari and Ariaratnam [2022]	2022	Journal	✓	✓	✓	✓	✓	✓	✓	Idea
[Das et al. [2022a]	2023	Journal	✓	✓	×	×	✓	✓	×	Idea
[Huang et al. [2022]	2022	Journal	✓	✓	✓	✓	✓	✓	×	Idea
[Udokwu et al. [2021]	2021	Conference	✓	✓	✓	✓	✓	×	✓	idea

Table 2.1. Summary of reviewed papers

Having reviewed five thematic clusters, we now synthesize the insights and identify trends, overlaps, and gaps in the state of the art. **A clear pattern emerges:** blockchain technology is being explored as a foundational infrastructure to enhance trust, transparency, and automation in AEC/FM processes, often in conjunction with IoT and BIM. Most studies converge on the notion that blockchain's immutable and decentralized ledger can solve many information management issues prevalent in construction and maintenance

– from disjointed data and lost documents to payment disputes and lack of accountability. At the same time, each cluster highlights different facets of the problem space, and together they paint a picture of progress as well as areas needing further development.

Across clusters, there is a **strong overlap** in the recognition of blockchain's core benefits: **immutability, transparency, and decentralization** are repeatedly cited as game-changers for collaboration in AEC/FM. Whether it is real-time maintenance logs (cluster 1), automated payment and workflow records (cluster 2), BIM change logs (cluster 3), document version histories (cluster 4), or city-wide sensor data (cluster 5), the literature agrees that having a tamper-proof, shared ledger accessible to all relevant parties increases trust and efficiency. Many works also overlap in combining technologies: it is rare to find a blockchain application in AEC/FM that does not involve IoT or BIM or both. For example, cluster 1 and cluster 5 both heavily involve IoT sensors feeding blockchain systems; cluster 3 and cluster 4 both involve BIM or structured data models interfacing with blockchain. There is consensus that blockchain is not a standalone silver bullet but part of a tech ecosystem including smart devices (for data input) and data models/analytics (for making sense of the information). Another overlapping point is the emphasis on **automation via smart contracts**. Clusters 1, 2, and 5, in particular, all use smart contracts to replace or streamline traditional human-mediated processes – be it assigning a maintenance job, executing a payment, or initiating a data-sharing process. This shows a common belief that many routine tasks in maintenance management can be encoded in smart contracts to reduce delays and errors. Overlaps are also seen in the anticipated outcomes: improved response times, reduced operational costs (by cutting out middlemen or redundant steps), better compliance (since everything is logged), and improved stakeholder satisfaction due to transparency.

Despite these overlaps in vision, the works also have distinct focuses and, in some cases, limitations when viewed collectively. One noticeable divergence is the **level of implementation and realism**: Some studies present fully developed prototypes and case studies (e.g., the MaaS system in cluster 1, or the document management prototype in cluster 4), whereas others remain theoretical or exploratory (e.g., the metaverse fusion in cluster 3, or broad surveys in cluster 5). This indicates that certain subdomains (like automated payments or IoT maintenance triggers) are closer to practical realization, while integrated BIM-blockchain systems are still largely conceptual.

In terms of technical scope, many works simplify one dimension to focus on another – for instance, a IoT+blockchain maintenance prototype might not incorporate BIM at all, working with a simple database model of assets instead. On the other hand, a BIM+blockchain framework might assume sensor data input but not deeply address how that data is ingested (perhaps glossing over IoT integration complexities). As a result, when surveying the whole field, we see *fragmentation*: different pieces of the overall puzzle (sensing, data modeling, blockchain logic, user interface) are addressed in isolation by different researchers. This fragmentation is partly due to the nascency of the field – tackling all aspects at once (IoT + BIM + blockchain + actual maintenance process) is challenging, so researchers have understandably broken the problem down. However, it means that **a gap exists in uniting these threads into a coherent whole.**

Several works identify issues like lack of standardization, regulatory uncertainty, and integration with legacy systems as barriers. For instance, if each project uses a different blockchain platform, there is no interoperability; if there are no agreed data standards, every integration becomes a custom job. Very few papers propose solutions to these issues – they are acknowledged but left for future work. Security is another aspect: while blockchain itself is secure, the applications built on it (smart contracts, oracles, etc.) could have vulnerabilities. Only a minority of works (perhaps some in cluster 1 regarding secure oracles) touch on cybersecurity of the overall system. Moreover, performance concerns like blockchain transaction costs (gas fees) and latency are only occasionally discussed (one mention in cluster 1 noted sharding to tackle performance). In practical maintenance management, if reporting a fault or updating a model incurs a fee or delay, users might be reluctant. So far, the literature has not fully resolved how to make these systems cost-effective and fast enough for real-world use, especially public blockchain networks. Many prototypes use either private/permissioned blockchains or simply do not address the cost model.

User adoption and human factors are not deeply studied in the technical literature. We can deduce that if technicians and managers are to use a blockchain-based DApp, it must be user-friendly and clearly superior to their current tools (CMMS, etc.). While the technical benefits are documented, we don't see studies on user acceptance or training. This is a shortfall that suggests future research (and indeed our thesis) should consider the end-user perspective: building a system that not only functions technically but also aligns with how people can and want to work.

Identified Gaps in the Literature: Based on the analysis, the following gaps have been identified:

- **Lack of Fully Integrated Prototypes:** As mentioned, no existing work appears to deliver a *complete DApp* that integrates a 3D BIM model interface with blockchain smart contracts to manage maintenance tasks in a real-world scenario. We see BIM–blockchain integration in theory, IoT–blockchain prototypes in isolation, and smart contract automation for discrete processes, but a holistic system combining all these (a true end-to-end maintenance management DApp) is missing. In other words, there's a gap between concept and reality, few if any studies *demonstrate a maintainer clicking on a BIM model, creating a blockchain-backed work order, and then updating the BIM model based on blockchain records all in one pipeline.* This is precisely the kind of demonstration needed to validate the viability of the integrated approach.
- **User-Centric Evaluation:** The current state of the art lacks evaluation of these systems in practice, for example pilot deployments in a facility management department or feedback from facility managers. Most works validate on technical criteria (does it work? is it faster? more secure?) but not on organizational criteria (does it fit existing workflows? improve user satisfaction? require regulatory changes?).

This gap means it's not yet proven how blockchain maintenance systems perform in the messy reality of daily operations.

- **Interoperability and Standards:** Another gap is the need for standards. While not a single study can fill this, collectively the field would benefit from common data models or protocols (perhaps extending IFC for BIM with blockchain transaction hooks, or a standard for IoT oracles in construction). Right now, each prototype might use its own assumptions. The absence of standards is noted as a barrier, and future work (including this thesis as a stepping stone) should ideally contribute to or at least be compatible with emerging standards in digital building logbooks, etc.

- **Scalability and Performance in Maintenance Context:** We also identify a gap in evidence about how these systems scale in a busy maintenance environment. For example, a skyscraper could have thousands of sensors and maintenance tickets per year, can a blockchain handle that volume smoothly? Techniques like sharding (cluster 1) or permissioned chains might be needed, but more data is required. Our thesis can help fill this by measuring our prototype's performance and discussing how to mitigate costs (perhaps by using layer-2 solutions or permissioned networks for internal data, etc.).

In summary, the literature has firmly established **the promise of blockchain in integrated maintenance management** but has yet to deliver a fully realized, widely tested solution. The works overlap in showing what could be done and why it's beneficial, but they fall short of showing it all working together in practice. There is a noticeable gap in taking these converging technologies from the conceptual and prototype stage to a production-ready, user-accepted system.

Chapter 3

Proposed Architecture

This chapter presents the **high-level architecture** of the decentralized building maintenance DApp. It provides a conceptual overview of how the system's components interact, without going into low-level implementation or code. The goal is to describe **system design choices** and how they meet the requirements of a building maintenance platform on blockchain. Key elements of the architecture are introduced, including a React front end with an IFC model viewer, on-chain smart contracts, and an off-chain server. The roles of different actors (building users, maintenance technicians, and administrators) are outlined to show how the DApp manages a maintenance workflow in a decentralized yet efficient manner. By the end of this chapter, the reader should understand the overall system structure and the rationale behind **design decisions** such as integrating both on-chain and off-chain components and using dual identifiers (physicalId and positionId) for assets.

3.1 Motivation and Context

One of the main motivations for implementing the maintenance Dapp is the *absence of an integrated solution* that merges blockchain and BIM in a single maintenance management framework. Blockchain is still a new technology in the costruction building field, it started gaining attention from the industry since 2017 and most of the pubication are still conceptual [[Elghaish et al. [2021]]. This leaves a substantial room for new applications and deeper investigations.

Many current maintenance systems **rely** on central servers or *third-party services*, this can lead to single points of failure problems, potential data tampering, and a lack of transparency for stakeholders. Instead, blockchain technology delivers decentralized storage alongside **transaction immutability**: each transaction or record is confirmed and stored across a distributed network, reducing the possibility of unauthorized modifications. This immutable nature proves crucial for building maintenance operations because it establishes verifiable records of faults and repair histories that act as essential evidence in audits and disputes.

Integrating **smart contracts** *into* a **BIM** context offers a **complete new approach** to automating tasks like fault reporting, maintenance scheduling, and payment workflows. Smart contracts removes many of the manual checks required in centralized systems and this ensures the process is both trustless and verifiable. Given the early stage of this combined technology stack in the literature, the present work aims to address a clear gap, demonstrating how a decentralized application (DApp) can simplify building maintenance while offering higher **data security**, **transparency**, and **process automation** than existing centralized platforms.

The **objective** of this system is to achieve an integration of a blockchain ledger with a 3D BIM model so that building maintenance data is both trustworthy and easy to use. User experience side, a React based interface is implemented with a 3D IFC model viewer which ensures that non-technical users have a easy way to visualize assets, submit fault reports, and track progress. Inside this interface, a role based access management is enhanced with smart contracts that imposes role-based access control, ensuring only approved users and technicians can execute certain actions. All critical events, such as a broken asset or successful maintenance (technician side), are logged in the blockchain. Meanwhile, an off-chain server handles non-critical data such as gas reimbursement records to reduce on-chain costs. This architecture is designed to achieve several key requirements:

- **Transparent Maintenance Logs**: All major events (like broken assets, completed repairs) are recorded on-chain to prevent tampering.
- **BIM Integration**: A 3D model interface allows maintenance staff to pinpoint and manage physical assets visually.
- **Role Management**: Smart contracts enforce separate permissions for unregistered users, regular users, technicians, and administrators.
- **Cost Efficiency**: Off-chain storage minimizes transaction fees, and a reimbursement mechanism compensates users who pay gas to log faults.
- **Scalable/Extensible**: The architecture can expand to include more advanced features, such as automated scheduling or AI-driven analytics, without changing its fundamental components.

In the subsequent Implementation chapter, we will examine the actual React components that realize these ideas, along with the specific Solidity contracts, database schema, and the data flows that make the system operational.

3.2 System Architecture Overview

In order to achieve transparent, tamper-proof maintenance logging while retaining a user-friendly experience, this system is composed of **three primary layers** that together form a hybrid decentralized application (**DApp**). These layers are: *front-end*, *on-chain contracts*, and *off-chain components*. They collectively meet the goals set out in the previous section (e.g., transparency, cost efficiency, role-based access). Figure 3.1 illustrates how these layers interconnect at a conceptual level. From a technical standpoint, the React app is responsible for:

3.2.1 Front-End Client (React + IFC Viewer)

At the forefront is the **user interface**, which runs entirely in the **browser**. Built with React, this client enables building occupants, technicians, and administrators to interact with the system through a clear, intuitive interface. One of the key features is a fully integrated 3D IFC (Industry Foundation Classes) model viewer, which allows a user to visually inspect assets in their correct spatial context. For example, someone reporting a fault can literally click on a lamp or an HVAC unit within the 3D environment, and the application will link that click to the corresponding asset data. From a technical standpoint, the React app is responsible for:

- **Visualizing the Building**: It fetches a BIM model (in IFC format) and renders it using a library based on Three.js. This model provides immediate visual feedback to the user, showing them exactly which floor or room they are dealing with.
- **Handling User Actions**: Whether someone reports a broken fixture, a technician initiates a maintenance operation, or an admin approves payment, the React app packages these user requests and sends them to the appropriate smart contract in order to register them inside the blockchain for critical state changes.
- **Wallet Connectivity & Metamask Integration**: Since this is a decentralized application, user authentication and transaction signing happen through Metamask. This ensures every maintenance event or role-related action is cryptographically signed and traceable.

In this layer, the **role-based views** also come into play. An *unregistered user* might have minimal access with only viewing building assets or requesting registration, while a technician and admin will see richer dashboards: *open tasks, pending maintenance approvals, or administrative panels.* All these role constraints are enforced at the smart-contract level, too, but the front end implements them in the UI to keep the user experience straightforward and reduce potential errors.

3.2.2 Blockchain Smart Contracts (On-Chain Layer)

The second layer is the “On-Chain”, it represents the **core decentralized logic** of the system. It is written in Solidity, these contracts store and manage crucial data structures, such as registered building assets, reported faults, maintenance events, and user roles. For instance, each asset might be represented in a contract with fields like an **assetId**, a **positionId** (location in the building), a **physicalId** (individual device reference), and a **status** (“Operational”, “Broken”, “Under Maintenance”). When a user reports a fault (e.g., “Broken lamp in Room 101”), the transaction updates that asset’s status, logs a brief description of the fault, and records the reporter’s wallet address for later reference or reimbursement. Additional contract functions allow:

- **Role Management**: An administrator can approve or deny a user’s request to become a “Technician,” while the system ensures only recognized technicians can initiate or complete maintenance.
- **Payment Flows**: Once a job is completed on-chain, the admin can trigger a payment from a payment manager contract loaded with funds. The same mechanism may reimburse users for the gas fees they spent on important transactions (like reporting a fault), ensuring the solution is fair and cost effective for building occupants.
- **Event Emissions**: Whenever a fault is reported, maintenance is started, or a job is finalized, the contract emits structured events. The events are detected real-time by the frontend client, which triggers an immediate UI updates for all relevant parties.

3.2.3 Off-Chain Backend (Express Server + SQLite Database)

Although a large portion of the system’s logic resides on-chain, certain tasks are impractical or cost-inefficient to implement entirely via blockchain. To address those needs, this architecture includes a minimal off-chain component, accessible via a standard REST API. Types of data handled off-chain mainly includes:

- **Gas Fee Tracking**: When a user reports a fault, they pay a small transaction fee (gas). The front-end can log that cost with the off-chain server, so an admin can later confirm the exact amount and reimburse it—without forcing extra complexity on the blockchain.
- **High-Volume or Large Data Storage**: In this database off-chain are stored all the raw reading from the simulated IoT sensor, because storing them directly on-chain would be too expensive.
- **Optional Integrations**: If in the future the system needs to send automated notifications (SMS/email) or integrate with existing corporate software, the off-chain server can facilitate these connections. It can forward relevant on-chain events to external services or store partial data for bridging legacy systems to the DApp.

Thanks to this design, the heavy, high frequency queries or big data storage **does not occur on the blockchain**. Instead, it remains on a familiar server-based environment, reducing the burden on end users. Meanwhile, the critical ledger of maintenance events remains decentralized, giving the system the transparency and security that purely centralized frameworks often lack.

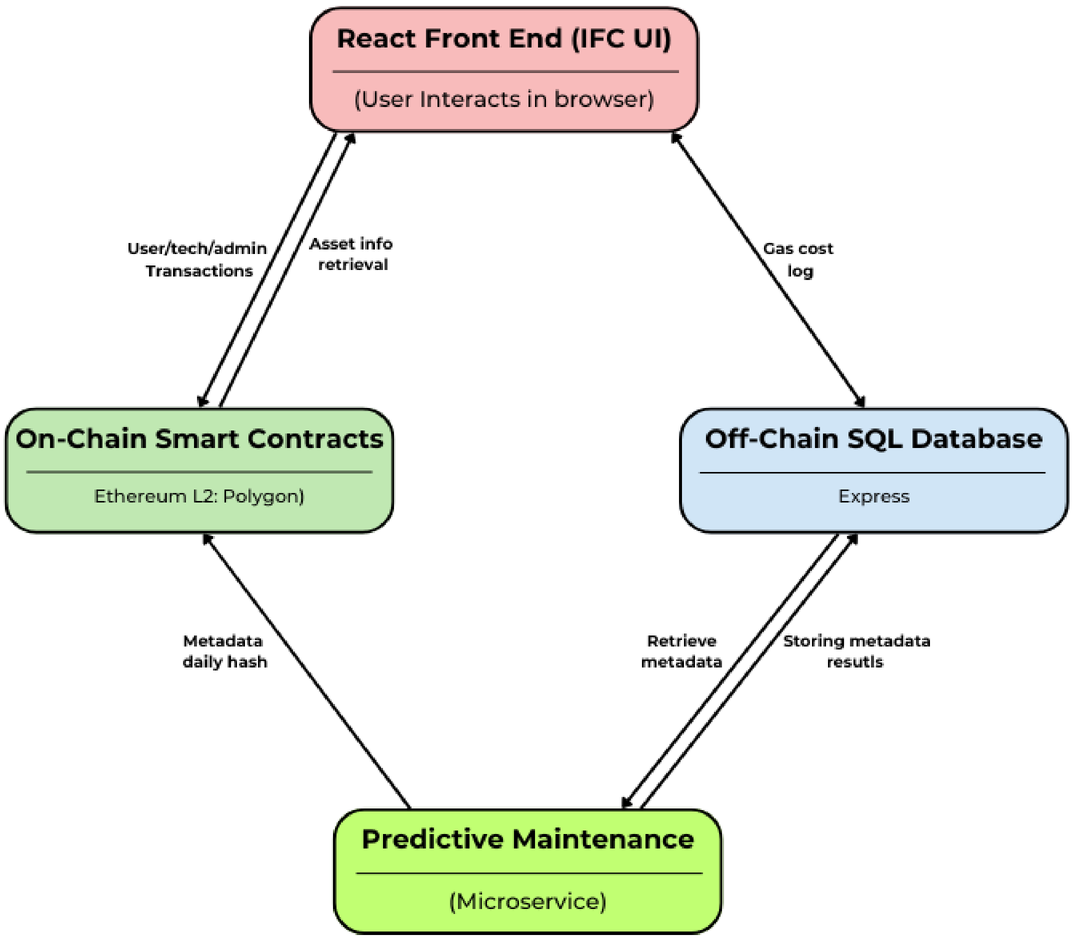


Figure 3.1. System Architecture Diagram

Figure 3.1 would visually represent the three layers (front end, on-chain layer, off-chain server) and the predictive system, along with arrows demonstrating how data flows among them. For instance, the figure might show a building occupant or a technician on the left interacting with the **React + IFC front end**, which then, in one arrow path, sends transactions to the smart contracts on Polygon for updating or retrieving asset states, and, in another arrow path, makes API calls to the Express + SQLite backend to log gas costs or retrieve large data sets. In a textual label, you could highlight how each user role sees a slightly different interface, but they all go through the same front-end module to reach the underlying logic.

Placing this diagram and short description helps the reader grasp the overall solution at a glance. They will see that:

- **Smart contract logic** ensures trust and enforces role-based policies.
- **React + IFC serves** as the user-facing layer, bridging building data and blockchain calls.
- **Express/SQLite** complements on-chain data for cost, performance, and convenience.

This structure lays the foundation for subsequent sections that detail either the workflow (**how user → technician → admin interactions proceed in a real maintenance scenario**) or deeper architectural considerations (like how assets are tagged with both physicalId and positionId). By presenting the system in layered form, you clarify how each piece cooperates to create a robust, scalable building maintenance DApp.

3.3 Components of the Proposed System

This section examines the architecture's main components and describes each in detail. Each subsection covers the role and responsibilities of one part of the system, highlighting how it contributes to the overall DApp functionality. The focus is on *what* each component does in the architecture (conceptually), and *how* it interfaces with other components, rather than how to implement it in code.

3.3.1 Front-End Application (React and IFC Model Viewer)

A significant element of this system's architecture is the front-end client, a React-based single-page application (SPA) that mediates between building occupants, technicians, administrators, the blockchain layer, and an off-chain server. Its design emphasizes user-friendliness, role-based controls, and deep integration with 3D building information models.

User Interface and IFC Integration: One of the distinguishing features of this front-end is the inclusion of a **3D IFC model viewer**, which renders a digital representation of the building directly within the browser. The *Industry Foundation Classes (IFC)* format is an open standard for Building Information Modeling (BIM), representing physical and functional building data in a structured manner. By embedding an IFC viewer, the front-end enables occupants or maintenance staff to visualize the real-world layout of a facility: floors, rooms, structural elements, and equipment.

When a user identifies a defective resource, for example, a flickering light or a malfunctioning HVAC unit—they **can click** on the corresponding element in the 3D model, rather than guessing an asset ID or specifying it by textual description alone. This linking of physical location to digital data offers a more intuitive approach, reducing errors in fault reporting. Rather than writing *"Lamp #47 in Hallway 2"* the user can simply select the lamp in the viewer, automatically capturing its unique identifiers for the maintenance request. This integration thus increases context-awareness for maintenance tasks and ensures each fault is correctly associated with the right building component.

Role-Based Interface Views: The **React** design of the front end is heavily influenced by the various user roles in the system: *unregistered visitors, regular building owners, technicians, and administrators*, each of whom needs different views and functionalities. When a user connects their **crypto wallet** (e.g. via MetaMask), the app queries the blockchain to ascertain that account's role. Based on the response:

1. **Unregistered Users** may only see limited options, such as the ability to request registration.
2. **Building Occupants** (Regular Users) can file fault reports, check the status of open maintenance tickets, or see a simplified asset overview.
3. **Technicians** gain additional panels to view and accept unassigned jobs, record maintenance progress, and eventually mark tasks as completed.
4. **Administrators** can access administrative dashboards, such as pending user or technician approvals, asset registration forms, and the ability to authorize payments or reimbursements.

Web3 Wallet Connectivity: To maintain decentralizaion, the front end rely on user's blockchain wallet (e.g., MetaMask) which serves as their identity. The React application integrates with a web3 provider library (such as Ethers.js) to handle:

- **Transaction Signing**: Whenever a user wants to report a fault, accept a job, or confirm a maintenance completion, the front end prompts them to sign and broadcast a blockchain transaction from their wallet.
- **Read Calls to Smart Contracts**: The app invokes read-only contract functions to fetch data about assets, role statuses, or maintenance records. This ensures the UI is always displaying up-to-date information directly from the on-chain source.

Because any significant system state change triggers a blockchain transaction, the user's **private key** (in their wallet) is required for each high-level action—improving security. The **front end never sees or handles the private key**, it only requests the wallet's signature, thus preserving the trustless nature of the application.

Communication with Backend: While the smart contracts store and manage the core maintenance logic (asset states, fault logs, role checks), certain support data is more economically or functionally handled off-chain. For instance, when a user reports a fault, the transaction itself costs some amount of gas and this cost might be reimbursable under the organization's policy. To facilitate that, the front-end makes an HTTP request to an off-chain server, recording the user's address and the exact gas fee they spent. Later, the admin can verify and compensate them via a separate on-chain payment. Similar back-end interactions might arise for:

- Uploading or retrieving large files (e.g., extended documentation or image attachments)
- **Caching** IFC models or subsets for performance
- **Integrating** with external enterprise software (like emailing notifications or updating a corporate asset-management system)

Importantly, the blockchain remains the authoritative source of truth for everything essential to maintenance records and system logic, while the off-chain server only augments that data. Hence, losing back-end data would not compromise the system's integrity (it would only affect convenience features).

3.3.2 Smart Contract Layer (On-Chain Logic in Solidity)

A core part of this **decentralized maintenance system** is that *mission-critical data and workflows are handled by self-executing smart contracts* that are on a blockchain. This approach guarantees trust, transparency, and immutability in storing building assets, fault reports, maintenance assignments, and payment disbursements. Rather than relying on a proprietary or central server, the on-chain layer acts as the application's "backend logic," forming a secure foundation for the entire DApp.

On-Chain Responsibilities: The smart contracts, written in Solidity, fulfill several pivotal duties:

1. **Asset Registry and State Tracking**: Each building asset—whether it be a light fixture, HVAC component, or elevator panel which is registered on-chain and linked to relevant metadata, such as its status ("Operational," "Broken," "Under Maintenance"), any fault descriptions, and timestamps of changes. When a user or technician interacts with an asset (e.g., *reporting a fault or starting maintenance*), the contract updates the asset record accordingly and stores a tamper-proof audit trail of events.

2. **Maintenance Request Management**: To initiate any maintenance process, a user's blockchain transaction is made, which changes the asset's state to **"Broken"**. Once under maintenance, further functions move the asset to **"Under Maintenance"** or **"Operational"**, culminating in a closed loop that thoroughly records every maintenance step.

3. **Enforcing Roles and Permissions**: Different participants have distinct privileges (User, Technician, Admin). The contract ensures that only authorized technicians

can mark maintenance tasks complete, and only an admin can approve new users, update roles, or register brand-new assets. This **on-chain role-based** access ensures that malicious or unqualified actors cannot claim or finalize maintenance tasks, nor can they unilaterally alter a record. If the front end's role-based UI is circumvented, the contract logic still blocks any disallowed functions from unauthorized addresses, underscoring the system's integrity.

4. **Immutability and Event Logging**: All status changes, fault reports, or role updates are stored in the contract's state or emitted as events. Because these actions require blockchain transactions, they are irreversibly recorded. This property provides robust **auditability**, letting administrators or external auditors prove exactly when a fault was reported, who accepted it, and when it was finished—all without trusting a single centralized authority.

Role Management and Security: to keep the system **secure** in an open blockchain environment, the contracts implement a clear role structure:

- **Unregistered**: A default role for addresses not yet recognized by the contract. Typically, unregistered users can only request registration.
- **User**: A building occupant, allowed to create fault reports and track existing maintenance records.
- **Technician**: A verified maintenance specialist. Only addresses labeled *"Technician"* are able to perform key maintenance operations on-chain, such as starting or completing repairs.
- **Admin**: A privileged address or set of addresses capable of approving registrations, removing or adding technicians, registering new assets in the system, and issuing critical commands (e.g., final payment approvals).

The *contract* implements address-based authorization to verify that users attempt authorized operations within their assigned domains. For instance, if a random user tries to complete the maintenance, the contract would reject that call unless the caller is recognized as a technician. This mechanism offers the security of a zero trust environment: no matter how an attacker tries to manipulate front-end code or API calls, the final on-chain logic stands firm, disallowing any role-violating actions.

Maintenance Workflow Logic: The on-chain system provides all the key steps needed to reflect real-world maintenance processes. In practice, these steps often align with the workflow described in Section 3.4 but from the contract's perspective:

1. **Report Fault**: A user transaction *reports the fault* within the asset with a message for the technician/admin. The contract updates the asset's status to "Broken," logs the reporter's address, sets a time/date stamp, and *emits an event* (e.g., fault has been reported).

2. **Start Maintenance**: A technician that comes in the building **start the maintenance** in the Web Application, switching the asset to "Under Maintenance," setting the assigned technician's address, and possibly storing initial commentary or a start timestamp.

3. **Complete Maintenance**: When the technician is done, they complete the maintenance for that asset with the transaction. The contract sets the status to "Operational" and logs the completion details.

4. **Verification/Payment**: I dedided not to make this step automatic, so the admin approves the maintenance done by the technician and finalyze the process by triggering the payment contract. This final step is optional in some designs if immediate payment is not required, but it's typical in an escrow scenario.

Because these transitions all require **blockchain transactions**, they leave a permanent record of the building's maintenance timeline. Anyone can query the chain (via, e.g., Ethers.js or blockchain explorers) to see the complete history for a particular asset, reinforcing system transparency.

Payment and Incentives via Smart Contracts: One of the system's distinctive features is that **technicians' compensation** can be managed by a dedicated payment contract or an integrated module within the main maintenance contract. The chosen approach depends on how one wishes to organize logic, but the essential idea is:

- **Escrowed Funds**: An admin or building manager can deposit a certain quantity of **POL** (*Polygon's native token*) into the payment contract.
- **Conditional Release**: Once the technician completes a job and the admin validates it (on-chain), the contract automatically transfers the agreed-upon fee to the technician's address. Optionally, the same contract can reimburse the fault reporter's gas fee.
- **Transparency**: All transfers are public, allowing building owners to see exactly how much is spent on maintenance each time and allowing technicians to track their verified payouts confidently.
-

This blockchain-based payment flow **removes** the need for an external accounting or invoice system to settle bills. Provided the contract is properly tested and has enough balance, technicians know their compensation is guaranteed without manual overhead or potential disputes.

Network Deployment (Sepolia Testnet and Polygon Testnet): In practice, these contracts were first deployed on **Ethereum Sepolia testnet** to allow safe trial runs without incurring real world fees or risking production data. After verifying correctness, *the final code was migrated* to **Polygon Mainet**, known for:

- **Lower Transaction Fees**: On Polygon, typical gas fees are only a fraction of the cost on Ethereum mainnet, enabling frequent updates (e.g. multiple daily fault reports or asset updates) without prohibitive expenses.
- **Faster Block Times**: Confirmations are generally quicker, improving user experience (no long waits to confirm a fault report).
- **EVM Compatibility**: Code written for Ethereum works on Polygon with minimal changes, significantly easing deployment and maintenance.
- Security via Ethereum Anchoring: Polygon periodically commits checkpoints to the Ethereum chain, leveraging mainnet security for its sidechain consensus.

This deployment choice underscores a real world architectural trade off: a strongly decentralized, trustworthy solution must remain cost-effective for practical usage. Polygon's environment meets both functional and economic needs, making it well-suited for an application that might see many small value transactions.

3.3.3 Off-Chain Backend (Express.js Server and SQLite Database)

While the **primary logic and state are on-chain**, the architecture includes a supporting off-chain backend component. This subsection describes why an off-chain server is used and what role it plays in the overall system. Key points to cover:

Rationale for Off-Chain Components: Clarify that not all aspects of an application are efficient to handle on a blockchain. Some data might be too large, change too frequently, or be auxiliary (not worth the cost of on-chain storage). The off-chain backend in our architecture addresses these needs by handling **complementary tasks** that augment the on-chain functionality. It's important to note that this does not re-centralize the core of the application, instead, it is a purposeful design to **achieve a hybrid architecture** where the blockchain is used where it adds value (trust, integrity) and conventional server/database is used for everything else (performance, cost savings).

Express Server and REST API: The backend is built with Express.js (a Node.js framework) and provides a RESTful API that the front-end can call. Describe how the front end might send HTTP requests to this server for certain operations. For example, one implemented feature is tracking the **gas costs** that users spend when they report faults. When a user submits a maintenance request (which costs a small amount of cryptocurrency as transaction fee), the front end can record this cost by calling an API endpoint on the Express server. The server will store that information in its SQLite database. This way, the system can later retrieve it (e.g., when an admin is ready to reimburse the user's gas fee via the smart contract, they can query the backend for how much the user spent). By using a simple REST API, the integration between the front-end and this backend is straightforward, and it allows capturing useful data **without putting it on-chain** (since recording every gas fee on-chain would ironically cost more gas).

SQLite Database for Lightweight Data: Explain that the backend uses a lightweight file-based database (SQLite) to persist the data it manages. The data stored is relatively small and specific. Emphasize that this database is **not a source of truth for maintenance status**, it only holds supplemental information. All critical state (like whether an asset is broken or fixed, or who the technician is) lives on the blockchain. The SQLite just helps with things like tracking usage metrics, gas maintenance track and the predictive maintenance log data.

Security and Trust Considerations: Since we introduce an off-chain element, address how we maintain trust. Note that the data kept off-chain is either non-critical or is cross-validated with on-chain data. For instance, if the backend suggests reimbursing a certain gas cost to a user, the admin will still cross-check that a legitimate fault report transaction exists on-chain for that user/asset before actually approving reimbursement. The architecture deliberately keeps the **authoritative data on-chain** to avoid any single point of failure or tampering on the server. Additionally, because the backend is not involved in core decision-making (it cannot, for example, change an asset's status, only the contract can do that), the worst-case impact of a backend failure is limited to some convenience features, not the integrity of the maintenance process.

3.4 Workflow of Maintenance Operations

After describing the static components of the system, this section focuses on the **dynamic behavior**: how the different actors interact with the system through a sequence of steps to accomplish building maintenance tasks. We provide a conceptual **flow of actions** from the moment a maintenance issue is identified to the resolution and payment. A flowchart (Figure 3.2) will accompany this explanation, illustrating the interactions among a **User**, a **Technician**, and an **Admin** through the DApp. Also, a BIM model of the building used in this DApp is shown in Figure 3.3. The flowchart should depict the key stages (reporting a fault, performing maintenance, and payment authorization) and how information and control passes between the actors and the system components (front end, smart contract, etc.). Below is a narrative (and step-by-step breakdown) of the maintenance process: Figure 3.2 Maintenance Interaction Flow. This diagram will show the sequence of actions and decisions in the system. For instance, it can be drawn as a diagram with four lanes for User, Technician. Admin and Smart Contract. Arrows will indicate actions like "User submits fault report" going to the smart contract, and then "Technician receives task" and so on. Ensure the figure captures the logic that only valid role players can perform certain actions (e.g., only technician can complete the task, only admin can approve payment). In the text, each major step in the flow is described.

User Identifies a Fault & Reports It: The process begins when a building occupant (in the role of **User**) encounters a maintenance issue (for example, a broken light or malfunctioning equipment). Using the DApp front-end, the user selects the affected asset. They do this by clicking on the asset's location in the IFC model viewer (which provides the asset's unique identifiers to the system). The user then fills in a fault report

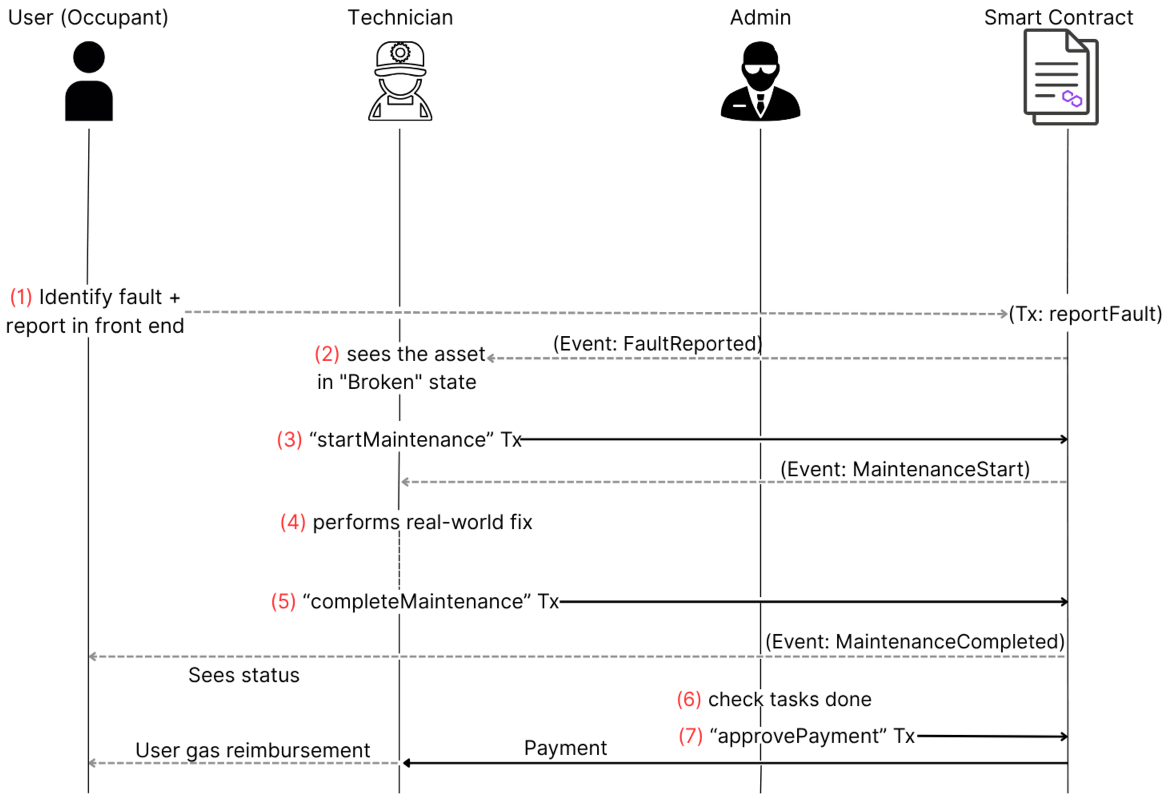

User (Occupant)
Technician
Admin
Smart Contract
(1) Identify fault + report in front end
(Tx: reportFault)
(2) sees the asset in "Broken" state
(Event: FaultReported)
(3) “startMaintenance” Tx
(Event: MaintenanceStart)
(4) performs real-world fix
(5) “completeMaintenance” Tx
(Event: MaintenanceCompleted)
Sees status
(6) check tasks done
(7) “approvePayment” Tx
User gas reimbursement
Payment

Figure 3.2. Workflow Diagram

form describing the issue (fault type, notes) and submits it. When the user submits, the front end triggers a **smart contract transaction** (calling the function to report a fault for that asset). The smart contract logs this fault: it updates the asset’s status to “Broken” (or an equivalent status indicating it needs maintenance) and records the fault details along with the reporter’s address. An event (FaultReported) is emitted on-chain, which can be picked up by the interface to confirm the action. The user’s interface now shows that the fault has been reported successfully. *(If applicable, mention that at this point the front end also calls the backend to record the gas cost of this transaction for potential reimbursement, as a behind-the-scenes step.)* From the user’s perspective, their job is done until the issue is fixed; they can later check on status updates through the app. All the relative graphic implementation is shown in Figure 3.4.

Task Creation & Technician Awareness: Once reported, the maintenance issue becomes visible to those in the **Technician** role. Describe how the system makes technicians aware of new faults. Each technician using the DApp can see a list of open maintenance jobs awaiting attention. The architecture allows a technician to **self-assign** a task. In our implementation, any authorized technician can choose to handle the task, the first to respond will effectively get it, as they will initiate maintenance in the next step. This is a

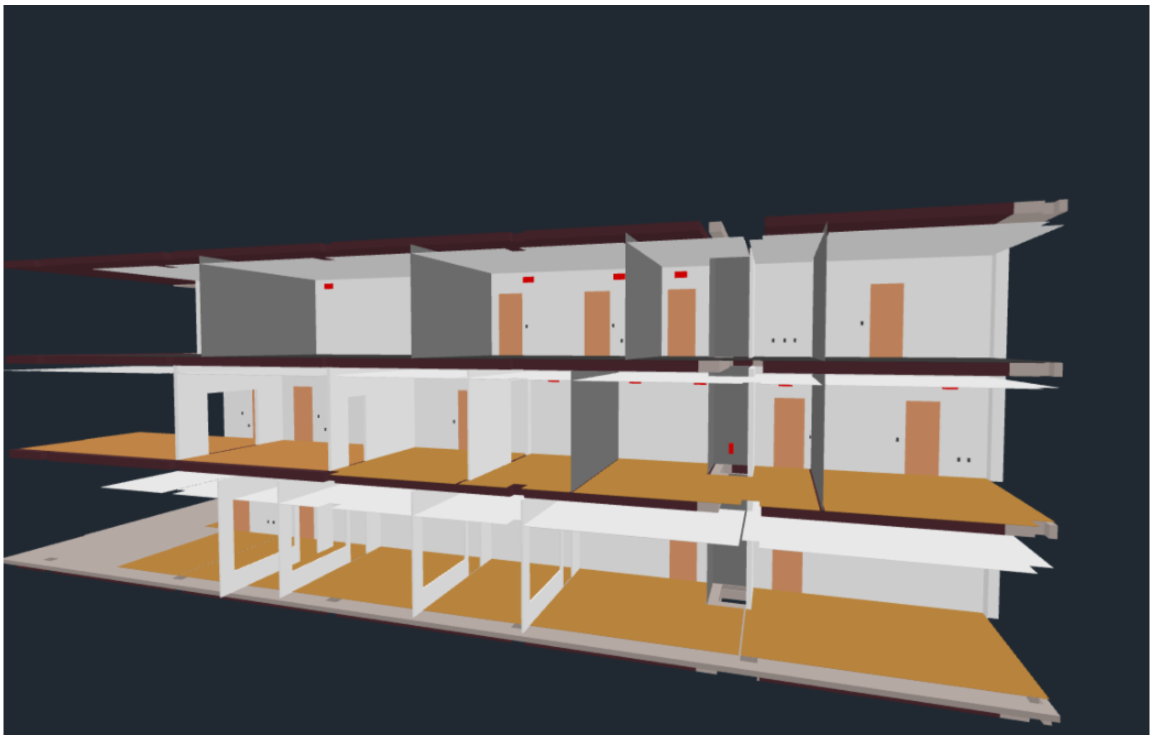


Figure 3.3. Building

design choice to keep the system decentralized and flexible, avoiding the need for manual assignment by admin in every case.

Technician Starts Maintenance: A technician decides to address the reported fault. Through the front-end, the technician selects the task (the specific asset fault) and initiates the maintenance process. This triggers another smart contract transaction, such as a "start maintenance" function for that asset. The contract will change the asset's status to "Under Maintenance" and record the technician's address as the one responsible for this job, Figure 3.5. It may also timestamp the start of the maintenance. This on-chain update prevents other technicians from also starting the same job (ensuring one technician per task) and provides accountability (we now know on-chain who is handling the issue). The DApp's UI can update in real time, for example: the asset changes to a different status in the model (e.g., Under Maintenance), and other technicians' interfaces will show that this task is no longer available. Also, users they see a status update ("Technician XYZ is fixing the issue") when they check the app, providing transparency that their request is being handled. Also, both technician and admin can see all the asset registered in the blockchain with the relative "Locate" button to locate assets in the building, this is shown in Figure 3.6.

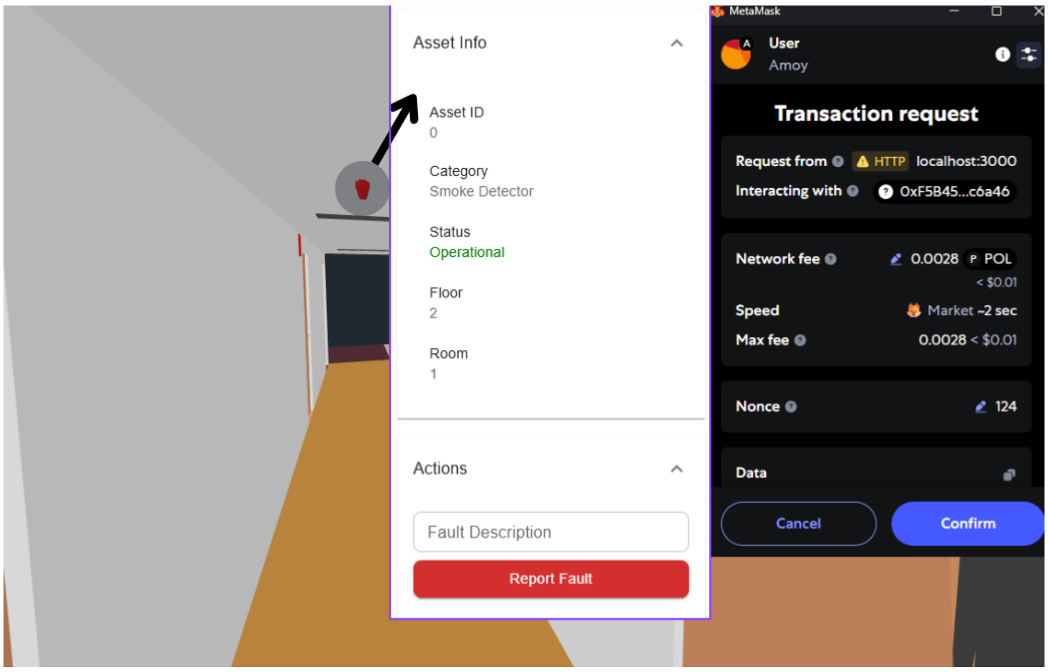
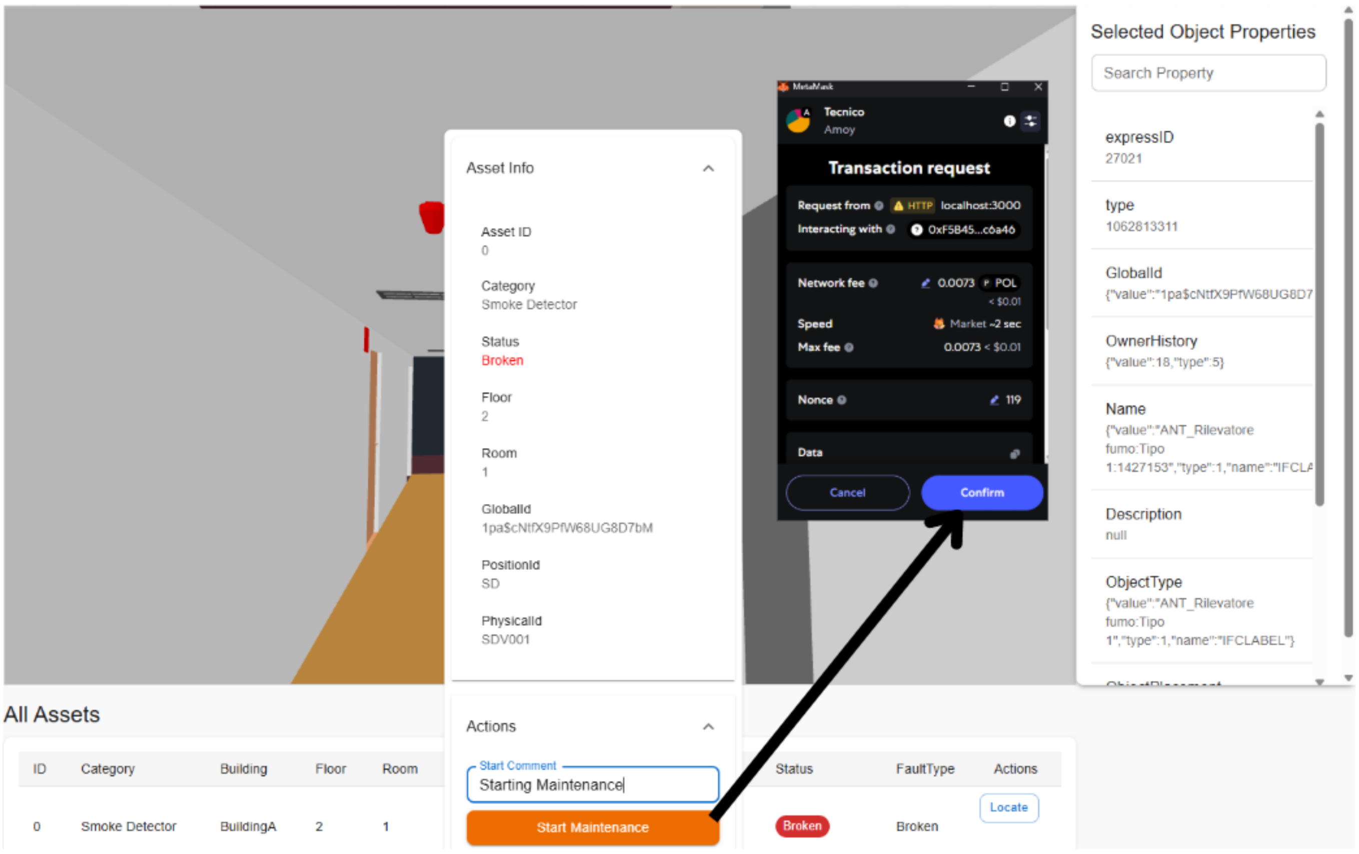


Figure 3.4. User Report Fault

Maintenance Work Performed (Off-Chain Human Step): At this point, the technician actually performs the required maintenance in the real world (this is outside the digital system, e.g., replacing the light bulb or repairing the equipment). While this physical work is being done, the DApp might not have much activity, but it's worth noting in the flow that the system can allow the technician to input some details. For example, if the maintenance requires replacing a part, the technician might prepare to input a new physicalId for the asset (since the physical device could have changed, see Section 3.5.1 on asset identification). The technician might also have a place to add comments about the maintenance done. These details, if any, would be submitted in the next step when closing the task.

Technician Completes Maintenance: After fixing the issue, the technician uses the DApp to mark the maintenance as completed. They may enter notes about what was done (e.g., "replaced broken part, tested working") and, if a part was replaced, provide the new physicalId of the asset now in place. When the technician submits this, a **transaction to the smart contract** is executed to mark completion. In Figure 3.7 we can see the technician sumbitting the task of completing the maintenance for a certain asset. The contract will update the asset's status back to "Operational" (or equivalent) and record the maintenance completion details: who the technician was, what time it finished, and any metadata (like the old vs new physicalId and technician's comment) in a maintenance record. Also, crucially, this action flags that this maintenance task is now **awaiting approval/payment**. In the contract's data model, there might be a record indicating



Asset Info

Asset ID
0

Category
Smoke Detector

Status
Broken

Floor
2

Room
1

GlobalId
1pa$cNtfX9PfW68UG8D7bM

PositionId
SD

PhysicalId
SDV001

Actions

Start Comment
Starting Maintenance

Start Maintenance

MetaMask

Tecnico
Amoy

Transaction request

Request from HTTP localhost:3000
Interacting with 0xF5B45...c6a46

Network fee 0.0073 POL
< $0.01
Speed Market ~2 sec
Max fee 0.0073 < $0.01

Nonce 119

Data

Cancel Confirm

Selected Object Properties

Search Property

expressID
27021

type
1062813311

GlobalId
{"value":"1pa$cNtfX9PfW68UG8D7

OwnerHistory
{"value":18,"type":5}

Name
{"value":"ANT_Rilevatore fumo:Tipo 1:1427153","type":1,"name":"IFCLA

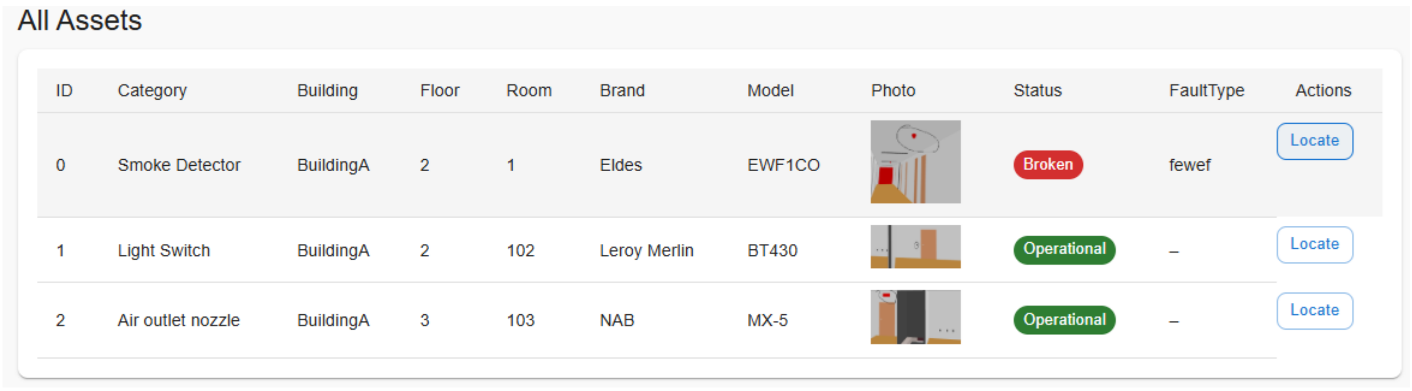
Description
null

ObjectType
{"value":"ANT_Rilevatore fumo:Tipo 1","type":1,"name":"IFCLABEL"}

All Assets

ID	Category	Building	Floor	Room	Status	FaultType	Actions
0	Smoke Detector	BuildingA	2	1	Broken	Broken	Locate

Figure 3.5. Technician starting maintenance

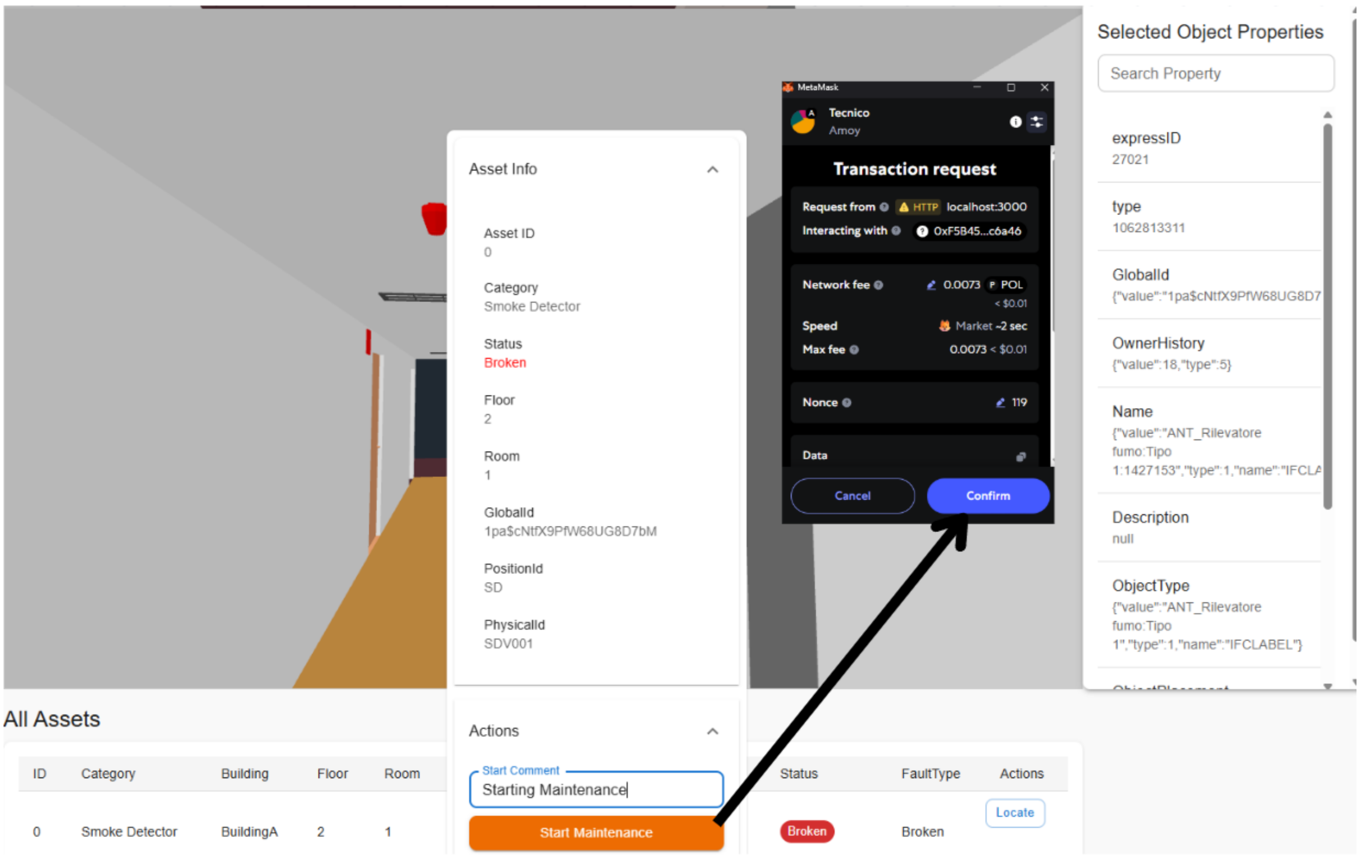


All Assets

ID	Category	Building	Floor	Room	Brand	Model	Photo	Status	FaultType	Actions
0	Smoke Detector	BuildingA	2	1	Eldes	EWF1CO		Broken	fewef	Locate
1	Light Switch	BuildingA	2	102	Leroy Merlin	BT430		Operational	–	Locate
2	Air outlet nozzle	BuildingA	3	103	NAB	MX-5		Operational	–	Locate

Figure 3.6. Locating Fault

"maintenance done, pending admin confirmation/payment" along with a suggested payment amount (if the technician is to be paid, they might input an expected cost or it's predetermined). At this stage, the heavy lifting by the technician is done. The system now has a logged record of a completed maintenance action, but payment is not released yet and it's awaiting the admin's review.



Asset Info

Asset ID: 0
Category: Smoke Detector
Status: Broken
Floor: 2
Room: 1
GlobalId: 1pa$cNtfX9PfW68UG8D7bM
PositionId: SD
PhysicalId: SDV001

Actions

Start Comment: Starting Maintenance

MetaMask — Tecnico, Amoy

Transaction request

Request from: HTTP localhost:3000
Interacting with: 0xF5B45...c6a46
Network fee: 0.0073 POL < $0.01
Speed: Market ~2 sec
Max fee: 0.0073 < $0.01
Nonce: 119
Data

Selected Object Properties

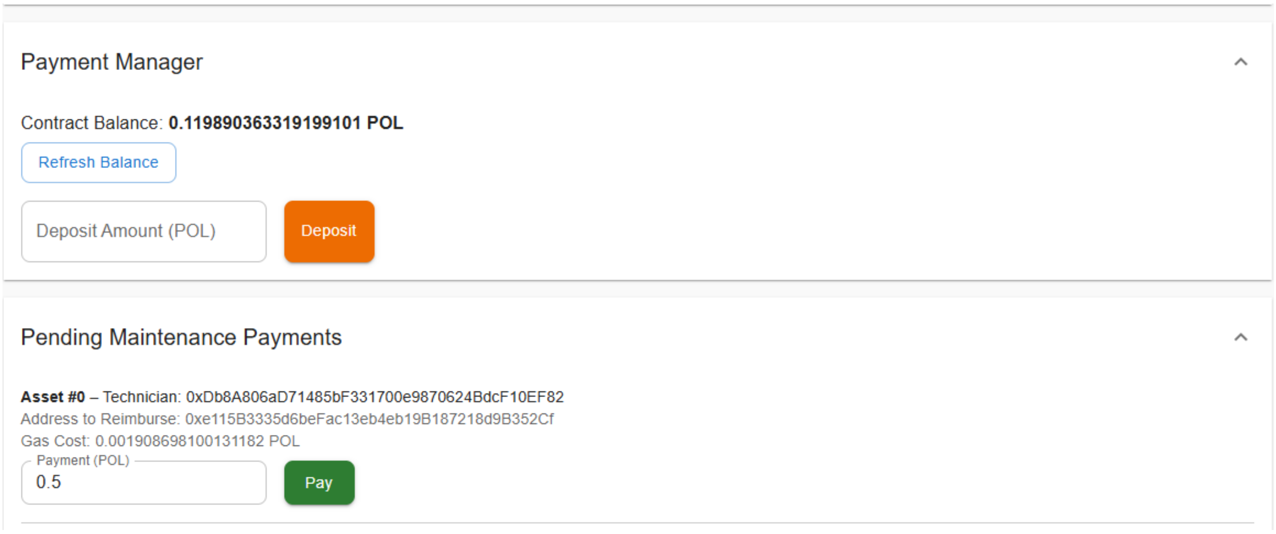
expressID: 27021
type: 1062813311
GlobalId: {"value":"1pa$cNtfX9PfW68UG8D7
OwnerHistory: {"value":18,"type":5}
Name: {"value":"ANT_Rilevatore fumo:Tipo 1:1427153","type":1,"name":"IFCLA
Description: null
ObjectType: {"value":"ANT_Rilevatore fumo:Tipo 1","type":1,"name":"IFCLABEL"}

All Assets

ID	Category	Building	Floor	Room	Status	FaultType	Actions
0	Smoke Detector	BuildingA	2	1	Broken	Broken	Locate

Figure 3.7. Technician completed maintenance

Administrator Verification & Payment Authorization: The final part of the workflow involves the **Admin** (e.g., building manager or owner) reviewing the completed maintenance. The admin will have an interface in the DApp showing all tasks that have been completed by technicians and are pending approval. The admin checks the details (they might verify on-site that the issue is resolved, or simply rely on the technician's report and maybe user feedback). Once satisfied, the admin authorizes the payment by entering the approved amount (or confirming the suggested amount) and confirming the task. This triggers the last important smart contract transaction, the one responsible to pay the technician for the job done. The smart contract, upon this call, will transfer the payment to the technician's address from the on-chain funds. Moreover, the architecture includes reimbursing the user's reporting cost (as we have designed), and the contract will also refund the user's gas cost at this point, sending a small amount back to the user's address. After executing the payments, the contract marks the task's record as closed/paid. This completes the on-chain workflow for the maintenance request. Figure 3.8 shows the interface for releasing funds held by the smart contract.


Payment Manager

Contract Balance: **0.119890363319199101 POL**

Pending Maintenance Payments

Asset #0 – Technician: 0xDb8A806aD71485bF331700e9870624BdcF10EF82
Address to Reimburse: 0xe115B3335d6beFac13eb4eb19B187218d9B352Cf
Gas Cost: 0.001908698100131182 POL
Payment (POL): 0.5

Figure 3.8. Admin pays technician

After enumerating these steps, tie them together in a concluding paragraph for this section. Emphasize how the **flowchart (Figure 3.2)** captures these interactions: the user, technician, and admin each interact via the front-end to invoke on-chain transactions that drive the process. The smart contracts orchestrate state changes and enforce rules at each step, while the front-end and (occasionally) the backend coordinate to provide a smooth user experience (such as updating status in real-time, etc.). This workflow demonstrates the **end-to-end operation of the DApp**, showing how a real-world maintenance procedure is translated into a series of digital interactions that ensure accountability, transparency, and automation (e.g., automatic payment) through the proposed architecture.

3.5 Key Design Decisions and Considerations

In designing this architecture, several important decisions were made to meet the project's goals and to address challenges unique to building maintenance in a decentralized context. This section highlights two key design considerations: the strategy for uniquely identifying building assets (to tie physical equipment to the digital system) and the choice of platforms/technology for deployment. These decisions significantly influenced the architecture's effectiveness and are discussed here at a conceptual level to justify why they were taken.

3.5.1 Asset Identification Scheme: physicalId & positionId

One notable design choice in the system is the use of two distinct identifiers for each asset in the building: a *physicalId* and a *positionId*. This subsection explains the reasoning behind this dual-ID scheme and how it is reflected in the architecture (without going into code specifics).

Definition of the IDs: Clarify what each identifier represents. The **physicalId** is an identifier for the actual physical device or component. It could be a serial number, a manufacturer ID, or any tag that uniquely marks that piece of equipment. For example, a specific fire extinguisher or a lamp might have a physicalId like *"Lamp_ABC123"*. On the other hand, the **positionId** denotes the asset's location or position within the building's context. This is a stable identifier tied to where the asset is installed, for instance, *"Floor2_Room10_LampSpot#1"* might indicate the light fixture in Room 10 on Floor 2, position 1. In the context of an IFC model, the positionId could relate to the BIM hierarchy (building > floor > room > spot) or even correspond to an IFC element GUID if that element's placement is fixed. Essentially, positionId encodes *where* the asset is, while physicalId encodes *what* the asset is as an individual unit.

Motivation for Dual Identifiers: Discuss why a single identifier was not sufficient. In building maintenance, assets can be moved, replaced, or swapped over time. If we only used a physicalId, tracking an asset's history would become confusing when that device is replaced with a new unit – the new device would have a different physicalId, breaking the historical continuity if we tried to attach it to the same record. Conversely, if we only used a location-based identifier (positionId), different physical units occupying that location over the years would be conflated as one "asset," which could obscure the fact that a new device with potentially different properties is now in place. By using both:

- The **positionId provides continuity in the maintenance log for a given location** in the building. No matter how many times a component is replaced at that spot, all maintenance records for that location can be aggregated, which is useful for facilities management (e.g., "this particular light socket has had 3 bulbs replaced in 2 years").

- The **physicalId allows differentiation of individual units**. Maintenance records can explicitly note that a physical device with ID X was removed/replaced on a certain date and a new device Y installed. This is crucial for asset inventory and warranty tracking (e.g., "Chiller unit serial *#SN1001* was installed in the rooftop location replacing *#SN1000*, which failed" – now if *#SN1001* fails, you know it's a different unit with its own warranty).

Architectural Role of Each ID: Explain how these IDs are used in the system architecture. The smart contract (asset registry) stores both identifiers for each asset, treating them as part of the asset's metadata. When a user reports a fault via the front-end's IFC model, the system likely uses the position-based info (from the model selection) to identify which asset record to update. The physicalId might be displayed to admins/technicians so they know exactly which device they're dealing with (especially if the device has a label). When technicians complete maintenance, if they replaced the device, they update the **physicalId** in the contract for that asset's record, while the **positionId** remains the same (because the location in the building hasn't changed). This way, the architecture cleanly separates the concept of "asset identity" into two layers: **physical identity** and **spatial identity**. Both are needed for a comprehensive maintenance system. The front-end's design also reflects this: it can highlight assets in the 3D model via positionId/globalId (for spatial context) while showing the physicalId in information panels for clarity about the equipment piece.

Benefits of the Dual-ID Approach: Summarize why this design improves the system. It enhances **data consistency** and **historical traceability**. For example, if a trend shows that every summer the AC unit in Room 301 fails, that is tied to positionId (the location) – perhaps that room has high usage or some environmental issue. Meanwhile, if a particular pump (physicalId) fails wherever it is installed, tracking by physicalId helps identify a faulty batch or model of equipment. In terms of architecture, this means our system can answer different queries: "What maintenance has been done in this location?" versus "What maintenance has been done on this particular device across locations?", which is powerful for a facility manager. Another benefit is during **initial asset registration**: having both IDs forces the data model to capture installation info (position) and device info separately, which is aligned with BIM practices. The positionId usually doesn't change unless you physically relocate the asset, which typically would be a new maintenance event itself.

Comparison to Alternatives: Optionally, mention what it would look like if only one ID were used, to reinforce the point. If only a physicalId were used, once device ABC123 is replaced by XYZ789, the system might treat it as a completely different asset with no link to past records at that site. If only a positionId were used, replacing a device would overwrite or merge details, potentially losing the fact that it's a new physical unit. Neither scenario is ideal for a long-term maintenance log. Thus, the dual identifier scheme is a deliberate architectural decision to model the reality of building maintenance more accurately.

With this asset identification scheme, we demonstrate how the architecture is customized for building management through its asset identification system. By implementing this approach we obtain a more resilient system which manages physical environment changes efficiently though it adds some complexity to the data model. It exemplifies the thought process of aligning the DApp's design with real-world requirements, which is critical in an academic project of this nature.

3.5.2 Platform and Deployment Considerations

Another important aspect of the architecture is the choice of technology platforms and how the system is deployed. This subsection briefly discusses decisions around the **blockchain network** and other technology choices that impact the architecture:

Choice of Blockchain Network: Reiterate that the DApp was developed and tested on **Ethereum Sepolia (testnet)** and then deployed to **Polygon Testnet Amoy** for testing use. The decision to use Polygon was strategic: Polygon is known for its scalability and low fees, which aligns with the need for frequent transactions in a maintenance system (reporting issues, status updates, payments, etc.) without incurring high costs. From an architectural standpoint, Polygon being EVM-compatible means we could write the smart contracts in Solidity once and use them on both networks with minimal changes. The architecture remains blockchain-agnostic to some extent (it could run on any EVM network), but choosing Polygon ensures the **system is economically feasible** to run. The decision showcases understanding of the **operational reality** which building maintenance often includes numerous small transactions, a high fee environment like Ethereum mainnet could discourage usage, whereas Polygon's environment encourages active participation by users and IoT devices alike.

Off-Chain Tech Stack Choice: Note the simplicity of the off-chain stack, Express.js and SQLite. This was a deliberate choice for ease of deployment and because the off-chain needs were lightweight. SQLite, being a file-based DB, removes the need for managing a separate database server, which fits the scale of this project. If in the future the off-chain component grew (more data or more concurrent use), one could switch to a larger database (like PostgreSQL) or a more robust server framework. However, architecturally, keeping the off-chain part minimal ensures the focus remains on the blockchain part for critical functions. It's also easier to maintain and can be run on modest infrastructure (even a small cloud VM or a Raspberry Pi, for example, could host the Express server for a pilot installation).

Integration of BIM (IFC) Technology: Including an IFC model viewer in the front-end was a design decision to leverage existing building data for maintenance. This required choosing a suitable IFC viewing library and ensuring the front end could handle rendering a potentially complex 3D model in the browser. The architectural consideration here was that the **front-end needed to handle potentially heavy graphical content**. This influenced using React (which can integrate with Three.js through IFC libraries) and making sure the user's device does some processing. It's worth noting that using IFC ties the system to having IFC model data available. In practice, that means initial setup of the DApp for a building would involve loading the building's IFC file into the viewer. The design expects that as an input, which is fine for our context. This decision roots the maintenance system deeply in the building's digital twin, providing a richer context than a traditional maintenance ticket system. From an academic perspective, it shows an interdisciplinary approach (combining BIM with blockchain).

Chapter 4

Implementation and Development

4.1 Introduction to the Implementation

In this chapter, we present the implementation of the proposed architecture. The system was realized as a decentralized web application with a layered structure, composed of a React front end, smart contracts deployed on a blockchain network, and an Express-based off-chain server. The overall structure is illustrated in Figure 4.1.

4.2 Development Tools and Environment

The development of this decentralized application (DApp) leveraged a modern web and blockchain tech stack. The smart contracts are written in Solidity (version 0.8.28) for execution on the Ethereum compatible blockchain, in our case Polygon. The frontend client is built in **JavaScript** using **React**, a popular library for building user interfaces [React]). React enables a **single-page application (SPA)** architecture with dynamic components and state management on the client side. On the server side, **Node.js** (JavaScript runtime) is used with **Express.js**, which is a fast and minimalist web framework for Node that provides a robust set of features for building web applications and APIs [Express]. The Express server interacts with an SQLite database for lightweight storage, offering a file-based relational database suitable for prototyping and low to moderate data volumes. The overall project is thus hybrid: Solidity for on-chain logic, and JavaScript/Node for off-chain and client logic.

Development and Blockchain Tools: For blockchain development, the project uses **Hardhat**, a development environment for Ethereum that streamlines compilation, testing, and deployment of smart contracts [hardhat]. Hardhat enables running a local Ethereum node for testing and includes plugin support (e.g., for Ethers.js library integration) to manage contract interactions. The **Ethers.js** library is used both in scripts and in the

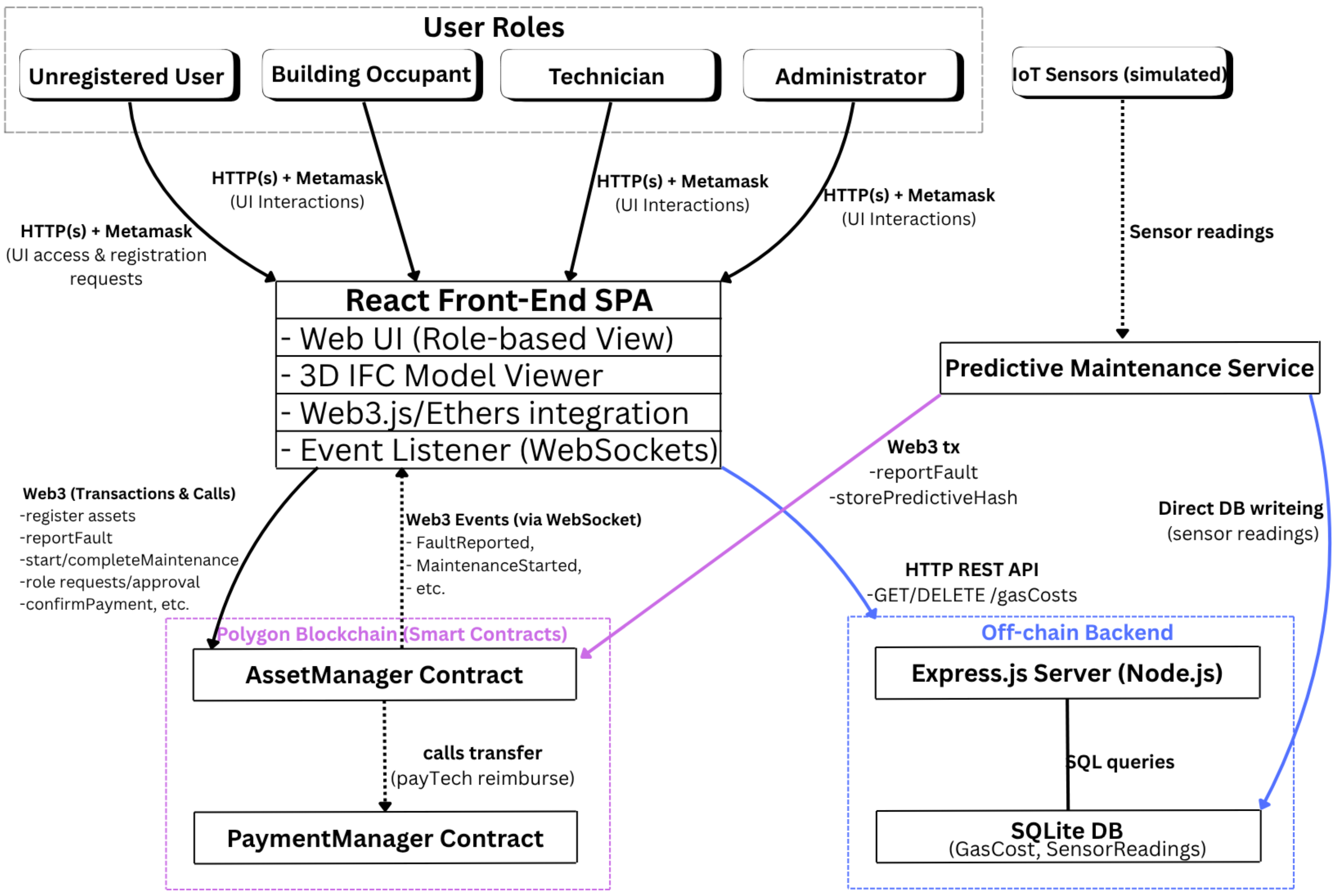

User Roles
Unregistered User
Building Occupant
Technician
Administrator
IoT Sensors (simulated)
HTTP(s) + Metamask (UI access & registration requests
HTTP(s) + Metamask (UI Interactions)
HTTP(s) + Metamask (UI Interactions)
HTTP(s) + Metamask (UI Interactions)
Sensor readings
React Front-End SPA
- Web UI (Role-based View)
- 3D IFC Model Viewer
- Web3.js/Ethers integration
- Event Listener (WebSockets)
Predictive Maintenance Service
Web3 (Transactions & Calls) -register assets -reportFault -start/completeMaintenance -role requests/approval -confirmPayment, etc.
Web3 Events (via WebSocket) - FaultReported, - MaintenanceStarted, - etc.
Web3 tx -reportFault -storePredictiveHash
Direct DB writeing (sensor readings)
HTTP REST API -GET/DELETE /gasCosts
Polygon Blockchain (Smart Contracts)
AssetManager Contract
calls transfer (payTech reimburse)
PaymentManager Contract
Off-chain Backend
Express.js Server (Node.js)
SQL queries
SQLite DB (GasCost, SensorReadings)

Figure 4.1. Diagram Dapp

React front-end for communicating with the Ethereum network (to deploy contracts, call functions, listen to events, etc.). During development, **MetaMask** was used as the in-browser wallet to manage accounts and sign transactions from the React app. For version control and project organization, the codebase is structured as a monorepo with separate sections for frontend, contracts, and backend. The **repository structure** is organized into folders such as /contracts (Solidity sources and Hardhat config), */frontend* (React application source code), and */backend* (Express server and database files). This separation aligns with the three-tier architecture of the system (client, on-chain, off-chain). The project also utilizes configuration files for environment-specific settings: for example, a *.env* file or *config* module holds network URLs (e.g., Infura endpoints for Polygon), contract addresses after deployment, and private keys (kept out of source code) for deployment.

Environment and Version Info: The smart contracts were developed and tested on Ethereum's Sepolia test network initially, then migrated to Polygon Amoy (an Ethereum Layer2 testnet) for testing. Polygon was chosen due to its **EVM-compatibility** and low transaction costs, providing faster and cheaper transactions compared to Ethereum mainnet [Crypto.com [2025]]. The front-end uses React 18 with modern React Hooks (for state and effects) and the interface is styled with **Material-UI (MUI)**, a component library that accelerates UI development. The IFC model viewer relies on **Three.js**, a JavaScript 3D engine, via a specialized IFC viewing library. Three.js is a powerful webGL-based library for rendering 3D graphics in the browser [SourceForge [2025]], and the IFC component library (@thatopen/components, part of the **IFC.js** toolkit) builds on **Three.js** to provide BIM-specific functionality [ThatOpen]. The Node/Express server was run on Node.js v16 LTS during development, and SQLite3 was used as the database engine (no separate DB server needed). In summary, the environment consisted of Node.js (for running development scripts, backend, and build tools), Hardhat for blockchain development, and React/Webpack for the front-end development server. All components were developed on a Linux environment, with cross-platform compatibility (the Node and browser-based components are OS-agnostic).

Development Workflow: We utilized Hardhat for compiling smart contracts and running tests locally. Hardhat's testing environment allows for Solidity tests, ensuring the AssetManager and PaymentManager contracts behaved correctly before deployment. The React app was bootstrapped with Create React App, providing a convenient development server with hot-reloading to iterate on the user interface. The development tools also included browser devtools for debugging the React app and MetaMask interactions, and Hardhat's console for directly invoking contract calls when needed. Source code management was done with Git, and project dependencies were managed via **npm** (Node's package manager). Tables from 4.2 to 4.6 provide a summary of the key tools, libraries, and their versions used in the implementation.

4.3 Front-End Implementation (React + IFC Viewer)

The front-end is a **React** single-page application that serves as the user-facing layer for all interactions. It was designed to be intuitive for non-technical building occupants, while still providing advanced functionality for technicians and administrators. The interface is organized around a **3D Building Model Viewer** and context-aware panels for asset information and actions. One of the distinguishing features is the integration of a **3D IFC model viewer** (IFC = Industry Foundation Classes). IFC is an open standard for representing building information models, containing data about building components in a software-neutral [Ren and Zhang [2020]]. The frontend's IFC viewer functionality allows users to explore the building visually while they can interact with building elements in components in their spatial context. For example, a user can rotate and zoom into

Blockchain Interaction and Deployment		
Tool/Library	**Version**	**Role in Project**
Ethers.js	6.13.5	Library for calling smart contracts, signing transactions (front-end + Node).
Web3.js	4.16.0	Alternative JS library for Ethereum (included for compatibility).
MetaMask	–	Browser wallet used for signing transactions (user side).
Infura	v3 API	Cloud node provider (RPC, WSS endpoints) for Polygon/Ethereum networks.
Polygon	–	EVM-compatible sidechain on which the DApp's contracts are deployed.

Table 4.1. Tools enabling blockchain calls, user authentication, and Polygon deployment.

the virtual building model and click on a specific asset (like a lamp or an HVAC unit) to select it. This triggers the display of that asset's details and relevant maintenance options.

Layout and Components: The React app uses a **responsive layout** with a top navigation bar and a main content area. The top bar includes the application title and a **WalletConnector** component (displayed as a user account icon) for managing blockchain wallet connection and user identity. The main content area is split between the 3D model view and various overlay panels or lists that appear based on user interactions and role. Key React components include the following components. Table 4.2 lists all the libraries

Front-End Tools		
Tool/Library	**Version**	**Role in Project**
React	18.2.0	Main front-end JS library for SPA, providing a component-based architecture.
Material-UI (MUI)	6.4.5	UI component framework for React, offering pre-styled elements and layout utilities.
Emotion (CSS-in-JS)	11.14.0	Styling library (used with MUI) for writing CSS within JavaScript.
Create React App	5.0.1	Toolchain to bootstrap and configure the React-based single-page application.

Table 4.2. Key libraries and versions used for the front-end (React) implementation.

and frameworks used in the front-end of the DApp, along with their version numbers. **IFCViewer** is a custom component encapsulating the IFC model rendering. It utilizes the IFC.js (Open BIM Components) library to load a *.ifc* model file of the building and render it with Three.js. In different words, this sets up a Three.js scene, camera, and controls, then parses the IFC geometry and properties. The IFCViewer component also implements picking logic: using Three.js raycasting, it detects when the user double-clicks on an object in the scene and maps that to a specific building element. The IFC model provides each element's unique GlobalId (an IFC-assigned UUID for the object). The viewer retrieves the GlobalId of the clicked element and calls an onObjectSelected callback with that ID. This allows the rest of the app to identify which asset (if any) corresponds to the selected model object.

The integration of the 3D BIM model allows users to select an asset in the building's

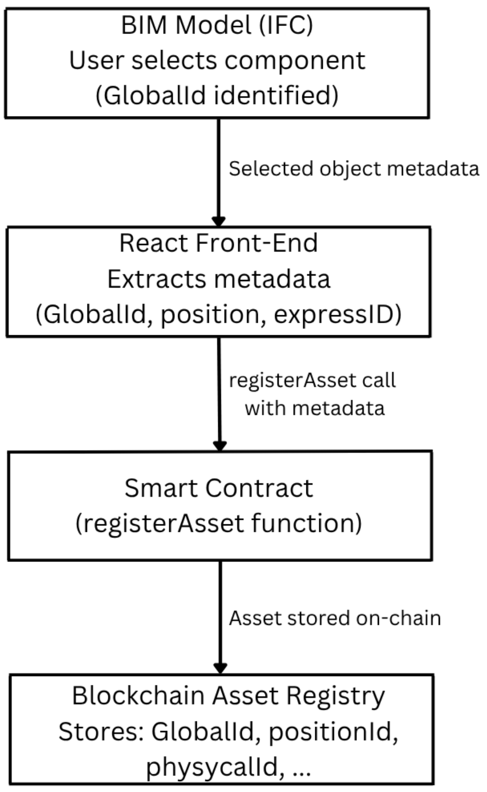


Figure 4.2. IFC to Blockchain

digital representation and trigger an on-chain registration. As shown in Figure **??**, the system extracts metadata such as GlobalId, positionId, and physicalId from the selected IFC element and submits it via a blockchain transaction to the AssetManager contract.

The use of this BIM viewer fulfills the goal of an intuitive, visual interface application and users can see the facility virtually and select assets by location rather than by cryptic IDs. The IFC viewer is powered by an open-source BIM toolkit built on Three.js, which provides ready-made tools for viewing and interacting with IFC models in the browser

[ThatOpen [2025]]. This integration demonstrates how a complex BIM model (which might be tens of thousands of polygons and many object metadata) can be handled on the client side and linked to blockchain records.

BIM Visualization Tools		
Tool/Library	**Version**	**Role in Project**
`@thatopen/components`	2.4.5	IFC-based library enabling a 3D viewer inside React, linking building elements to data.
`@thatopen/fragments`	2.4.0	3D features on top of Three.js for parsing and rendering IFC geometry.
Three.js	0.160.1	Core WebGL library used to render the IFC model in the browser.
web-ifc	0.0.66	Parser that reads IFC files in the browser, exposing building element data to React.

Table 4.3. Key libraries for IFC-based BIM integration and 3D visualization.

FloatingAssetPanel is a component that appears when an IFC object is selected. This is a floating info panel (rendered as a Material-UI Paper) positioned near the cursor, showing details and actions for the selected asset. It displays the asset's name, ID, current status (with color-coded text: e.g., green for *Operational*, red for *Broken* and yellow for *Under Maintenance*), and other metadata such as location (building, floor, room) and any model identifiers (like the IFC *GlobalId* and a *physical* tag ID). If the selected object does not correspond to any known asset on the blockchain, the panel indicates "No linked asset found on-chain," meaning that element is not managed by our maintenance system. For linked assets, the panel provides context-specific actions:

- For a **regular user** (building occupant) viewing an operational asset, the panel allows them to report a fault. There is a text field to describe the issue (fault description) and a **"Report Fault"** button. This triggers a transaction via Ethers.js to call the AssetManager.reportFault(assetId, description) function on the blockchain. On success, the UI will update the asset's status to Broken. The use of a visual click on the model to report a fault makes it easy for non-technical users to precisely identify the asset because they don't need to know any database IDs since the system automatically links the model's GlobalId to the on-chain asset ID.

- For a **technician**, if the asset is in Broken status, the panel will show a **"Start Maintenance"** button (and an optional text field to add a comment about the maintenance start). This calls the startMaintenance(assetId, comment) function in the contract, changing the asset's status to *Under Maintenance* and logging the start time and technician info. Once an asset is under maintenance and being viewed by the assigned technician, the panel then provides a **"Complete Maintenance"** button (with a field for an end comment). Clicking this calls completeMaintenance(assetId, comment) on-chain, which marks the maintenance as completed (asset status back to Operational) and internally flags the task as pending admin approval for payment. Technicians also have an option to input if they replaced a physical component: the panel includes a field to record a new physicalId and a button to submit replacePhysicalItem(assetId, newId) to the contract (this updates metadata like serial numbers in case the maintenance involved swapping out hardware).

- For an **administrator**, the FloatingAssetPanel might show administrative overrides such as the ability to **cancel a reported fault**. For instance, if a fault was reported erroneously or has been addressed outside the system, the admin can input a reason and click "Cancel Fault", which invokes the cancelFault(assetId, reason) function. This returns the asset status to Operational and emits an event so everyone is aware the fault case was closed. The panel consolidates asset information and actions in one place, improving the UX by providing immediate feedback and available actions in context.

AssetList is a component that lists all registered assets in a table form. This is primarily used by technicians and admins for an overview of assets and their statuses. The table displays columns such as Asset ID, Category (equipment type), Location (building-floor-room), Status, and assigned Technician (if under maintenance). For each asset in the list, context-specific action buttons are provided similar to the floating panel: e.g., an admin or technician viewing the list can also initiate or complete maintenance from here. Regular users (occupants) don't see this component since they interact through the 3D model instead. The AssetList is mainly shown to **Technician** and **Admin** roles (in fact, in our implementation it is rendered only if the user's role is Technician or Admin). This design choice was made to simplify the interface for end users, while giving power-users (tech/admin) a more direct tabular interface in addition to the 3D view. Each row in the AssetList also has a **"View History"** toggle. If expanded, it fetches the maintenance history for that asset (via AssetManager.getMaintenanceHistory(assetId) which returns an array of past maintenance records) and displays a list of past maintenance events (dates, technician, summary of actions).

WalletConnector is a component that manages user authentication and role. It appears as an icon (account avatar) in the top bar. When clicked, it opens a menu for the user to connect/disconnect their Ethereum wallet (MetaMask). Upon connecting, the DApp obtains the user's blockchain address and queries the AssetManager contract for that address's assigned role (by calling a getRole(address) view function). Based on the returned numeric code (0=Unregistered, 1=User, 2=Technician, 3=Admin), the front-end adjusts what the user can see and do. The WalletConnector also allows unregistered users to **register as a User**. If the contract indicates the address is not registered, an option **"Register as User"** is shown. Clicking it will send a transaction calling requestUserRegistration() on the contract, thereby adding the user to the pending registration list. The UI then indicates the registration is pending admin approval. Similarly, if the user is already a registered User, the menu will offer **"Request Technician Role"** (unless they have already requested it). This calls requestTechnicianRole() on-chain. By using on-chain role data and transactions for role changes, the front-end ensures that role-based access control is enforced by the smart contract, not just by the UI. However, the UI gives helpful prompts and hides unauthorized buttons. For example, if you are logged in as a Technician, you won't even see the admin-only controls in the interface (they are conditionally rendered only for role=Admin). This prevents accidental usage and guides the workflow.

AdminActions is this is a component shown only to administrator accounts (role=Admin). It presents an interface for all administrative functions in an organized manner (using Material-UI Accordions to group sections). The sections include: **User and Technician Approvals**, **Asset Registration**, **Funding & Payments**, and **System Monitoring**. In the User/Technician Approvals accordion, the admin can see lists of pending user registrations and technician role requests (fetched via getPendingUsers() and getPendingTechnicians() from the contract). Each entry has Approve/Deny buttons which call the respective contract functions approveUser(address) or denyUser(address) (for users) and approveTechnician(address)/denyTechnician(address) (for techs). When a user is approved as a User on-chain, the contract also triggers an automatic micro-payment of 0.01 POL to that user's address as a gas cost reimbursement (this uses the PaymentManager, see next section). The UI reflects this with a confirmation message. The **Asset Registration** section allows the admin to add new assets to the system. It provides a form to input all asset metadata: category (e.g., HVAC, Electrical), building name, floor number, room number, brand/model, an IPFS hash (if an associated document or model fragment is stored on IPFS), the IFC GlobalId for linking to the BIM model element, a positionId (a human-readable stable location identifier), and a physicalId (e.g., a serial number or barcode of the physical equipment). Upon form submission, it calls the AssetManager.registerAsset(...) function, which creates a new asset entry on-chain and emits an event. Successful registration updates the asset list. By capturing the GlobalId at registration, the system cements the link between the 3D model and the blockchain record – when the IFCViewer later provides a GlobalId on click, the front-end can find which on-chain asset corresponds to it. The AdminActions component also has a section for **Funding the Payment Pool**, where the admin can view the current balance in the PaymentManager contract and deposit funds (in POL) to ensure there is money available for technician payouts and user reimbursements. This calls a deposit() function on the PaymentManager (a simple payable function to transfer POL into it).

Another critical part of AdminActions is the **Maintenance Payments** management. After a technician completes maintenance on an asset, the contract marks the maintenance record as readyForPayment. The admin interface fetches all such pending payments (by scanning all assets for any CompletedMaintenance records that are ready and not yet paid). These are shown in a list with details: asset ID, technician's address, and the timestamp of completion. The interface also integrates data from the off-chain server here: it queries the Express API at /gasCosts to retrieve any logged gas cost that a user spent for reporting that asset's fault. If available, it shows the address of the user who reported the fault and the gas cost they incurred (in POL). The admin can then input the amount to pay the technician (e.g., the agreed maintenance fee) and click **"Pay"**. This triggers a call to the AssetManager.confirmPayment(assetId, techFee, userAddr, userGasCost) function (through Ethers.js). In the contract, this function in turn calls PaymentManager to transfer the specified techFee to the technician's address and, if userGasCost is > 0, to reimburse the user who reported the fault. After the transaction is confirmed, the UI will move that maintenance task from "pending" to "paid" (there is another accordion listing paid maintenance history). The admin's payment interface ensures technicians are compensated and users are reimbursed, with a single on-chain transaction per maintenance task. Importantly, because the gas cost data was fetched from off-chain storage, the contract did not have to compute or store the gas used – the admin just passes it in as a parameter. The design choice to combine on-chain events with off-chain data fetching provides a balance between **trustlessness** and **cost-efficiency**: critical state changes (asset status changes, role changes, etc.) are on the blockchain, but supplemental data (like exact gas spent by whom) can be kept off-chain until needed. The React app clearly separates these concerns: for authoritative data it queries the smart contracts (via Ethers providers), and for auxiliary data it queries the Express REST API.

TechnicianActions is a smaller component for technicians that shows a summary of their work. Specifically, it lists "My Completed Jobs," i.e. all maintenance tasks that the logged-in technician has finished and whether they have been paid. This is done by calling a view function getTechnicianCompletedAssets(techAddr) on the AssetManager to get a list of asset IDs the tech worked on, then for each, retrieving its maintenance records and checking payment status. The UI then displays each asset ID with either "Paid on [date] amount X" or "Payment pending" etc. This gives technicians a transparent view of their work and earnings. It's essentially a convenience feature since the same data could be obtained by scanning events or calling the contract directly, but a UI listing improves usability.

Role-Based Views and Routing: The front-end heavily uses conditional rendering instead of multi-page routing, given the SPA nature. Depending on the current user's role (which is stored in a React state after login), different components are shown or hidden. For example, when an Admin is logged in, the AdminActions and AssetList components are rendered, whereas a normal User would not have those in the DOM at all. This approach ensures that the interface remains uncluttered and appropriate for each user type. Unregistered users (not logged in with MetaMask or not approved yet) will see a minimal interface – basically just the model viewer and the wallet connector prompting them to connect and register. Once registered, their available actions increase. The application does not use URL-based navigation; all interaction happens in one view, but React's state and conditional components create an experience of distinct "screens" (e.g., an admin essentially has an asset management dashboard, whereas a user has a reporting interface).

Wallet Integration: The React app uses Ethers.js to integrate with MetaMask. When the user clicks "Connect Wallet", the app prompts MetaMask and fetches the user's account. It then automatically calls a function to query the smart contract for the user's role, as described. The app also listens for account or network changes (MetaMask can notify the DApp if the user switches accounts or networks) to update the UI accordingly. The addresses of deployed contracts (AssetManager and PaymentManager) and their ABIs are configured in a central config.js file, which the React components import. This config is updated with the Polygon deployment addresses so that the front-end knows where to direct its calls. For reading data and events, the app uses both Web3Provider (which routes through MetaMask for transactions) and a WebSocketProvider (Infura's WebSockets endpoint for Polygon) for listening to events. In our implementation, on app load, we establish a WebSocket connection to the AssetManager contract and set up listeners for events like FaultReported, MaintenanceStarted, MaintenanceCompleted, etc. When those events fire (meaning some state change happened on-chain), the listener callbacks will refresh the local asset list state to keep the UI in sync. This event-driven update mechanism ensures that if, say, a technician starts maintenance (triggering a MaintenanceStarted event), the admin's view and others will update the asset's status to "Under Maintenance" in near real-time without needing a page refresh. It makes the DApp reactive to blockchain changes, which is important in a decentralized setting where multiple users might be interacting concurrently.

In summary, the front-end implementation delivers a rich, role-sensitive UI on top of a complex backend. It combines declarative UI components (React/MUI) with blockchain interactions (via Ethers.js) and a 3D BIM model viewer to meet the usability and transparency goals. By allowing visual selection of assets and providing immediate feedback (e.g., updating model colors or status labels after transactions succeed), the UI helps bridge the gap between the physical facility (represented by the BIM model) and the digital ledger (the blockchain records).

4.4 Smart Contract Implementation

The on-chain component of the system is realized with two smart contracts written in Solidity: **AssetManager** and **PaymentManager**. AssetManager handles the core maintenance logic and data (assets, faults, roles, maintenance records), while PaymentManager is a dedicated vault and payment processor for financial transactions (holding funds, paying technicians, reimbursing users). Dividing the functionality across two contracts improves **modularity** and **security**: the PaymentManager contract can be kept simple and narrowly scoped to monetary operations, and it is only callable by the AssetManager (through known interfaces), reducing the risk of misuse. Class diagrams of both smart contract are shown in Figure 4.3.

Smart Contract Development		
Tool/Library	**Version**	**Role in Project**
Solidity	0.8.28	Language for Ethereum-compatible contracts (AssetManager, PaymentManager).
Hardhat	2.22.18	Framework for compiling, testing, and deploying the Solidity contracts.
Hardhat Toolbox	5.0.0	Plugin set (includes Ethers.js, debugging utilities) simplifying contract workflows.

Table 4.4. Smart contract-related frameworks and tools used for on-chain logic.

4.4.1 AssetManager Contract

This is the primary smart contract that represents the state of building assets and the workflow of maintenance. It contains data structures to track assets, user roles, maintenance history, and events to signal important changes. Key components of AssetManager include the following.

Roles and Access Control: An enum *Role{Unregistered, User, Technician, Admin}* defines the possible roles for addresses. The contract maintains a mapping userRoles(address) -> Role and functions for role requests and approvals. Only an address with the Admin role can approve/deny user registrations or technician promotions. The contract uses **modifier-based access control** (onlyAdmin, onlyTechnician, onlyUser) on functions to ensure that only authorized roles can invoke certain actions. For instance, reportFault is onlyUser and this means that only a registered building user can report a fault (this prevents a random technician from marking something broken). Similarly, *startMaintenance* and *completeMaintenance* are onlyTechnician (only a technician can perform those), and the various approval functions are onlyAdmin. Role management functions in AssetManager:

- *requestUserRegistration()* allows an unregistered user to apply to join;
- *approveUser(address)* and *denyUser(address)* let the admin finalize the registration (assign Role.User or reject);
- *requestTechnicianRole()* is for Users to request upgrade to technician
- *approveTechnician(address)/denyTechnician(address)* for the admin to grant or refuse the Technician role. These maintain internal lists of pending requests (arrays of addresses) so the admin can query who is awaiting approval.

When a user is approved, an event *UserApproved(address)* is emitted (and the PaymentManager is instructed to reimburse initial gas), and similarly *TechnicianApproved(address)* is emitted for techs. The contract also provides a view *getRole(address)* returning the enum value, which the front-end calls on login to get the user's current role.

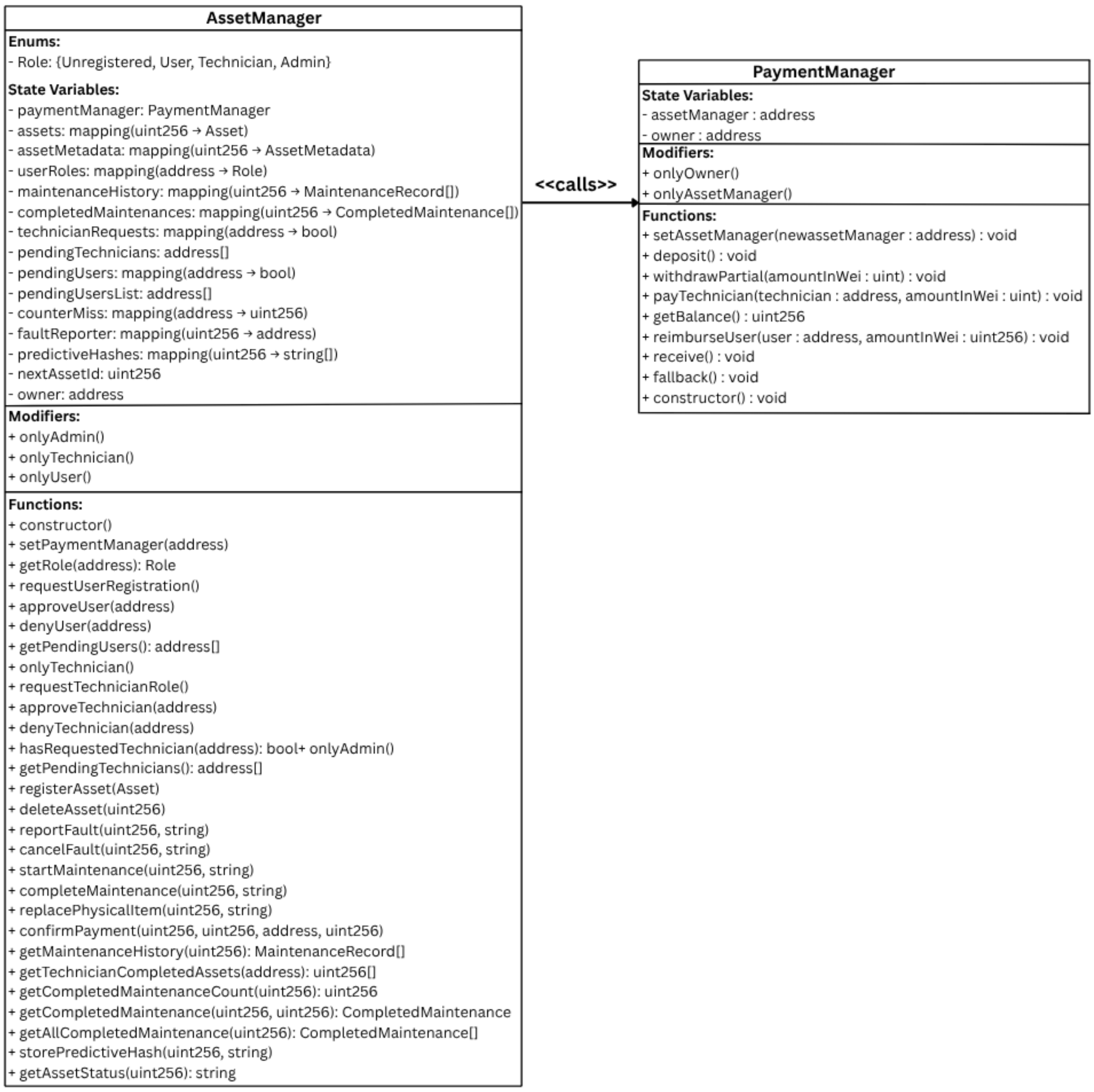

AssetManager
Enums:
- Role: {Unregistered, User, Technician, Admin}
State Variables:
- paymentManager: PaymentManager
- assets: mapping(uint256 → Asset)
- assetMetadata: mapping(uint256 → AssetMetadata)
- userRoles: mapping(address → Role)
- maintenanceHistory: mapping(uint256 → MaintenanceRecord[])
- completedMaintenances: mapping(uint256 → CompletedMaintenance[])
- technicianRequests: mapping(address → bool)
- pendingTechnicians: address[]
- pendingUsers: mapping(address → bool)
- pendingUsersList: address[]
- counterMiss: mapping(address → uint256)
- faultReporter: mapping(uint256 → address)
- predictiveHashes: mapping(uint256 → string[])
- nextAssetId: uint256
- owner: address
Modifiers:
+ onlyAdmin()
+ onlyTechnician()
+ onlyUser()
Functions:
+ constructor()
+ setPaymentManager(address)
+ getRole(address): Role
+ requestUserRegistration()
+ approveUser(address)
+ denyUser(address)
+ getPendingUsers(): address[]
+ onlyTechnician()
+ requestTechnicianRole()
+ approveTechnician(address)
+ denyTechnician(address)
+ hasRequestedTechnician(address): bool+ onlyAdmin()
+ getPendingTechnicians(): address[]
+ registerAsset(Asset)
+ deleteAsset(uint256)
+ reportFault(uint256, string)
+ cancelFault(uint256, string)
+ startMaintenance(uint256, string)
+ completeMaintenance(uint256, string)
+ replacePhysicalItem(uint256, string)
+ confirmPayment(uint256, uint256, address, uint256)
+ getMaintenanceHistory(uint256): MaintenanceRecord[]
+ getTechnicianCompletedAssets(address): uint256[]
+ getCompletedMaintenanceCount(uint256): uint256
+ getCompletedMaintenance(uint256, uint256): CompletedMaintenance
+ getAllCompletedMaintenance(uint256): CompletedMaintenance[]
+ storePredictiveHash(uint256, string)
+ getAssetStatus(uint256): string
<<calls>>
PaymentManager
State Variables:
- assetManager : address
- owner : address
Modifiers:
+ onlyOwner()
+ onlyAssetManager()
Functions:
+ setAssetManager(newassetManager : address) : void
+ deposit() : void
+ withdrawPartial(amountInWei : uint) : void
+ payTechnician(technician : address, amountInWei : uint) : void
+ getBalance() : uint256
+ reimburseUser(user : address, amountInWei : uint256) : void
+ receive() : void
+ fallback() : void
+ constructor() : void

Figure 4.3. CLass Diagram Smart Contracts

Asset Data Model: Each asset in the system is represented by two structs: Asset and AssetMetadata. The Asset struct holds the primary mutable state of an asset, for example an id (uint identifier), a status string (Operational, Broken, Under Maintenance), the address of the currently assigned technician (if under maintenance), a timestamp of last maintenance, a faultType description if it's broken, and a boolean isDeleted flag. The separate AssetMetadata struct contains more static info: category, building, floor, room, brand, model, an ipfsHash for any associated documents, the globalId (IFC model GUID for linking), *positionId* (a stable human label for location), and physicalId (like a serial number). These metadata fields are set on registration and not frequently changed (except *physicalId* might change if an asset is replaced during maintenance). In the contract, we have mappings assets(uint => Asset) and *assetMetadata(uint => AssetMetadata)* keyed by asset ID. The asset IDs are generated sequentially (the contract has a nextAssetId counter that increments on each new registration). The separation into two structs/mappings is partly to avoid stack too deep errors and also to logically separate frequently-updated vs mostly-constant data. The registerAsset function populates both mappings for a new ID and emits an AssetRegistered event. The contract also allows an admin to logically delete an asset (deleteAsset(id) setting isDeleted flag and emitting AssetDeleted), in case an asset is permanently removed from service.

Fault Reporting and Maintenance Workflow: This is the core logic enabling the maintenance use case. When a user notices an issue with an asset, they invoke *reportFault(uint assetId, string faultType).* The function checks that the caller is a User and that the asset exists and is currently "Operational". Then it sets the asset's status to "Broken" and records the fault type (the description provided). It emits a FaultReported(assetId, "Broken", faultType) event to notify off-chain listeners.

Note that in this design, the contract does not assign a technician at fault report time, it simply marks it broken. Assignment of a technician could be handled off-chain by an admin or by a future extension (like auto-assigning the next available technician). In our system, a technician would see the asset is Broken (via the front-end) and then proceed to address it.

The next step is *startMaintenance(uint assetId, string startComment)*, callable by a Technician. This checks the asset is in Broken status (so the fault was acknowledged). It then creates a new entry in the maintenance history: we keep an array *maintenanceHistory[assetId]* of struct MaintenanceRecord (containing start time, end time, technician addr, oldPhysicalId, newPhysicalId, and a comment).

On start, it appends a MaintenanceRecord with **startTime** = now, **endTime** = 0 (meaning ongoing), technician = msg.sender, oldPhysicalId set to the current assetMetadata.physicalId (so we log what physical component was there), and the technicianComment set to the startComment. The asset's status is updated to "Under Maintenance" and the asset's technician field is set to msg.sender (to mark who is working on it). Then an event MaintenanceStarted(assetId, technicianAddr) is emitted.

Once the technician finishes the work, they call *completeMaintenance(uint assetId, string endComment).* This requires the asset to be "Under Maintenance" and that a MaintenanceRecord exists with endTime=0 (meaning the maintenance was indeed started and not yet completed). The function then updates that record's endTime to now and, if an endComment is provided, attaches it (in our implementation, it simply overwrites or appends to the technicianComment). It sets the asset's status back to "Operational", clears the faultType, and updates lastMaintenanceTimestamp. Most importantly, it creates an entry in a separate completedMaintenances[assetId] array (mapping asset -> list of CompletedMaintenance structs).

A CompletedMaintenance struct **stores info** about a finished maintenance task that might need admin actions: it has fields like readyForPayment (bool), technician (address), suggestedAmount (if the tech suggested a fee, but in our case we left it as 0 for admin to decide), isPaid (bool), paymentTimestamp, paidAmountWei, and fields to record if a user was reimbursed for this maintenance.

On completion, we push a new CompletedMaintenance with readyForPayment=true, technician = msg.sender, and the rest default (payment info empty). This data structure accumulates all maintenance jobs per asset, which the admin can later review. Finally, *MaintenanceCompleted(assetId, technician, timestamp)* event is emitted. There are additional helper functions in this workflow: *cancelFault(uint assetId, string reason)* allows an admin to cancel a reported fault before maintenance starts (resetting status to Operational and emitting FaultCancelled(assetId, reason)).

Also *replacePhysicalItem(uint assetId, string newPhysicalId)* for a technician to log that they swapped a device during an ongoing maintenance (it updates the assetMetadata.physicalId and the current MaintenanceRecord's newPhysicalId field). This ensures traceability of component replacements and the history will show old and new serial numbers.

Payments and Reimbursements Triggers: The AssetManager itself does not hold or transfer funds (that's PaymentManager's job), but it does orchestrate when payments should happen. The key function is *confirmPayment(uint assetId, uint technicianWei, address faultReporter, uint faultCostWei).*

This function is onlyAdmin, after verifying a maintenance is ready for payment and not already paid, it invokes the PaymentManager: first paymentManager.payTechnician(techAddress, technicianWei) to transfer the technician's fee, then if faultCostWei>0 it calls paymentManager.reimburseUser(faultReporter, faultCostWei).

It then marks the CompletedMaintenance record as paid (isPaid=true, stores the payment timestamp and amount, and change readyForPayment to false). We emit events for important things like UserApproved, TechnicianApproved, AssetRegistered, FaultReported, MaintenanceStarted, MaintenanceCompleted, and also PredictiveHashStored (discussed later).

These events serve as **log entries** on the blockchain that the off-chain systems or any observer can subscribe to, thereby creating an immutable audit trail of all significant maintenance events. Indeed, one of the system requirements was to have **transparent maintenance logs on-chain** by emitting events and also by making the state queryable (functions like getMaintenanceHistory, getCompletedMaintenance, etc.), the blockchain becomes the source of truth for the maintenance history.

The AssetManager contract thus encapsulates a **finite-state workflow** for assets (Operational -> Broken -> Under Maintenance -> back to Operational) with appropriate role checks at each transition. It uses on-chain data to enforce that sequence and uses events to inform external systems. The logic ensures that only valid transitions happen (e.g., you cannot start maintenance on an asset that isn't broken, or complete maintenance that wasn't started), which prevents inconsistent states. All of this is done transparently:

because the contract is on Polygon, all participants can verify these state changes on a public ledger, fulfilling the transparency and trust goals for the maintenance process.

4.4.2 PaymentManager Contract

The PaymentManager is a much simpler contract whose only purpose is handling funds (denominated in the native cryptocurrency of the chain, which on Polygon is POL). It is **owned by the admin** and linked to the AssetManager. The keys aspects of PaymentManager are:

It stores an *owner* address (set to the deployer/admin in constructor) and an assetManager address (which must be set after deployment). The modifiers onlyOwner and onlyAssetManager restrict who can call certain functions. The AssetManager address is given permission to call the payout functions.

Deposit & Balance: The admin (owner) can deposit funds into PaymentManager by sending a transaction with value or calling the *deposit()* function. PaymentManager does not have any complex logic on deposit; it uses Ethereum's standard ability to hold a balance. We included a getBalance() view function to retrieve how much POL it currently holds. The admin can also withdraw funds (there's a withdrawPartial(amount) function in code for owner, though it might not be used in practice if admin manages funds externally).

Paying Technicians: The function *payTechnician(address technician, uint amountInWei)* transfers the specified amount from PaymentManager's balance to the technician's address. It is marked onlyAssetManager, meaning only our AssetManager contract (once set) can call it – technicians cannot directly call PaymentManager to pay themselves. This function also checks that the contract's balance is sufficient before transferring. The transfer uses the transfer or call method (in our case, *payable(technician).transfer(amount)* as shown), which sends the funds out. If it fails (insufficient balance), it will revert to ensure no partial logic goes through.

Reimbursing Users: Similarly, *reimburseUser(address user, uint amount)* is available for AssetManager to call. This will transfer the specified amount to the user (presumably covering the gas fees they spent in maintenance reporting). We separated this from payTechnician for clarity, though both are simple transfers. Emitting an event for payments was optional (the code comments note we could emit TechnicianPaid or GasReimbursed events). Even without explicit events, these transfers will be visible as internal transactions on the blockchain.

Security considerations: By isolating the funds in PaymentManager, we reduce the attack surface on the main AssetManager. Even if, hypothetically, a vulnerability existed in AssetManager's logic, the funds can only move if PaymentManager receives a valid call from AssetManager. The admin linkage is set up by calling PaymentManager.setAssetManager(assetManagerAddress) after both contracts are deployed. Conversely, AssetManager stores a reference to PaymentManager set via setPaymentManager(address) (only callable by admin). This mutual reference setup is done once during deployment and initialization. After that, the AssetManager's calls like paymentManager.payTechnician(...) route to the correct contract.

4.4.3 Final Thought

Contract Relationships and Data Flow: The two contracts work in tandem during the payment phase of the maintenance workflow: AssetManager accumulates the intention to pay (in its CompletedMaintenance record and confirmPayment function), and PaymentManager executes the value transfers. This design also means we could upgrade or replace the PaymentManager (if needed to move to a different funding model, for example) without affecting AssetManager's other logic, since the interface is abstracted to a few calls. All other logic (registration, reporting, status changes, history logging) lives in AssetManager.

We also carefully considered **gas costs and optimization** in the contract implementation. Each on-chain operation (especially ones storing data) costs gas, and since we expect possibly many maintenance events, we stored only necessary information on-chain. For instance, instead of storing long text for comments or descriptions on-chain extensively, we keep them reasonable (short strings like "Lamp broken" or "Replaced filter"). The sensor readings themselves (which can be high-frequency data) are not stored on-chain at all, but only their hashes (discussed below). This aligns with common blockchain design practices: **store only hashes or pointers to large data, not the data itself**, due to cost [Stackexchange [2019]]. Indeed, storing even a single 32-byte hash on Ethereum costs around 20,000 gas [Stackexchange [2019]], so one would not put raw sensor logs on-chain. Our solution is to use the chain for integrity (hashes) and use the off-chain database for raw data, achieving a balance. This principle is evident in our contracts where the predictiveHashes mapping holds arrays of hash strings for each asset but no raw sensor values.

In summary, the smart contracts implement a robust, self-contained logic for asset lifecycle in maintenance. They ensure **integrity** (because all critical changes are on-chain and thus tamper-proof), **role-based security** (only authorized roles can do certain actions, enforced by code), and **financial automation** (automatic payouts and reimbursements once admin triggers a single function). By deploying these contracts on the Polygon blockchain, we benefit from the decentralization and security of blockchain – no central server can secretly alter a maintenance record, and all stakeholders (users, technicians, auditors) can verify the history of events. The contract code (provided in Appendix B) was carefully tested for edge cases (e.g., double reporting faults, completing maintenance out of order, etc.) to ensure the state machine cannot be broken by improper usage.

4.5 Off-Chain Backend and Predictive Maintenance

While the blockchain contracts provide the trust and transparency backbone, certain aspects of the system are handled off-chain by a **Node.js/Express server** and a **Predictive Maintenance service**. These components manage data that either does not need to be on-chain or is impractical to put on-chain, and they complement the DApp by providing automation (predictive fault detection) and cost-saving measures (logging gas usage off-chain).

Express.js Server and SQLite Database: The Express server exposes a simple RESTful API used mainly by the admin interface and the predictive service. It serves two primary purposes:

1. **Gas Cost Logging**: Every time a user reports a fault (which is a blockchain transaction they pay for), we want to record how much gas they spent so that the admin can later reimburse that cost. Capturing gas cost via a smart contract would itself cost gas and add complexity, so instead the **front-end** (or a script) communicates with the Express server to log this information in a database. The server has an endpoint **GET /gasCosts** which returns a list of recorded gas expenditures. These records include the assetId for which the fault was reported, the user's address, and the cost in Wei (smallest unit of Ether/MATIC) that the user spent.

   In practice, when a user calls reportFault via MetaMask, we can get the transaction receipt and gas used on the client side. The front-end then sends an HTTP request to our server, e.g., **POST /gasCosts** with JSON assetId, user, costWei. The server stores this in the SQLite database (in a table, e.g., GasCosts(assetId, user, costWei)). Then, when an admin is deciding on payments, the front-end calls GET /gasCosts to retrieve all entries, and filters for the relevant asset. In our implementation, the AdminActions component's *loadGasCostsFromServer()* fetches from *http://localhost:4000/gasCosts* and maps the list into an in-memory map keyed by assetId. This provides the costData (user and cost) that we displayed in the "Pending Payments" section for each asset. After the admin processes a payment and reimbursement on-chain, the front-end calls a **DELETE /gasCosts** endpoint to remove that entry (it sends the assetId and user in the query, and the server deletes that row).

   This cleanup prevents reusing the same record twice and keeps the off-chain log lightweight. The use of an off-chain database for gas records drastically reduces on-chain storage requirements while still maintaining integrity in a practical sense (the admin is trusted to use the correct values; even if not, any user can see on-chain whether they were reimbursed or not). It is worth noting that in a fully trustless design, one might record a hash of the gas receipt on-chain to prove the value later. However, since the admin is anyway a trusted party in terms of approving payments, a simpler approach suffices. This aligns with a **hybrid architecture** principle: critical state transitions are on-chain, but supplementary data is off-chain for efficiency.

2. **Sensor Data Storage for Predictive Maintenance**: The SQLite database also contains tables for **predictive maintenance sensor readings**. In this project, we simulate sensors (e.g., temperature sensors on equipment) to demonstrate predictive maintenance integration. The predictive maintenance microservice (described below) inserts sensor readings into the database via the Express API or direct DB access. For example, a table *SensorReadings(assetId, timestamp, value)* might store temperature values for a given asset over time. Storing these on-chain would be infeasible due to volume and cost, so we keep them in SQLite.

   The server retrieves recent readings for display (e.g., an admin could query the last week's data if needed). In our setup, the predictive service actually runs in the same environment and can write to SQLite directly, so the Express endpoint for sensor data is not strictly necessary for function, but could be used if one wanted to decouple the services.

Backend Tools		
Tool/Library	**Version**	**Role in Project**
Node.js	18 LTS	JavaScript runtime environment executing the Express server and scripts.
Express.js	5.1.0	REST API server enabling off-chain data access (gas costs, sensor logs, etc.).
SQLite	3.x	Lightweight file-based SQL database storing off-chain records.
better-sqlite3	11.9.1	Node.js library for synchronous, performant SQLite interactions.
dotenv	16.5.0	Loads environment variables for config (e.g., private keys, Infura keys).

Table 4.5. Key dependencies for the Node.js/Express backend and SQLite database.

The Express server and database thus act as the **off-chain data layer** for our DApp. The database schema is fairly simple: one table for **GasCosts**, one for **SensorReadings**. We also included an endpoint **GET /resetDB** during development to clear the database (this was helpful to reset the state between test runs without redeploying contracts). The presence of an off-chain database does not compromise the integrity of the maintenance records because the source of truth for asset status and maintenance events remains the blockchain. The off-chain data is used for **enhancements**: cost calculations, sensor analytics, etc., which complement the on-chain records. This synergy is typical in enterprise blockchain solutions, where a blockchain is used alongside databases to get the best of both worlds (immutability vs. query efficiency).

One interesting feature we've implemented in our system is the integration of a **predictive maintenance module**. It is microservice that simulates IoT sensors monitoring the building's assets and automatically flags issues or records data on-chain. The predictive service operates as follows:

- **Sensor Simulation:** The service periodically generates sensor readings for certain assets. For example, suppose we have a temperature sensor on an HVAC unit (asset ID 5). The service will, every N minutes, produce a random/real temperature value and timestamp it. It then stores this reading in the SQLite database (SensorReadings table). Over time, this creates a historical log of conditions.

- **Fault Prediction Logic:** The service monitors these readings to detect anomalies. In our implementation, we use a simple rule: if **3 consecutive readings** exceed a certain high temperature threshold (e.g., 70°C), the service will interpret that as an impending failure. This is a rudimentary predictive model (essentially threshold-based anomaly detection). When this condition is met, the service triggers a **fault report on behalf of the system**.

  How can the service report a fault? There are a couple of approaches: **(a)** have the service directly call the *AssetManager.reportFault* function using a dedicated Ethereum account (like an "IoT agent" account with User role for that asset), or **(b)** have it notify the admin/off-chain and let an admin confirm it. We opted for the automated on-chain route for demonstration. The predictive service is configured with an Ethereum wallet (private key) that is recognized by the system (it could be pre-registered as a User or even given a special role if desired). When a predicted fault occurs, the service uses Ethers.js to send a transaction to *reportFault(assetId, "Auto-detected high temperature").* This marks the asset as Broken on-chain just as if a human user reported it.

  The event FaultReported will then propagate through the system and the front-end will catch it and update the UI, alerting technicians/admin that asset 5 has a fault. This is a powerful concept: the DApp not only accepts manual input, but can also leverage IoT data to proactively log issues on the blockchain, ensuring they are noticed and cannot be ignored. (In a real deployment, one might require the admin to approve auto-generated faults, to avoid false positives, but our implementation focuses on the concept.)

- **Data Hashing and On-Chain Storage:** To further secure the sensor data and provide an audit trail, the predictive maintenance service periodically takes a batch of recent sensor readings and computes a cryptographic **hash** (e.g., SHA-256) of that data. It then stores this hash on-chain via the *AssetManager.storePredictiveHash(assetId, dataHash)* function. The AssetManager maintains a mapping predictiveHashes[assetId] which is essentially an array of strings (hashes) for each asset.

  When storePredictiveHash is called, it appends the provided hash to the array and emits an event *PredictiveHashStored(assetId, hash).* This mechanism provides **data integrity** assurance for the sensor readings. Even though the raw readings are off-chain, by periodically anchoring a hash of them on-chain, we can later prove that the off-chain data wasn't tampered with (as long as we have the original data). This pattern is aligned with known best practices for blockchain and data integrity: only store the hash of large data on-chain to ensure integrity without incurring high storage costs.

In our case, the predictive service might, for example, take the last 10 readings for asset 5, concatenate them in a JSON string, compute SHA-256, and get a digest like 0xabc123.... It then calls storePredictiveHash(5, "0xabc123..."). That hash is now forever on the blockchain. If later someone questions the accuracy of the sensor data, we can produce the 10 readings and recompute the hash to show it matches the on-chain record, proving those readings were indeed observed at that time (and not altered retroactively). We scheduled the service to do this hashing maybe every hour or every day per asset, depending on data frequency. The predictiveHashes stored on-chain effectively act as an audit log or **blockchain-anchored** log of the sensor telemetry.

- **Implementation details of the predictive service:** This service is written in Node.js (so it can reuse code and config from the main project, e.g., using Ethers with the same contract ABIs). It runs in a loop or uses timers. For example, it may have a setInterval for each asset of interest. On each interval, generate a reading, save to DB. After saving, check the last 3 values for that asset (it can query the DB easily) – if the last 3 are all above threshold and the asset isn't already marked Broken (it can call AssetManager.getAssetStatus(assetId) via a provider to double-check), then trigger the reportFault transaction. Additionally, on every 10th reading or at a timed interval, compute the hash of the recent chunk and call storePredictiveHash. To avoid spamming the blockchain, one might batch multiple assets' data into one hash or adjust frequency. In our controlled setup, the volume is low, so we directly store each asset's hash periodically.

Predictive Maintenance		
Tool/Library	**Version**	**Role in Project**
Node.js scripts	18 LTS	Periodic sensor data simulation, threshold checks, auto-fault submission.
ethers.js	6.13.5	Used by Node scripts to call AssetManager (reportFault, storeHashes).
SQLite	3.x	Local database storing sensor values for each asset, hashed on-chain.

Table 4.6. Predictive maintenance components for automated fault detection.

The interplay between the predictive service and the on-chain contract is an illustration of how **IoT and blockchain integration** can work. Blockchain alone typically cannot pull data from sensors (known as the oracle problem, it relies on external services to push data). Here, our predictive module acts as an **oracle** that feeds the contract with sensor-based insights. By doing so, the blockchain ledger now contains **some** information about the conditions of the asset (in form of events and stored hashes) without the need to store every data point. This approach is congruent with literature suggestions to use blockchain for data provenance in maintenance and IoT scenarios, where hashed data on-chain ensures later verification [Stackexchange [2019]]. It demonstrates a next-generation maintenance approach: issues can be detected automatically and recorded in a tamper-proof way, enabling a shift from reactive maintenance to proactive maintenance.

Express API structure summary: To enumerate the key endpoints:

- *GET /gasCosts*: returns all gas reimbursement records (assetId, user, costWei) as JSON. Used by admin UI
- *DELETE /gasCosts?assetId=X&user=Y&costWei=Z*: deletes a specific gas cost record after reimbursement
- *POST /gasCosts*: (implied) to add a new record; the front-end would call this right after a fault report transaction is sent. The server code for this would parse the body JSON and insert a row in SQLite. (We ensure to include the unique combination of asset and user so duplicates are not created.)
- *GET /resetDB*: (development only) clears the database tables for a fresh start.
- *POST /sensorData*: (optional, if predictive service used HTTP) to add a new sensor reading. In our case, not needed because the service writes to DB directly.
- *GET /sensorData?assetId=X*: to retrieve recent sensor readings for an asset (could be used for visualization or debugging of predictive logic). These endpoints are protected or open depending on context; since this is a prototype, we didn't implement authentication for the API, but in a real deployment, you'd secure it or keep it internal.

In conclusion, the off-chain backend components provide necessary support to the DApp by handling data that either doesn't need global consensus (like gas usage logs) or would overwhelm a blockchain (high-frequency sensor data). They are kept relatively simple and their data can be cross-verified with on-chain references (hashes and events) to ensure trust.

The presence of the predictive maintenance service highlights the extensibility of the architecture, it is straightforward to plug in additional services (even AI/ML analytics) that observe the on-chain state and off-chain data and then act by sending transactions to the contracts. This demonstrates an **integrated cyber-physical system**: IoT sensors feed a blockchain-based maintenance log, and the blockchain in turn triggers real-world actions (maintenance work orders) all while being user-friendly via the BIM front-end.

4.6 Integration of Components

With the front-end, smart contracts, backend server, and predictive service described individually, it's important to explain how they interoperate as a cohesive system. The integration of components in this DApp follows a **hybrid architecture** where on-chain and off-chain parts communicate through well-defined interfaces (web3 calls, HTTP calls, and events). Here we outline typical interaction flows and how data moves through the system:

User-Initiated Maintenance Workflow: Consider a building occupant using the application to report a problem:

1. The user opens the React app in a browser and connects their MetaMask wallet. The front-end retrieves their blockchain address and role by calling the AssetManager's getRole function. Suppose the user is already registered as a User (Role=User).

2. The user navigates the 3D model (via IFCViewer) and double-clicks on, say, a light fixture that is flickering in real life. The IFCViewer captures the click, finds the object's IFC GlobalId, and calls handleObjectSelected(GlobalId) in the React app. The app looks up a mapping of GlobalId to asset ID (this mapping was built when the app loaded all assets via AssetManager calls on startup). It finds that GlobalId corresponds to asset ID 7, "Ceiling Light in Room 101".

3. The FloatingAssetPanel for asset 7 is shown, displaying its status (Operational) and details. The user enters "Light is flickering and about to fail" in the fault description field and clicks **Report Fault**. The front-end uses Ethers.js to send a transaction: *assetManagerContract.reportFault(7, "Light is flickering. . . ").* This transaction goes to the Polygon network. The MetaMask prompt shows the estimated gas (say 50,000 gas, costing 0.001 POL) to the user.

4. On-chain, the AssetManager checks permissions (the user's address is in userRoles as User) and updates asset 7's status to Broken, stores the fault string, and emits FaultReported(7, "Broken", "Light is flickering. . . ").

5. The user's MetaMask confirms the transaction. The front-end, after getting the tx receipt, sends an HTTP request to the Express server: POST /gasCosts with assetId:7, user:"0xUserAddress", costWei: <actual gas cost> (the actual gas cost is obtained by multiplying gasUsed * gasPrice from the receipt). The server logs this in SQLite.

6. Meanwhile, the front-end's WebSocketProvider (listening to AssetManager events) catches the FaultReported event emitted from Polygon. In the callback, it calls loadAssets() which fetches updated asset info from the contract (or it could simply update asset 7's status in state). The UI updates: asset 7 now appears as Broken (in red) in the asset list and on the model (we could even change the object's color in the Three.js view to indicate it's broken).

7. A maintenance technician looking at the app (maybe on their own login) will also see asset 7 marked Broken (since the event is broadcast and their front-end also refreshes, or they manually refresh asset list). The technician goes to Room 101 in real life to inspect the light.

8. The technician, using the app, selects asset 7 from the list or model and clicks **Start Maintenance**. This sends a tx startMaintenance(7, "Investigating light flicker"). On-chain, asset 7 status -> Under Maintenance, tech address recorded, MaintenanceStarted event emitted. All UIs update accordingly (admin and user UIs will see that asset 7 is now being serviced).

9. After fixing or replacing the light, the technician in the app clicks Complete Maintenance, adding a comment “Replaced ballast, issue resolved”. This calls completeMaintenance(7, "Replaced ballast"). On-chain, asset 7 status -> Operational, maintenance record closed, CompletedMaintenance entry added (readyForPayment=true), event MaintenanceCompleted emitted.

10. The admin’s interface picks up MaintenanceCompleted (through event or by polling pending payments) and now sees asset 7 in “Pending Payments” with technician = TechAddress. The Express server is queried for gasCosts: it returns that user 0xUserAddress spent e.g. 0.00087 POL on asset 7. The admin sees this info.

11. The admin enters a technician fee (say 0.02 POL) and clicks Pay on asset 7’s entry. The front-end calls confirmPayment(7, 20000000000000000 wei, 0xUserAddress, 870000000000000 wei) (passing the user address and gas cost in wei). On-chain, AssetManager calls PaymentManager: transferring 0.02 POL to TechAddress and 0.00087 POL to UserAddress. These transfers occur from the funds the admin had deposited earlier. The CompletedMaintenance entry is updated as paid.

12. The admin UI refreshes: asset 7 disappears from pending list and appears in paid history with details (who was paid how much, when). The front-end also issues a DELETE /gasCosts?assetId=7user=0xUserAddress... to remove the gas log now that it’s handled.

13. The cycle concludes with asset 7 fully operational and all parties updated. The blockchain now has an indelible record that asset 7 had a fault at a certain time, was repaired, and payment was made, all traceable via events and state. The off-chain DB has the raw sensor and gas info (should anyone need to audit energy usage or costs).

Predictive Maintenance Workflow: Now consider the predictive service’s operations: The predictive service is running continuously. Every 5 minutes, it logs a temperature reading for, say, asset 10 (an HVAC unit). These go into the DB as [time, value]. Over an hour, values might be normal (e.g., 60°C). But then it starts reading 85°C, 88°C, 90°C consecutively. Upon the third high reading, the logic triggers. The service uses its Ethereum key (it could be the admin key or a special oracle key pre-registered as a user) to send reportFault(10, "Overheat detected") automatically. This transaction is mined; asset 10 becomes Broken. The event FaultReported is emitted.

All clients (admin dashboard, etc.) see asset 10 turn Broken with fault "Overheat detected". A technician is dispatched proactively **before** a complete failure occurs. This could prevent downtime. The admin knows this came from the predictive system (by the fault description or by the fact no human submitted it).

The predictive service also computes a hash of the day’s readings for asset 10. Suppose the hash is H. It calls storePredictiveHash(10, H). On-chain, this is stored in the predictiveHashes mapping and event PredictiveHashStored is emitted. The admin interface could have a view to see these hashes (or it could ignore them; they are mainly for audit). Storing the hash costs a small amount of gas (to save a 32-byte string), but this is done maybe once per day.

Later, if someone wants to verify that the data which led to the fault was legitimate, they can retrieve all readings for that day from the database and hash them to see if they match the on-chain hash H. If they match, it proves those readings haven't been altered (ensuring trust in the predictive process). This approach addresses trust in automated decisions by **anchoring data provenance on the blockchain**.

Communication Patterns: The integration uses several communication channels:

1. **Web3 calls (transactions and queries):** Front-end to blockchain, Admin backend to blockchain, Predictive service to blockchain. The system interacts with smart contracts through user-initiated MetaMask/Ethers transactions and predictive module programmatic transactions. Users and the predictive module both access identical smart contracts which maintain state consistency.

2. **Web3 events:** Blockchain to front-end. The React app listens to events from AssetManager via WebSocket (Infura) and updates UI state. This decouples the front-end from constantly polling and provides real-time feedback to users. For example, when one user reports a fault, another user's interface updates almost immediately. It leverages the publish-subscribe nature of blockchain logs.

3. **HTTP REST API calls:** Front-end to Express server (for gas costs) and Predictive service to Express or DB (for sensor data). These are standard web requests on the local network (in our dev setup, the React app calls *localhost:4000* for the API). The server responds typically in JSON. This pathway is entirely off-chain and is used for data not stored in the contracts.

4. **Direct DB access:** Predictive service to SQLite. Running in the same environment, the predictive script can use an SQLite client library to insert/query data without going through HTTP. This improves performance for large data ingestion. The trade-off is it bypasses the Express API validations (not a big concern in our controlled scenario).

The components are **synchronized** via the unique IDs of assets and the consistent logic of the workflow. For instance, asset 7 in the database corresponds to asset 7 on-chain (we use the same ID as the primary key in both realms). When an event refers to assetId, the front-end can join that with any off-chain info by assetId. Similarly, the predictive service uses the same assetId when logging data and when calling on-chain functions. By designing a clear interface (asset IDs and perhaps GlobalIds when needed), we ensure all parts speak the same "language" regarding referencing assets.

Error Handling and Resilience: Integration-wise, we considered cases like the Express server being down or the predictive service failing. The system is designed such that these would not compromise the core functionality:

- If the Express server is down, users can still report faults and technicians can still do maintenance, only the gas reimbursement logging would be temporarily unavailable. Those could be cached client-side and sent later, or worst case the admin manually reimburses by checking transaction receipts on Polygonscan.

- If the predictive service is down, the system just loses the automated insights, but manual reporting still works. Predictive service can be restarted and it will continue logging data. Because it stores data in the DB, a restart doesn't lose past readings. And because it stores hashes on-chain occasionally, even a restart (which might produce a different hash slice) is fine as long as it eventually covers the data in a new hash.

- If the WebSocket event connection fails, the front-end will not get live updates, but the UI has manual refresh pathways (the admin can always click "Refresh Assets" which calls loadAssets() from the blockchain directly). In our implementation, we call loadAssets() after each important user action anyway as a fallback to ensure UI state is synced.

Polygon Network Integration: Using Polygon (or any EVM network) is abstracted away by Ethers.js configuration. The contract addresses (for Polygon deployment) are plugged into the React config, and Infura provides a WebSocket URL for Polygon. We also had to ensure MetaMask was pointed to the correct network (Polygon mainnet or testnet) when users connect, otherwise transactions would fail. The integration tested on Sepolia vs Polygon required only changing network RPC URLs and contract addresses, all code remained the same because the EVM interface is consistent. This proves the **portability** of the DApp across Ethereum-compatible chains.

In summary, the integration of components is orchestrated such that **blockchain is the single source of truth for asset states and critical events**, while the **Express server and predictive service enrich the system with additional data and automation**. The React front-end acts as the unifying interface that pulls from both sources: it **fetches blockchain state via ethers** and **calls Express APIs via HTTP**, presenting a seamless experience to the user. Figure 4.1 (in document) illustrates this interaction flow: users on the left interact with the React+IFC front end, which sends transactions to Polygon and fetches data from the smart contracts, while also making requests to the Express server for supplemental data; simultaneously, the predictive maintenance module feeds data into the system and triggers smart contract calls when conditions are met. This integration approach ensures that each technology is used for what it's best at: blockchain for **immutability and trust**, [dibs42 [2023]] databases for efficient storage and query, and a 3D front-end for **user-friendly visualization**

4.7 Deployment Process and Execution

Deploying and running the entire system requires setting up each component in the correct order and providing the necessary configuration (particularly the contract addresses to all parts that need them). This section outlines the step-by-step deployment and execution procedure, as documented for our prototype:

Smart Contract Deployment: We use Hardhat for deploying the AssetManager and PaymentManager contracts to the blockchain network. A deployment script (e.g., scripts/deploy.js) automates this. The script likely does the following:

- Uses Hardhat's runtime environment to get a signer (the deployer account, which in our case is also the admin).
- Deploys PaymentManager first. This returns the deployed address (say 0xPaymentAddr).
- Deploys AssetManager next. Since AssetManager's constructor doesn't require PaymentManager's address (it has none in constructor), we deploy it independently and get its address (0xAssetAddr).
- Calls assetManager.setPaymentManager(0xPaymentAddr) from the deployer account (admin) to link the PaymentManager into AssetManager
- Calls paymentManager.setAssetManager(0xAssetAddr) from the deployer (owner) to reciprocally link AssetManager into PaymentManager
- Optionally, we might preset the deployer's role as Admin in AssetManager. In our design, by default the deployer could be considered Admin (for example, we could initialize userRoles[deployer]=Admin in the constructor or just treat the deployer as an implicit admin since only they have the keys to call admin functions initially). If needed, a function could assign an Admin role to the deployer's address.
- The script outputs the two contract addresses and perhaps saves them to a config file or prints to console.

On migrating from **Sepolia (Ethereum testnet) to Polygon**, we had to update Hardhat's network configuration. We added a network entry for Polygon (or Polygon Amoy if using testnet) with the RPC URL (Infura or Alchemy endpoint) and chain ID. The private key of the deployer is provided (via environment variable) so Hardhat can sign deployments. We then ran npx hardhat run scripts/deploy.js –network polygon to deploy to Polygon. This yielded new addresses for AssetManager and PaymentManager on Polygon. These addresses were then inserted into the front-end configuration and any other place they are needed (e.g., predictive service).

Environment Variables: We used environment variables to avoid hardcoding sensitive or environment-specific info:

- PRIVATE_KEY: the deployer/admin's private key for deployment (Hardhat uses it).
- INFURA_API_KEY): for connecting to Polygon network.

- POLYGON_RPC_URL: the full RPC endpoint constructed using the above key. These are loaded in Hardhat config. Similarly, the front-end might have a .env for REACT_APP_INFURA_ID or the Infura WebSocket URL (though in our case we put the WSS in a config constant). The Express server might use env for its port or DB file path, etc.

2. Front-End Setup and Launch: After deploying the contracts, we update the React app's config.js with:

- ASSET_MANAGER_ADDRESS = "0xAssetAddr" (from deployment).
- ASSET_MANAGER_ABI = [...] (the contract ABI JSON, which we can generate from Hardhat artifacts).
- PAYMENT_MANAGER_ADDRESS = "0xPaymentAddr".
- PAYMENT_MANAGER_ABI = [...].
- INFURA_WSS = "wss://polygon.infura.io/ws/v3/_PROJECT_ID" (the WebSocket endpoint for events). These ensure the front-end talks to the correct contracts on the correct network. Next, we build/serve the React app. In development mode, running npm start will start the dev server on localhost (port 3000 by default). For production, we would build (npm run build) and deploy the static files to a web server. In our testing, we ran the dev server which opened the app in a browser. The front-end will immediately try to connect to MetaMask and the blockchain. It's important that the user's MetaMask is on the Polygon network (or the network we deployed to). If not, we prompt the user to switch network. Once connected, the front-end loads initial data by calling loadAssets() – this function uses Ethers to connect to the blockchain and fetch all assets (likely by iterating asset IDs up to nextAssetId). It will also fetch pending user/tech requests if the user is admin, etc., to populate the UI.

3. Backend Server Setup: We navigate to the /backend directory and install dependencies (if any, e.g., npm install express sqlite3). We ensure a SQLite database file is in place (the server code might create one if missing). We then start the Express server, e.g., node server.js or npm start depending on how it's set up. The server prints a message like "Express listening on port 4000". We verify it by calling GET /gasCosts which should return an empty list (since nothing logged yet). The Express server is relatively independent; it doesn't need to know contract addresses (except maybe if it also listened to events, which in our case it does not). It just needs to be accessible to the front-end (which it is, since both can be on localhost different ports, or they could be hosted together).

4. Predictive Maintenance Service: We launch the predictive script, e.g., node predictiveService.js. This script will likely output logs like "Simulated sensor reading X for asset Y" every few seconds/minutes. If it detects a threshold breach, it might log "High temp detected on asset Y, reporting fault..." and then "Transaction hash: 0xabc...". We ensure this service has access to:

- The blockchain (it might use the JSON RPC HTTP endpoint or WebSocket). We provide it with the RPC URL and perhaps use the deployer's key or a dedicated key for signing transactions.
- The contract addresses/ABIs (so it can call the functions).
- The SQLite DB (if it writes directly). If it uses HTTP to server, it needs server address. For simplicity, we might just import the Hardhat artifacts (which include ABIs) and use Ethers with the same INFURA API key. We also load a private key for it (which could be the admin's or another account that has User role). Ideally, we register that account in the system: e.g., if using a separate key, after contract deployment, we (as admin) call approveUser(predictionBotAddress) to authorize it as a User so it can call reportFault. This step would be done either in deployment or manually via Hardhat console.

5. Populate Initial Data (if needed): With everything running, the admin may want to register some assets initially to test. This can be done through the Admin UI: go to "Register New Asset", fill details (including a valid GlobalId from the IFC file), and submit. Alternatively, one could have a migration script that registers a bunch of assets via Hardhat after deployment. We did it manually for demonstration, registering a few sample assets corresponding to elements in the IFC model (ensuring the GlobalIds match those in the MyBuilding.ifc used by the viewer).

6. Running the System: At this point:

- The React front-end is served (http://localhost:3000), the user can open it and use the app.
- The Express server is running (http://localhost:4000) to handle logs.
- The predictive service is running in the background.
- The contracts are deployed on Polygon and contain whatever initial state (e.g., maybe admin already approved themselves and the predictive bot as users).
- MetaMask is configured for Polygon with the user's account funded with a bit of POL for transactions (e.g., a few tenths of a POL for testing). When a user performs actions, we watch the terminal running the Express server – it will show any incoming requests (for instance, after a fault report, we'd see a POST log). We also watch the Hardhat or Infura logs for contract events or the predictive service output. If using Infura, one can also monitor transactions on Polygonscan by the contract address to see events getting recorded.

Migrating from Sepolia to Polygon: Initially, we tested the system on the Ethereum Sepolia testnet. That involved deploying the contracts to Sepolia (which uses ETH test tokens) and configuring MetaMask to use Sepolia. Once things were stable, we transitioned to Polygon Amoy (the Polygon testnet) for further tests, and finally to Polygon mainnet for demonstration. The migration steps included:

- Deploying new instances of the contracts on the target network.
- Updating the front-end config with new addresses and possibly using the appropriate ABI (if unchanged, ABI is same).
- Switching Infura endpoints from Sepolia to Polygon.
- Distributing POL to test accounts (instead of ETH).
- Different currency: PaymentManager code was initially written with "ETH" in mind (naming), but on Polygon those are POL. This required no code change since POL is also the native currency (just semantic clarity that 0.01 ether in code means 0.01 POL on Polygon). We double-checked all constants like the reimbursement amount (0.01) to ensure it made sense value-wise on Polygon.
- Because Polygon has faster blocks and lower gas, the UX improved (transactions confirmed within $\tilde{5}$ seconds, and costs were pennies).
- We also took into account chain IDs and network differences in MetaMask: MetaMask may treat Sepolia and Polygon networks differently, so we had to add the Polygon network config to MetaMask (RPC URL, chain ID 137 or 80001 for Mumbai) in order for it to connect. The front-end can prompt this by an Ethereum API call to switch network if needed.

Verification and Final Checks: After deployment, we performed a (full integration test): registering a new user via the UI, approving them via admin UI (checked that PaymentManager did reimburse **0.01 POL**, verifying the PaymentManager balance decreased accordingly), that user reporting a fault, etc. We also tested the predictive flow by artificially triggering sensor conditions (or by lowering the threshold temporarily). We monitored the Polygon explorer to see events: for example, a **FaultReported** event carries the assetId and description in logs we could see those on the blockchain, proving the system's transparency.

For maintaining the system, we documented how to restart each component. The state mostly lives on-chain, so restarting the front-end or server doesn't lose data. The SQLite DB holds sensor data and gas logs; if it's deleted, the only loss is the gas log (which could be recovered manually by checking past transactions if absolutely needed). We set up the server such that it creates the DB file if not present, so redeploying backend is simple.

In a production scenario, one would likely containerize each component (Docker for the server and possibly the service, serve the React build on a CDN or static server) and use environment-specific config for dev vs prod (for example, pointing to testnet or mainnet). Our focus was to get a working prototype in a local environment and on a public test network.

4.8 Summary of Implementation

This chapter detailed the realization of a decentralized application for building maintenance management, integrating blockchain and BIM technologies. We described how a **React front-end with a 3D IFC model viewer** was developed to provide an intuitive interface, allowing users to click on building elements and report issues in-context. The front-end is role-aware, altering available actions for building occupants, technicians, and administrators, thereby enforcing usability constraints on top of the smart contract's role-based access control. We leveraged the **IFC (Industry Foundation Classes)** BIM model to bridge the physical world and the digital records, an occupant sees a faulty light in the 3D model and generates a blockchain transaction to log that fault, all in a few clicks. The use of the IFC model viewer (built on Three.js) demonstrates how **immersive, data-rich environments** can be combined with blockchain to enhance user experience and reduce ambiguity in maintenance processes.

On the blockchain side, we implemented two Solidity smart contracts: **AssetManager** and **PaymentManager**. The AssetManager contains the core logic of maintenance workflows: asset registration, fault reporting, maintenance tracking, and role management. By storing asset states and emitting events for each significant change (faults, maintenance start/completion, etc.), it creates an **immutable ledger of maintenance events** that stakeholders can trust and verify. The PaymentManager contract handles the financial transactions, enabling automated, rule-based disbursement of maintenance fees and gas reimbursements. This smart contract layer realizes the key promises of blockchain in our context: **transparent logs** (any broken equipment report or completed repair is recorded on-chain), **permissioned actions** (only authorized roles can do certain things, enforced by code rather than human procedures), and **automatic execution of payments**(once conditions are met, the contract logic ensures technicians and users are compensated fairly).

We also integrated an **off-chain Express server and SQLite database** which serve as a supportive subsystem for data that is either too voluminous or not security-critical enough to warrant on-chain storage. This includes tracking the gas costs that users expend (to facilitate an off-chain reimbursement mechanism that feeds into on-chain transactions) and storing high-frequency **sensor data** for predictive maintenance analysis. By off-loading this data, we keep on-chain operations efficient and cost-effective, a design choice consistent with best practices to use off-chain storage for large datasets while using blockchain for verification. The server acts as a mediator that the front-end can query for aggregated info (like cost reports), complementing the direct smart contract calls. This demonstrates a pragmatic approach to DApp architecture: using blockchain where it adds value (integrity, decentralization) and traditional databases where they are more practical (complex queries, large data).

A significant extension we built is the **predictive maintenance microservice**. This component underscores the flexibility and extensibility of our architecture. It simulates IoT sensors monitoring asset conditions and uses a simple algorithm to predict faults (e.g., detecting an overheating pattern). Upon detecting a likely issue, it autonomously interacts with the blockchain by invoking the same smart contract functions that a human user would (reporting a fault). Thanks to this, it creates a record on-chain of a potential failure was detected by the system, initiating the maintenance workflow without the need of the user. We also ensured the trustworthiness of this process by hashing sensor data periodically and storing it on-chain, thereby **timestamping and sealing** thedata in the blockchain. This approach merges IoT and blockchain, illustrating how **automated data-driven decisions** can be made transparent and auditable. The predictive service's integration shows that our DApp is not just a static application, but part of a larger cyber-physical ecosystem that can include AI/ML analytics, sensor networks, and more. In the context of facility management, this enables the system to detect upcoming maintenance requirements which it records in an auditable ledger to shift from reactive maintenance methods to proactive ones.

Overall, the implementation achieves the goals set out in the architecture (Chapter 3). We have a working system where:

- **Users, Technicians, and Admins interact through a unified platform** (web app) that ties together a BIM model and blockchain backend. The role-based interface and smart contract permissions together enforce the correct workflows and data visibility for each stakeholder
- **All critical maintenance events are recorded on the blockchain**, ensuring an immutable history. For example, if a dispute arises whether a repair was made on time or if a device consistently fails, the blockchain logs provide evidence. We met the requirement of transparency and tamper-resistance by using Polygon's public ledger for these events.
- **The BIM integration (via IFC)** provides context, making the data easily understandable (seeing which asset in which room is broken, rather than dealing with IDs in isolation). This addresses the usability and data integration aspect – bridging maintenance management systems with BIM models, which has been a challenge identified in prior research
- **Off-chain components optimize performance and cost**, handling things like gas cost tracking and sensor storage, which would be impractical on-chain. At the same time, we didn't compromise the trust model: for every off-chain piece of data that matters, we anchored it to the blockchain via hashes or events (for instance, the gas usage ultimately gets approved by admin on-chain, and sensor data gets hashed on-chain). This hybrid approach satisfies cost efficiency requirements without sacrificing too much decentralization
- **Payments to technicians and reimbursements to users are automated and trustless.** The admin just confirms the amounts, and the smart contract executes the transfer from a prefunded contract. This removes the need for separate accounting outside the system; all parties can verify payment events on-chain. It showcases how blockchain can streamline financial workflows in maintenance operations (no

need to wait for end-of-month billing cycles; payment can trigger immediately upon job completion and approval).

- **Deployment on Polygon** gave us real-world feasibility insight. The transactions in our tests cost on the order of $0.01 each and confirmed within seconds, which proves that the system could be used in a live setting without prohibitive costs or latency. Polygon's EVM-compatibility meant we could use the same code from Ethereum testnets, highlighting the portability of our solution across blockchain networks.

In conclusion, the DApp demonstrates **complete blockchain integration for building maintenance management while IoT data and BIM visualization** provide additional efficiency benefits. These technologies validate their combined ability to establish a better maintenance process that functions transparently and efficiently while providing users with friendly interfaces. The successful realization of this prototype sets the stage for further evaluation (Chapter 5) regarding its performance, security, and user acceptance, and it lays the groundwork for potential enhancements (such as more sophisticated predictive analytics, integration with facility management systems, or scaling to larger deployments). The work in this implementation chapter confirms that the architecture proposed is not only theoretically sound but also practically achievable with today's web3 tooling, and it serves as a reference blueprint for similar blockchain+BIM applications in the construction and facility management domain. The key accomplishments include establishing an immutable maintenance log, achieving real-time synchronization between a BIM model interface and blockchain state, and introducing automated maintenance triggers, all in a decentralized application that operates across multiple platforms (browser, blockchain, server) in concert. The infrastructure implementation achieves the core objective of this thesis through proof of decentralized technology success for maintenance management. The system demonstrates decentralized technology can effectively **improve maintenance management** by delivering greater data security and automated processing while integrating better than centralized systems.

Chapter 5

Testing and Validation

5.1 Blockchain Network Evaluation (Ethereum vs. Polygon)

We conducted a comparative evaluation of deploying the maintenance DApp's smart contracts on two blockchain testnet networks: Ethereum (Sepolia testnet) and Polygon (Amoy testnet). The goal was to empirically assess transaction **costs** and **performance** on each network and determine which would be more suitable for a real-world deployment of our system. Key metrics observed include gas fees for typical operations, transaction confirmation times, and overall user experience in terms of responsiveness and cost-effectiveness.

Empirical Comparison Summary: Table 5.1 summarizes the empirical data from our deployments on the two networks:

	Ethereum Sepolia			**Polygon Amoy**		
Function & Role	Gas Used	Cost (ETH)	t_{conf} [s]	Gas Used	Cost (POL)	t_{conf} [s]
Admin `registerAsset`	362 877	1.81×10^{-4}	14.2	362 877	3.6×10^{-2}	5.6
User `reportFault`	65 687	4.2×10^{-5}	13.4	65 687	6.6×10^{-3}	5.1
Tech. `startMaintenance`	141 492	8.9×10^{-5}	10.4	141 492	1.4×10^{-2}	5.2
Tech. `completeMaintenance`	114 628	7.4×10^{-5}	13.4	114 628	1.1×10^{-2}	5.2

Table 5.1. Gas usage, on-chain fee, and confirmation latency for core maintenance calls on Sepolia vs Amoy

5.1.1 Cost-Efficiency Comparison (ETH vs. POL per Function Call)

Ethereum Sepolia and Polygon Amoy differ greatly in cost per transaction due to both gas price and token value differences. On Sepolia, gas fees are paid in ETH, which is highly valuable, whereas on Amoy (Polygon's testnet), fees are in POL which has a much lower USD value. In our tests, an identical transaction cost orders of magnitude less on Amoy than on Sepolia, for example, let's take the function registerAsset:

- **Sepolia (Ethereum)**: For the same function execution, the fee was about **0.000181 ETH**. At current prices on 25 July 2025 ($2,106 per ETH [coinmarketcap [a]]), that's roughly **$0.38** in cost. Smaller function calls (e.g. tens of thousands of gas) would correspond to only fractions of this ETH amount, but still notably higher in USD than on Polygon.

- **Amoy (Polygon)**: The POL fee for the same transaction was about **0.036 POL**. With POL trading around **$0.1567 USD** [coinmarketcap [b]] on 25 July 2025, this comes out to only about **$0.0056**. Even when using a high priority gas price (30 Gwei on Amoy) the dollar-cost remains negligible due to POL's low price and Polygon's efficiency.

Polygon Amoy is far **more cost-efficient**. Even after accounting for gas usage being the same, each unit of gas on Polygon costs less in USD because POL is cheaper than ETH and Polygon's network fees are lower. Function calls that might cost a few cents or dollars worth of ETH on Ethereum cost mere fractions of a cent on Polygon. This cost advantage makes Polygon very attractive for dApps concerned with transaction fees. For example, deploying our test smart contract on Sepolia cost about **$0.38** vs **$0.005** on Amoy for the same gas used.

5.1.2 Gas Usage Analysis per Function

Gas consumption for each function call was virtually identical on Sepolia and Amoy. This is expected, since both networks run the Ethereum Virtual Machine (EVM) with the same gas cost schedule. The provided data shows that for every tested function, the **gas units used were the same on both chains** (differences, if any, were negligible). For example, the contract **deployment** itself consumed about **2,879,407 gas** on **Sepolia** and the same **2,879,407 gas** on **Amoy**. If a particular function used 50,000 gas on Sepolia, it would use ~50,000 on Amoy as well.

This parity occurs because gas usage depends on the computational steps and storage operations of the function, which do not change between Ethereum and Polygon. Our contract's bytecode was executed in the same way on both networks. The **gas usage per function** in the table5.1 can be analyzed without worrying about chain-specific quirks. F|or example, a state write or a hash computation costs the same gas on Sepolia as on Amoy. The identical 2.88M gas used in deploying the contract on both networks confirms this consistency. In summary, **Polygon doesn't reduce gas usage**, but it reduces what you pay for that gas (as seen above). Gas usage optimizations must come from the code itself; the EVM charges the same gas for the same logic on both chains.

5.1.3 Confirmation Latency Analysis

Transaction confirmation was significantly faster on Polygon Amoy. Ethereum Sepolia operates with a block time roughly in line with Ethereum mainnet with about **12 seconds**

per block under PoS. In contrast, Polygon's PoS chain (and thus Amoy testnet) has a much higher block frequency, yielding typical confirmation times on the order of **only 4-6 seconds**.

In our observations, a transaction sent to Sepolia generally got included in the next block within ∼10–15 seconds and achieved full confirmation within ∼1–2 minutes (after a handful of blocks). Sepolia's **block finalization** is around 12–24 seconds per block in normal conditions, meaning even one block (first confirmation) takes that long. On Polygon Amoy, blocks are produced far more quickly (multiple per 10 seconds), so transactions were often confirmed **within 2 seconds** or a few seconds at most. Our contract deployment on Amoy was mined in the first block (block position 0) and was confirmed almost instantly after submission, whereas on Sepolia the deployment had to wait for the next 12-second slot to be mined. This aligns with Polygon's design as a high-throughput chain: *"on average, confirmation time ranges from 2 to 5 seconds"* on Polygon ([maxelpay [2024]]), versus Ethereum's fixed ∼12-second blocks.

Faster confirmation on Polygon can greatly improve user experience for dApps (near-instant feedback), while on Ethereum one usually waits at least ∼12 seconds for a single block confirmation. However, it's worth noting that finality on Ethereum (the point at which a transaction cannot be reverted) is very robust with its PoS consensus, whereas Polygon PoS relies on checkpointing to Ethereum, but for typical app purposes, the **speed of Amoy's block confirmations is a clear advantage**.

5.1.4 Current ETH and POL Token Prices (Today)

As of today (May 20, 2025), the token prices are as follows:

- **Ethereum (ETH):** Approximately **$2,106.29 USD** per ETH [coinmarketcap [a]]. Ethereum's price is up about 5.4% in the last 24 hours. This high value per ETH is why even tiny fractions of ETH in gas fees can translate to significant cost (e.g. 0.001 ETH ∼$2.53).
- **Polygon (POL):** Approximately **$0.1567 USD** per POL token [coinmarketcap [b]]. POL is up ∼0.9% in the last 24 hours. With a price well under a dollar, POL makes gas fees on Polygon extremely cheap in absolute terms. For instance, 1 POL (worth ∼$0.23) can pay for many transactions on Amoy.

These up-to-date prices put the cost differences in perspective. **Paying 0.000181 ETH is about $0.38**, whereas **0.036 POL is only about $0.0056**, as noted. The huge gap between ETH's and POL's unit price underpins the cost-efficiency of Polygon for transaction fees.

The same smart contract deployment address is currently:

- **Ethereum Sepolia Contract:** 0x8D6427c2a8Ed5bCf83246E8cb59CD926742C6B40
- **Polygon Amoy Contract:** 0x02d6Ae3bFC1aecd57D732610b5A2d0e6fc252735

Both testnet contracts were deployed by the same account and have the same logic (and gas profile).

5.1.5 Conclusion

Ethereum Sepolia and Polygon Amoy both executed the same smart contract with **equal gas requirements**, but **Polygon Amoy achieved vastly cheaper and faster transactions**. Gas usage per function did not change between networks, underscoring that performance in terms of execution is equivalent. However, the cost to the user (in USD or fiat terms) was dramatically lower on Polygon because of lower gas prices and a lower-value token. Likewise, **transaction throughput and confirmation speed** on Polygon outpaced Sepolia, with users seeing confirmations in a few seconds on Amoy versus around 12+ seconds on Sepolia. These differences highlight the typical trade-offs between Ethereum mainnet (simulated by Sepolia testnet) and Polygon PoS sidechain (Amoy testnet): **Ethereum offers the base security and decentralization at higher cost and latency**, while **Polygon provides speed and cost-efficiency at the expense of being a separate chain anchored to Ethereum.**

5.2 Usability Evaluation of the DApp

Beyond technical performance, we evaluated the **usability** of the front-end decentralized application (DApp) to ensure that end-users can interact with the maintenance system effectively. This DApp provides a 3D BIM interface (IFC model viewer) and various forms for reporting issues and managing tasks. A **usability test** was designed with scenarios covering all user roles and their primary tasks. The aim was to identify any UI/UX problems and measure how easily typical users (occupants, technicians, administrators) can complete maintenance workflows using the system.

User Roles and Test Participants: We defined three key roles in the system, and for each role recruited representative test participants or personas:

- **Building Occupant (End-User):** Typically a resident or office worker who would use the DApp to report maintenance issues (faults) in the building. This user has basic permissions (can report a fault on an asset they select in the BIM model, view status updates on their reports, etc.). For testing, we assumed a participant with minimal technical knowledge to see if the interface is intuitive for laypersons.
- **Maintenance Technician:** A user responsible for addressing and fixing faults. In the DApp, technicians can view assigned work orders, update the status of a maintenance task (e.g., start or complete maintenance), and possibly add notes. We enlisted a participant with some technical background to perform technician tasks, ensuring the workflow (from seeing a reported fault to logging completion) is straightforward.
- **Administrator (Facility Manager):** This is the admin role with the highest privileges. Admins can register new assets, approve new users or technicians, assign

reported faults to technicians, and initiate payment reimbursements. The test participant for this role might be, for example, a facility manager or an IT manager familiar with maintenance processes. This role tests more complex interactions in the DApp (like navigating admin dashboards and managing user roles).

Test Tasks and Scenarios: We crafted realistic tasks for each role to perform during the usability test. Each task was designed to be representative of a core use-case of the DApp:

- *Occupant Task:* **Report a Fault on an Asset**, The occupant is instructed to use the 3D model to locate a specific building element (e.g., a HVAC unit on the third floor) and submit a fault report describing an issue with that element. This task evaluates the intuitive nature of the BIM interface (can the user find the item by clicking the model or searching?) and the simplicity of the fault reporting form. We measure if the occupant can complete the report without guidance, and how long it takes.

- *Administrator Task:* **Assign a Fault to a Technician**, After a fault is reported, the admin must review the new fault report and assign it to an available technician. The admin may need to navigate to an admin panel, view a list of pending fault tickets, and use an interface to choose a technician and assign the task. This scenario tests whether the admin UI clearly presents pending issues and provides an easy way to dispatch them. It also checks that role-based features (only admins see the assign option) work correctly. Time to complete the assignment and any confusion in the process are noted.

- *Technician Task:* **Complete a Maintenance Task**, The technician finds that an issue has been assigned to them (perhaps they receive a notification or see it in their task list). The task is to update the system when the maintenance work is done. This involves changing the asset's status to "Operational" via the DApp and possibly triggering the PaymentManager smart contract to record a maintenance completion (which might include a reimbursement or payment if applicable). We observe if the technician can easily locate their tasks, mark one as completed, and whether the interface provides appropriate feedback (like a confirmation that the status changed and an event was emitted). This scenario assesses how well the system supports the technician's workflow and whether all necessary information (asset location, fault details) is accessible for them in the UI.

Overall, the usability evaluation provided valuable feedback. It confirmed that our role-based front-end design is largely effective (each user could access the features they needed and no one could perform unauthorized actions), and it identified a few areas to refine (mostly in front-end clarity and guidance).

5.3 Smart Contract Security Testing

To ensure the robustness of the blockchain logic, we performed a static security analysis of the smart contracts using **Slither** in a local environment, an open-source Solidity analyzer. *Slither* is a security framework by Trail of Bits that automatically detects common vulnerabilities and code issues in smart contracts [Feist]. It is crucial to use such a tool in blockchains since it can detect flaws very fast, before launching, with very little human work required. By detecting bugs and issues ahead of time, this way of working helps ensure the contract won't have problems that require costly fixes in the future.

Using Slither on our AssetManager.sol and PaymentManager.sol contracts revealed several issues, which we addressed as follows:

- **Missing Event Emissions:** Slither flagged that some state-changing functions did not emit events. For example, operations like asset registration or owner updates were being performed without logging an event. This is a vulnerability for transparency, as off-chain systems would find it difficult to track critical changes without events [slither]. **Resolution:** We introduced appropriate Solidity **events** and emit calls for all important state changes (e.g., asset status updates, new asset or user registrations, payment disbursals). Emitting events ensures that every critical action (such as a change of ownership or a fault report) is recorded on-chain logs, facilitating off-chain monitoring and audits.

- **Dangerous Use of Timestamps:** The analysis warned about the use of block.timestamp in the contracts. Relying on timestamps can be unsafe because miners can manipulate the timestamp within a reasonable range [slither]. In our maintenance system, timestamps were used (for instance, to record when a fault was reported or to enforce time-based conditions). **Resolution:** We reviewed all timestamp usages. In contexts where exact timing was not security-critical (e.g., for logging events), using block.timestamp is acceptable. However, we added comments and minor checks to ensure that any time comparison has a tolerable window. In future designs, a safer approach could be using block numbers for critical time logic. The key point is that we acknowledged this warning and confirmed that no game-breaking reliance on timestamps (such as for randomness or strict deadlines) was present.

- **Costly Operations in Loops:** Slither identified loops in our contract functions that could become costly if the arrays grew large. For instance, iterating through a list of all assets or maintenance requests on-chain could consume a lot of gas and even lead to out-of-gas errors if not bounded [slither]. **Resolution:** We refactored functions to avoid unbounded loops. In cases where loops were necessary, we ensured that the list sizes are limited by design (for example, an admin function that iterates over a small set of pending reimbursements in PaymentManager). We also considered moving some logic off-chain if iteration over many items was needed. To solve this vulnerability we used mappings and splitting certain tasks, we eliminated any *costly-loop* Slither warnings. This optimization not only prevents potential denial-of-service via heavy loops but also improves overall gas efficiency.

- **Naming and Shadowing Issues:** The analysis discovered that some variable names were being used twice. For example, a local variable name in a function might have shadow a state variable or built-in name, which can lead to confusion or errors [slither]. Such shadowing can make the code harder to understand or even cause logic to malfunction if the wrong variable is referenced. **Resolution:** We corrected these by renaming variables and functions for clarity and uniqueness. Any function or variable that triggered a shadowing warning was reviewed and adjusted to follow Solidity best practices (e.g., prepend _ to function parameters that shadow state variables, or choose distinct names). This resolved Slither's naming-related warnings and improved code readability. Additionally, we enforced Solidity naming conventions (like capitalizing constant names, camelCase for functions) to align with community standards.

After these fixes, the smart contracts passed the Slither analysis with no high-severity issues. No reentrancy, arithmetic overflow, or access control bugs were detected by the tool, giving us confidence in the contracts' security. In summary, **static analysis** with Slither proved valuable: it highlighted subtle weaknesses (missing events, timestamp reliance, gas-inefficient code, and naming problems) that were not obvious at first glance, and we systematically resolved each issue. This process strengthened the smart contracts before they were deployed to any network, which is crucial for a decentralized maintenance system where vulnerabilities could compromise trust.

Chapter 6

Conclusion and Future Work

This final chapter provides a summary of the accomplishments of this research, discusses the limitations of the developed decentralized maintenance management system, and outlines directions for future work. The implemented prototype demonstrated how Building Information Modeling (BIM) and blockchain can be integrated to improve building maintenance processes. Even though the results are good for transparency and efficiency, the system is still considered a proof-of-concept. This means that it is important to study the current barriers and think ahead about future improvements to move from a prototype to a working solution.

6.1 Summary of Achievements

The project successfully met its primary objective of creating a decentralized application for building maintenance management that leverages both BIM and blockchain technologies. **On the blockchain side**, a set of smart contracts was developed to handle maintenance task delegation, verification of work completion, and record-keeping of tasks. These smart contracts enforce the workflow rules transparently: for instance, a maintenance task can be delegated to a technician as a blockchain transaction, and the task is only marked as completed (and potentially triggers payment) once the assigned technician submits proof of completion on-chain.

This on-chain verification and tracking mechanism ensures that maintenance actions are tamper-evident and accountable. It aligns with concepts in prior research where smart contracts in a decentralized autonomous organization (**DAO**) setup automatically release payments upon human confirmation of task completion, thereby increasing trust and accountability in maintenance workflows. By using blockchain as the backbone, the system creates an immutable audit trail of all maintenance activities, which can boost transparency among stakeholders (facility managers, contractors, etc.) and reduce disputes over whether and when a task was performed.

On the front-end side, the system features a web application built with React, integrated with a 3D BIM viewer capable of rendering Industry Foundation Classes (IFC) models. This user interface allows maintenance personnel to interact with a digital twin of the building. They can navigate the building model, click on building elements to retrieve maintenance history, and visualize active or past work orders in their exact spatial context. The inclusion of BIM is a key functional achievement: the building model provides an intuitive 3D visualization and a standardized representation of building information across the facility's lifecycle.

In recent years, researchers have emphasized that BIM can support facilities management teams by linking maintenance data to the physical context of assets and by providing visual analytics for maintenance tasks [Bouabdallaoui et al. [2021]]. Our implementation realizes this by connecting each maintenance task to an element in the BIM model. For example, if a lamp or a HVAC unit requires servicing, the user can locate that component in the 3D model, view its details and past records, and even attach new maintenance records to it. This seamless integration of the UI with BIM greatly improves situational awareness: technicians can better understand the location and environment of equipment before going on-site, and managers can more easily comprehend which parts of the building are experiencing frequent issues.

System performance was evaluated through comparative testing on two blockchain networks: Ethereum Sepolia (a proof-of-stake Ethereum test network) and Polygon Amoy (a Polygon PoS test network). This comparison highlighted notable differences in transaction cost, speed, and scalability. The Ethereum network (Sepolia) exhibited higher gas fees and longer transaction confirmation times, and that is similar to the Ethereum mainnet's well-known performance constraints. In contrast, the Polygon network processed transactions faster and with negligible fees. Quantitatively, Ethereum's base capacity is on the order of only tens of transactions per second (roughly 15–25 TPS on mainnet) and transactions often incur dollar-scale fees, whereas Polygon can theoretically handle thousands of TPS with gas costs amounting to fractions of a cent [Arthur [2024]].

Our experiments reflected this gap: operations that would be expensive on Ethereum (even though Sepolia testnet ETH has no real monetary cost, the gas usage is indicative of cost on mainnet) were way cheaper on Polygon. For example, a typical task delegation transaction on Sepolia might consume a similar amount of gas units as on Polygon, but due to Polygon's lower gas price and faster block times, the transaction would confirm in around 2 seconds with a trivial fee, whereas on Sepolia it might take 15 seconds and correspond to a much higher theoretical fee. This performance edge of Polygon demonstrates the viability of the system at scale: a maintenance platform needs to handle potentially hundreds of transactions (work orders, updates, confirmations) in a day for a large facility, and doing so on Ethereum mainnet would be cost-prohibitive and slow. Using a Layer-2 solution like Polygon, the system gets the required speed and cost savings it needs to be practical and overcome the scalability problem blockchain apps often face.

In implementing the prototype, special consideration was given to **user convenience and data management**. One notable feature is the handling of blockchain transaction fees (gas costs) for end-users. In a naive blockchain application, each maintenance staff or building manager using the system would need to pay gas fees for every transaction (e.g., creating a task or marking a task complete), which would be a significant barrier to adoption. To mitigate this, our design employs an off-chain gas fee reimbursement mechanism. In practice, this means the DApp can be configured such that a central entity (for example, the facility owner or a dedicated relayer service) ultimately bears the gas costs.

By handling gas fees off-chain (for the user reimbursement service), the system lowers the entry barrier for users who may not be familiar with cryptocurrency. Another technical achievement is the use of off-chain data storage combined with on-chain hashes to manage large maintenance-related data. BIM models and maintenance records can include sizeable files or numerous sensor readings, which are impractical to store directly on the blockchain due to cost and size limitations.

Following best practices, the implementation stores detailed data (such as the generated data for the simulation of IoT) in off-chain repositories (database) and only keeps a cryptographic hash of that data on the blockchain. Storing hashes on-chain preserves the ability to verify the integrity and timestamp of the data—any future retrieval of the file can be checked against the hash to ensure it has not been altered. Crucially, this design avoids bloating the blockchain with large files and keeps transactions lightweight, as recommended in blockchain architecture guidelines (large data should remain off-chain with only links or hashes on-chain).

This approach was applied, for instance, to the predictive maintenance metadata: when sensor data was collected for equipment condition monitoring, the raw data remained in an off-chain database, but a hash of that dataset was recorded in a smart contract transaction. This way, if an algorithm later flags a potential fault (using the sensor data), we have an immutable on-chain proof of exactly which data was used for that prediction, ensuring trust in the predictive maintenance process without overloading the blockchain. Collectively, these measures — gas fee abstraction and off-chain data handling, improve the system's practicality and set the stage for scaling up to real-world use.

6.2 Limitations

Despite its technical progress, the system in its current form has several limitations. First, it has only been evaluated in a **controlled, non-production environment**. The testing was performed on Ethereum and Polygon test networks and with simulated maintenance scenarios, but **no live deployment in an actual building** has taken place yet. This means that real-world factors such as unpredictable user behavior, long-term system robustness, and integration with actual facility management processes remain untested. The prototype demonstrates feasibility and functionality, but issues of scale and reliability might emerge in a real deployment.

For example, in a large campus with hundreds of users, the throughput of even Polygon might be challenged, or network connectivity issues could disrupt MetaMask usage on-site. Likewise, we have not observed the system's behavior over months or years of operation; smart contracts cannot be easily changed once deployed, so any minor bug could become a long-term issue in production. The absence of a field trial thus far is a significant gap — one that is common in academic prototypes in this domain. Many prior studies on blockchain-BIM integration stopped at the conceptual or prototype stage without reporting lessons from real deployment, and our work, at this stage, shares that limitation.

Without direct user exposure in a live setting, we cannot fully assess *user acceptance* or measure concrete benefits like reduction in maintenance resolution times. This gap between the controlled environment results and the complexities of a real facility deployment is something that needs to be addressed before claiming the system is ready for industry use.

Secondly, the **access control and security model** of the system is relatively simple and not yet suitable for a production environment. The smart contracts and application assume a moderate level of trust and tech-savvy behavior from users. There is no fine-grained role-based access control implemented beyond the basic distinction of different Ethereum addresses. In practice, maintenance management involves roles such as administrators, supervisors, technicians, and external contractors, each with specific permissions.

In the current prototype, if someone has the right contract address and a funded account, they could call functions that perhaps they shouldn't (unless explicitly restricted by the contract logic). For example, any user with a valid account could attempt to mark a task as completed; the system relies on the contract checking that the caller is indeed the one assigned to the task, but there is no broader organizational authentication beyond that. There is no integration with existing identity management or authentication systems (like an LDAP/Active Directory or even a simple username/password server).

This lack of complex access control means the system may be vulnerable to misuse if deployed as-is—for instance, nothing currently prevents a user from registering themselves as a "manager" if they somehow obtained the necessary keys or if the UI does not enforce role separation strictly. Moreover, while the blockchain provides integrity of records, it does not inherently secure the front-end: an attacker with knowledge of the contract ABI could bypass the application and interact directly with the smart contract.

If our contract functions are not carefully permissioned, such direct interaction could break the intended business logic. In summary, the prototype does not yet implement the rigorous security measures (like multi-signature approvals, hierarchical roles, or smart contract-based role management) that a real system would require. This is a conscious limitation left for future enhancement, as the focus was on proving the core concept.

Another limitation is that the DApp **depends on MetaMask and client-side logic** and this could cause problems with how easy it is to use and how dependable the system is. Users have to use MetaMask (or an Ethereum wallet like it) to verify and authorize their transactions. This dependency implies that every end-user must have a certain level of familiarity with blockchain tools: they need to install the extension, manage a private key, switch to the correct network (Sepolia, Polygon, etc.), and handle transaction confirmations.

For maintenance staff who may not have any experience with cryptocurrencies, this is a significant hurdle. In its current form, the system's user experience is geared toward tech-savvy users or developers. There is a risk that non-technical users would find it cumbersome to carry out what should be simple tasks (like closing a work order) due to the extra steps of dealing with a cryptographic wallet. Additionally, because much of the workflow logic (such as form input validation, certain business rule enforcement, and triggering of transactions) is implemented in the React front-end, the system's correctness partly relies on the front-end functioning properly and as intended. If the front-end application has a bug, or if a user decides to interact with the blockchain back-end directly (bypassing the UI), there could be inconsistencies.

For example, the front-end might prevent a user from creating a task without attaching a certain document by design, but the smart contract might not strictly enforce that rule; a direct call to the contract could create a task without the document, leading to data inconsistency. Relying on the client side for critical checks is a known weak point, as it assumes users will not or cannot circumvent the provided interface. While this is acceptable in a prototype, a production system would need more robust server-side or on-chain enforcement of all rules.

Finally, the project's **testing and quality assurance** processes are limited. Beyond basic functionality tests during development and the use of Slither (a static analysis tool) to scan the smart contract code for common security vulnerabilities, no extensive automated testing framework was applied. There is a lack of unit tests for smart contracts and integration tests for the end-to-end system. This means there might be edge cases or concurrency issues (e.g., two users trying to update the same task at the same time) that were not observed.

Additionally, no formal code audit was performed; in industry, it is common to have third-party security audits for smart contracts, but our contracts have only undergone informal review. The absence of rigorous testing increases the risk that undetected bugs or security flaws remain in the system. For instance, while Slither can catch certain issues (like reentrancy vulnerabilities or unused variables), it cannot guarantee the absence of logical errors. Without deploying the system widely, we have also not tested its *user acceptance.*

We do not know how intuitive the interface is for facility managers or technicians in practice, or whether the blockchain latency would frustrate users accustomed to instantaneous app responses. In summary, the prototype's reliability and user-friendliness have not been proven under diverse scenarios, which is a limitation that tempers the interpretation of the results.

6.3 Future Work

Considering the above limitations and the evolving nature of technology, there are several important directions for **future work** to extend and improve the system:

1. Real-world deployment and evaluation: The immediate next step is to deploy the system into use in real operations to check its effectiveness and usefulness. You could start by conducting an experiment in a set-up area managed by the facility department, where pilots are encouraged. By conducting a pilot deployment, one can collect data on how the system behaves with real users, real maintenance tasks, and live integration into facility operations.

Key questions to answer would include: Does the blockchain-based workflow noticeably improve (or hinder) the efficiency of maintenance processes? How do actual users (technicians, managers) adapt to using MetaMask and the BIM viewer in their daily routine? What is the realistic transaction volume, and can the chosen blockchain network handle it with low latency at all times? During such a pilot, it would be valuable to measure quantitative metrics (e.g., the time taken from issue report to issue resolution before and after the system's introduction, number of tasks closed per week, etc.) and qualitative feedback (user satisfaction, perceived ease of use). Performing this kind of field study will likely reveal unforeseen challenges and guide refinements.

For example, users might say thatswitching networks in MetaMask is confusing, or the BIM model may be too hard to navigate on a tablet during on-site use. Those discoveries are important for guiding future adjustments. Moreover, real deployment would let us check how well the system works when many people use it and whether its security is maintained. Learnings from the field will add to the prototype findings and are necessary before scaling up. It is worth noting that many academic projects on blockchain in construction have stopped short of real deployment, leading to a gap in understanding the practical viability. Sharing the results of the deployment would help prove how BIM and blockchain can be used together outside of research settings.

2. Enhancing BIM visualization with spatial analytics: The integration of a BIM viewer opens up numerous possibilities for advanced functionality, and future work can focus on adding features like indoor navigation and spatial queries within the BIM interface. Currently, users can see the location of maintenance tasks in 3D, but they must rely on their own knowledge or external guidance to physically reach that location in the building.

A valuable enhancement would be an **indoor navigation module** that uses the BIM data to guide users through the building. For instance, if a technician needs to service an electrical panel, the system could not only highlight the panel in the model but also provide step-by-step directions (potentially on a floor plan or even in augmented reality) to get there from the technician's current position. This requires integrating indoor positioning systems (like Bluetooth beacons or Wi-Fi triangulation) or at least providing maps that technicians can follow. Implementing spatial query capabilities is another promising improvement.

Users could query the model for assets by type, location, or condition. For example, a manager might ask the system to "find all fire dampers on the second floor that haven't been inspected in the last year" – the system would then highlight those components in the BIM view and list relevant tasks or records. This kind of query-driven interaction would leverage the rich data in BIM and the blockchain records together. In the future, our system could incorporate AR features: using a tablet or AR glasses, a technician could hold up the device and see navigation arrows or equipment information layered on their view of the building, using the BIM model as the source of truth for where things are.

While AR and indoor positioning bring additional complexity (in terms of technology and calibration), they represent a frontier for improving maintenance efficiency. Thus, a future iteration of this project should explore integrating such spatial technologies, making the maintenance platform not just a passive record system, but an active guide for technicians moving through real space.

3. Usability and user experience research: Future work should also rigorously address the usability of the system. Introducing novel technology like blockchain into maintenance operations will only succeed if the end-users—maintenance technicians, facility managers, etc.—find the system accessible and helpful. A structured user experience (UX) study should be conducted following standard evaluation frameworks.

One recommendation is to use the **NASA Task Load Index (NASA-TLX)**, which is a widely-used tool for assessing perceived workload across mental, physical, temporal, performance, effort, and frustration criteria [Bouabdallaoui et al. [2021]]. Maintenance personnel could be asked to perform a set of typical tasks using the new system (for example, receiving a work assignment, finding the location in the BIM viewer, and logging the completion on the blockchain) and then fill out a NASA-TLX questionnaire. This would quantify how demanding the system is to use compared to their current maintenance management process. If the scores indicate high frustration or effort, that would highlight areas for improvement in the interface or workflow. Another metric to consider is the System Usability Scale (SUS), which provides a quick measure of overall usability.

In addition to surveys, observational studies and interviews should be used. Watching how technicians interact with the application can uncover UI pain points (perhaps they struggle to rotate the 3D model or to understand the MetaMask prompts). Interviews can the preferences for the user to us, such as “I would like that the system could do X” or “For me it takes too long to do Y”. When human-centered design principles are followed and feedback is acted on, the system can be adjusted to work smoothly in people’s daily lives.

The final goal is to ensure that the technology augments the maintenance process without overwhelming the user. Blockchain and BIM in the background can be complex, but the front-end should strive to present a simple, almost invisible, experience of those technologies. Continued usability testing, especially after major updates, will be important to track improvements. Over time, such evaluations can be documented and perhaps contribute to industry guidelines on how to design decentralized applications for non-technical users in the construction/facility sector.

4. Predictive maintenance and AI integration: A long-term and impactful direction for future work is the incorporation of **machine learning (ML) and predictive analytics** into the maintenance management system. Right now, the system is largely reactive: it logs issues that humans report and tracks the completion of scheduled or on-demand tasks. However, with IoT sensors increasingly deployed in modern buildings (monitoring equipment vibration, temperature, air quality, etc.), there is an opportunity to predict issues before they escalate.

The concept of predictive maintenance is to analyze operational data from equipment to anticipate failures, rather than waiting for a breakdown or following a fixed schedule. Future versions of the system could integrate data streams from building sensors and use ML models to detect anomalies or degradation in performance.

For example, an ML model could continuously analyze the current draw and temperature of an HVAC motor; if it notices a pattern indicative of impending failure, it could automatically create a maintenance task on the blockchain for a technician to investigate that motor. By doing so, maintenance becomes proactive: addressing problems before occupants even realize them. Research in facility management has shown that such data-driven approaches can significantly reduce unplanned outages and maintenance costs. Implementing this would require a pipeline where sensor data is collected (probably off-chain due to volume), processed by ML algorithms, and then triggers smart contract interactions.

The blockchain can play a role by timestamping sensor data hashes (to ensure the authenticity of data feeding the algorithms) and by recording the issuance of predictive alerts in a tamper-proof way. One could even envision smart contracts that hold a simple logic: if a trusted oracle or off-chain service signals that "Component X is likely to fail in 5 days," the contract could automatically flag this in the system and perhaps notify the responsible personnel via the UI. It is also useful to use ML to help manage maintenance activities by forecasting the right time to work on an asset to avoid causing much trouble and expense. Since these components are important, it will likely take experts from data science, building engineering and blockchain oracles to implement them. It also raises new considerations: ensuring the ML model's recommendations are explainable and justified (especially if they will trigger automated actions), and maintaining occupant privacy if sensors capture environmental data.

Still, introducing a predictive maintenance module would make the system far more valuable, allowing it to act as a guide that helps avoid problems and choose the best ways to maintain equipment. This aligns with the broader industry trend of moving from purely corrective or preventive maintenance to a predictive paradigm, leveraging IoT and AI for smarter facilities management. Our blockchain-BIM framework, enhanced with such capabilities, could serve as a foundation for an autonomous maintenance ecosystem where data flows securely, and the building effectively helps to take care of itself through advanced analytics.

Concluding remarks: In short, this thesis has demonstrated that decentralized building maintenance management is possible, as it connects the detailed data from BIM to the security and automated processes of blockchain. The prototype achieved core functionalities—secure task delegation, transparent tracking, and an interactive 3D interface—that show clear potential for improving how maintenance operations are conducted. At the same time, the project pointed out some of the issues that come up as you move from a prototype to a final system such as getting it to work smoothly, improving its security and ensuring it matches current practices. The work discussed above shows the path forward to address these issues. By deploying the system in real settings, refined for users, improved with new tech features and linked to business systems, the solution can transform into a strong industry tool. Ultimately, this research contributes to the vision of smarter, more resilient building management. A decentralized BIM and blockchain-based maintenance platform, once fully realized, can ensure that information flows more freely and securely between building stakeholders, that maintenance actions are verified and optimized, and that the built environment is managed with greater foresight and transparency. As these systems improve and different experts collaborate, they could become essential for the next phase of facility management, resulting in safer, more efficient and more sustainable buildings.

Acknowledgements

Ringrazio mamma e papà che mi ha supportato in questo percorso lontano da casa.
A Davide e Andrea che mi hanno spinto a diventare più indipendente e mi stanno guidando tutt'ora in questo percorso, chi lo avrebbe mai detto che sarei riuscito a laurearmi in ingegneria informatica.

Ringrazio la possibilità di essere andato in Cina, una delle esperienze più importanti della mia vita, che mi ha dato la possibilità di esplorare una parte del mondo ancora non troppo conosciuta e di aver incontrato tantissimi amici di diverse nazionalità e etnie con cui ho passato un periodo bellissimo.

Ringrazio i miei amici di Discord, specialmente Candio, Ronde e Seba, con cui passo ogni sera a ridere.

Ai miei amici di Torino, i Caruggi, con cui ho passato serate indimenticabili.
Infine, a Twice, i miei due amici di lunga data con cui ho passato di tutto.

Bibliography

Nana Akua N Adu-Amankwa, Farzad Pour Rahimian, Nashwan Dawood, and Chansik Park. Digital twins and blockchain technologies for building lifecycle management. *Automation in Construction*, 155:105064, 2023.

Mohammed S Alnahari and Samuel T Ariaratnam. The application of blockchain technology to smart city infrastructure. *Smart Cities*, 5(3):979–993, 2022.

Andreas M Antonopoulos and Gavin Wood. *Mastering ethereum: building smart contracts and dapps*. O'reilly Media, 2018.

Vincent Arthur. Polygon transactions: Fees, speed, limits, Aug 2024. URL https://coinwirez.com/polygon-vs-ethereum/#:~:text=Based%20on%20recorded%20gas%20fees%2C,congestion%20and%20high%20transaction%20demand.

Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

Abdeljalil Beniiche. A study of blockchain oracles. *arXiv preprint arXiv:2004.07140*, 2020.

Mirko Bez, Giacomo Fornari, and Tullio Vardanega. The scalability challenge of ethereum: An initial quantitative analysis. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 167–176. IEEE, 2019.

bimcommunity. Top benefits of integrating bim into facility management systems, Apr 2025. URL https://www.bimcommunity.com/community/top-benefits-of-integrating-bim-into-facility-management-systems/#:~:text=One%20of%20the%20standout%20advantages,to%20inefficiencies%20and%20potential%20mistakes.

Chaudhary Deng Bjelic, Nailwal et al. Pol: One token for all polygon chains. *GitHub repository*, 25(2022):1–25, 2022.

Apichart Boonpheng, Waranon Kongsong, Nopagon Usahanunth, and Chaiwat Pooworakulchai. Bringing blockchain technology to construction engineering management. *International Journal of Engineering Research & Technology*, 9(01):171–177, 2020.

Yassine Bouabdallaoui, Zoubeir Lafhaj, Pascal Yim, Laure Ducoulombier, and Belkacem Bennadji. Predictive maintenance in building facilities: A machine learning-based approach. *Sensors*, 21(4), 2021. ISSN 1424-8220. doi: 10.3390/s21041044. URL https://www.mdpi.com/1424-8220/21/4/1044.

Vitalik Buterin. What would a rollup-centric ethereum roadmap look like? URL https://ethereum-magicians.org/t/a-rollup-centric-ethereum-roadmap/4698.

Vitalik Buterin et al. Ethereum white paper. *GitHub repository*, 1(22-23):5–7, 2013.

coinmarketcap. price eth 20 may 2025, a. URL https://coinmarketcap.com/currencies/ethereum/.

coinmarketcap. price pol 20 may 2025, b. URL https://coinmarketcap.com/currencies/polygon-ecosystem-token/.

Crypto.com. Ethereum vs polygon: Scaling, collaboration, and the future of blockchain, Feb 2025. URL https://crypto.com/en/university/ethereum-vs-polygon-scaling-collaboration-and-the-future-of-blockchain.

Moumita Das, Xingyu Tao, Yuhan Liu, and Jack C.P. Cheng. A blockchain-based integrated document management framework for construction applications. *Automation in Construction*, 133:104001, 2022a. ISSN 0926-5805. doi: https://doi.org/10.1016/j.autcon.2021.104001. URL https://www.sciencedirect.com/science/article/pii/S0926580521004520.

Moumita Das, Xingyu Tao, Yuhan Liu, and Jack CP Cheng. A blockchain-based integrated document management framework for construction applications. *Automation in Construction*, 133:104001, 2022b.

dibs42. Building on blockchain: The next phase of bim technology, Apr 2023. URL https://www.dibs42.com/blog/building-on-blockchain-the-next-phase-of-bim-technology#:~:text=The20potential20of20blockchain20in,decisions20regarding20maintenance20and20upgrades.

Faris Elghaish, M Reza Hosseini, Sandra Matarneh, Saeed Talebi, Song Wu, Igor Martek, Mani Poshdar, and Nariman Ghodrati. Blockchain and the 'internet of things' for the construction industry: research trends and opportunities. *Automation in construction*, 132:103942, 2021.

Express. Node.js web application framework. URL https://expressjs.com/#:~:text=Express%20is%20a%20fast%2C%20unopinionated%2C,for%20web%20and%20mobile%20application.

Josselin Feist. Slither – a solidity static analysis framework. URL https://blog.trailofbits.com/2018/10/19/slither-a-solidity-static-analysis-framework/#:~:text=Next%20steps.

hardhat. hardhat. URL https://hardhat.org/.

Huakun Huang, Xiangbin Zeng, Lingjun Zhao, Chen Qiu, Huijun Wu, and Lisheng Fan. Fusion of building information modeling and blockchain for metaverse: a survey. *IEEE Open Journal of the Computer Society*, 3:195–207, 2022.

IBM. What is a cmms? URL https://www.ibm.com/think/topics/what-is-a-cmms.

Infraspeak. Maintenance statistics and trends 2025, Nov 2024. URL https://blog.infraspeak.com/maintenance-statistics-trends-challenges/#:~:text=%2A%2093,CXP%20Group%2C%202018.

Infraspeak. How to face the dark side of facilities management, Jan 2025. URL https://blog.infraspeak.com/dark-side-facilities-management/.

intellis. What are the benefits of blockchain for facilities managers? URL https://www.intellis.io/blog/what-are-the-benefits-of-blockchain-for-facilities-managers#:~:text=single%20source%20of%20shared%20truth%2C,record%20system%20for%20all%20transactions.

Kyeongbaek Kim, Gayeoun Lee, and Sangbum Kim. A study on the application of blockchain technology in the construction industry. *KSCE journal of civil engineering*, 24(9):2561–2571, 2020.

Jennifer Li and Mohamad Kassem. Applications of distributed ledger technology (dlt) and blockchain-enabled smart contracts in construction. *Automation in construction*, 132:103955, 2021.

Izabella V Lokshina, Michal Greguš, and Wade L Thomas. Application of integrated building information modeling, iot and blockchain technologies in system design of a smart building. *Procedia computer science*, 160:497–502, 2019.

Weisheng Lu, Xiao Li, Fan Xue, Rui Zhao, Liupengfei Wu, and Anthony G.O. Yeh. Exploring smart construction objects as blockchain oracles in construction supply chain management. *Automation in Construction*, 129:103816, 2021. ISSN 0926-5805. doi: https://doi.org/10.1016/j.autcon.2021.103816. URL https://www.sciencedirect.com/science/article/pii/S0926580521002673.

Dena Mahmudnia, Mehrdad Arashpour, and Rebecca Yang. Blockchain in construction management: Applications, advantages and limitations. *Automation in construction*, 140:104379, 2022.

maxelpay. Polygon transactions: Fees, speed, limits, Dec 2024. URL https://www.maxelpay.com/blog/polygon-transactions.

Mark Mergenschroer and Justin Lipsey. Building operations: Exploring facility data specifications to address the 95 URL https://www.autodesk.com/autodesk-university/class/Building-Operations-Exploring-Facility-Data-Specifications-to-Address-the-95-of-Wasted-Con

Satoshi Nakamoto and A Bitcoin. A peer-to-peer electronic cash system. *Bitcoin.–URL: https://bitcoin. org/bitcoin. pdf*, 4(2):15, 2008.

Tom Oulton. The missing link: How blockchain can help construction industry leaders to monitor and manage change, Dec 2023. URL https://www.rlb.com/americas/insight/perspective-2023-vol-2/the-missing-link-how-blockchain-can-help-construction-industry-leaders-to-monitor-an #:~:text=Tracking%20building%20safety%20data%3A%20The,the%20safety%20of%20their%20buildings.

React. React. URL https://github.com/facebook/react.

Ran Ren and Jiansong Zhang. Comparison of bim interoperability applications at different structural analysis stages. *Construction Research Congress 2020*, pages 537–545, 2020.

Casey Mullen Sean Olcott. Digital twin consortium defines digital twin. URL https://www.digitaltwinconsortium.org/2020/12/digital-twin-consortium-defines-digital-twin/.

slither. Node.js web application framework. URL https://github.com/crytic/slither/wiki/Detector-Documentation.

SourceForge. Best javascript libraries, 2025. URL https://sourceforge.net/software/javascript-libraries/#:~:text=Best20JavaScript20Libraries20,The20current.

Ethereum Stackexchange, 2019. URL https://ethereum.stackexchange.com/questions/65488/saving-the-actual-data-vs-hash-of-the-data-which-one-is-better/65489.

Nick Szabo. Formalizing and securing relationships on public networks. *First monday*, 1997.

ThatOpen. Thatopen. URL https://github.com/ThatOpen/engine_components.

ThatOpen. Thatopen/engine$_c omponents, 2025. URL.$

Viet Hoang Tran, Bernard Lenssens, Ayham Kassab, Alexis Laks, Etienne Rivière, Guillaume Rosinosky, and Ramin Sadre. Machine-as-a-service: Blockchain-based management and maintenance of industrial appliances. *Engineering Reports*, 5(7):e12567, 2023.

Žiga Turk and Robert Klinc. Potentials of blockchain technology for construction management. *Procedia engineering*, 196:638–645, 2017.

Chibuzor Udokwu, Alexander Norta, and Christoph Wenna. Designing a collaborative construction-project platform on blockchain technology for transparency, traceability, and information symmetry. In *2021 2nd Asia service sciences and software engineering conference*, pages 1–9, 2021.

Jinlong Wang, Xu Wang, Yumin Shen, Xiaoyun Xiong, Wenhu Zheng, Peng Li, and Xiaoxue Fang. Building operation and maintenance scheme based on sharding blockchain. *Heliyon*, 9(2), 2023.

Jun Wang, Peng Wu, Xiangyu Wang, and Wenchi Shou. The outlook of blockchain technology for construction engineering management. 2017.

Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

Dylan Yaga, Peter Mell, Nik Roby, and Karen Scarfone. *Blockchain Technology Overview*, Oct 2018. 10.6028/nist.ir.8202.

Dylan Yaga, Peter Mell, Nik Roby, and Karen Scarfone. Blockchain technology overview. *arXiv preprint arXiv:1906.11078*, 2019.

Botao Zhong, Xing Pan, Lieyun Ding, Qiang Chen, and Xiaowei Hu. Blockchain-driven integration technology for the aec industry. *Automation in Construction*, 150:104791, 2023.