

POLITECNICO DI TORINO

Master Degree course in Mathematics Engineering

Master Degree Thesis

Smart Contracts for Transparent Contractualist Decision Making in Human Centered Robotic Systems

Supervisors
Giovanni Dalmasso
Valentina Gatteschi

Candidate
Giulia Bodrito

Academic Year 2025–2026

Abstract

This thesis originates from the research context of the Valawai project, which investigates value-aware artificial intelligence and human-centered autonomous systems. Building on this broader framework, the thesis investigates how blockchain technology, smart contracts, and oracles can support traceability and auditability around candidate actions proposed by a human-centered robotic agent inspired by contractualism. Contractualism is used as the ethical motivation for the scenario, according to which an action should be justifiable to the people affected by it. The thesis does not implement contractualist moral reasoning on-chain and does not include a deployed physical robot. Instead, it focuses on the infrastructure that follows a proposed cleaning task in a student-residence environment.

The proposed system uses smart contracts to encode selected validation rules and record their effects. Users are represented through blockchain addresses, preferences are stored and checked on-chain, task priority is computed through an explicit scoring model, and final task validation depends on user status, preferences, robot resources, priority threshold, and contextual information. External weather data are integrated through a Chainlink CRE workflow, allowing one off-chain contextual variable to be delivered to the smart contract and used in the on-chain priority computation.

The case study shows that selected downstream checks can be made inspectable through contract state, function calls, access restrictions, and emitted events. In particular, the prototype makes it possible to inspect which inputs were considered, which validation conditions were checked, which weather factor was stored, and which state changes followed a successful validation. These results support a technical claim about transparency, traceability, and auditability. They should not be interpreted as a fairness evaluation, a user-understanding study, a validation of the off-chain agent's moral reasoning, or evidence that a physical robot completed a cleaning task.

Contents

List of Tables 6

List of Figures 7

1 Introduction 9

2 Technical Background and Definitions 13
- 2.1 Blockchain and Distributed Ledger Technologies . . . 13
- 2.2 Consensus . . . 14
- 2.3 Ethereum . . . 14
- 2.4 Gas in Ethereum . . . 14
- 2.5 Smart Contracts . . . 15
- 2.6 Oracles, the Oracle Problem, and Chainlink CRE . . . 15
- 2.7 AI and Smart Contracts . . . 16

3 State of the Art and Research Gap 17
- 3.1 Contractualist AI and Justifiable Decisions . . . 17
- 3.2 Transparency, Provenance, and Accountable AI . . . 18
- 3.3 Smart Contracts for Transparency and Automation . . . 19
- 3.4 Blockchain, AI, and Robotics . . . 19
- 3.5 Oracles and External Data in Smart Contract Systems . . . 20
- 3.6 Trust, Governance, and Privacy Boundaries . . . 20
- 3.7 Research Gap . . . 22

4 Proposed Model and Architecture 23
- 4.1 Introduction to the Model . . . 23
- 4.2 System Boundary and Architecture Layers . . . 23
- 4.3 Candidate Task Interface . . . 26
- 4.4 Constraint Categories . . . 26
- 4.5 Task Lifecycle and Pending-State Management . . . 27
- 4.6 Priority Model . . . 29
- 4.7 Priority Threshold in the Current Prototype . . . 29
- 4.8 Oracle and External Context . . . 30
- 4.9 Paymaster-Inspired Accessibility Feature . . . 30
- 4.10 Transparency and Verifiability . . . 31

5 Smart Contract Implementation 33
5.1 Development Environment . . . 33
5.1.1 Remix VM . . . 34
5.1.2 Ganache . . . 34
5.1.3 Sepolia . . . 34
5.2 Smart Contract Architecture . . . 36
5.3 User Contract . . . 36
5.3.1 Owner and Administrator Logic . . . 38
5.3.2 User Registration . . . 38
5.3.3 Role Update and Deactivation . . . 39
5.3.4 Reading User Information . . . 39
5.4 Preference Contract . . . 39
5.4.1 Connection with User Contract . . . 39
5.4.2 Preference Structure . . . 40
5.4.3 Submitting Preferences . . . 40
5.4.4 Checking Time Preferences . . . 41
5.4.5 Access Helper . . . 42
5.4.6 Preferences as Feedback for Future Adaptation . . . 42
5.5 TaskPriority Contract . . . 42
5.5.1 Purpose of the Priority Contract . . . 43
5.5.2 Access Control . . . 43
5.5.3 Weight Update . . . 45
5.5.4 Health Score . . . 45
5.5.5 Time and Urgency . . . 46
5.5.6 Dirtiness Score . . . 47
5.5.7 Contextual Factor . . . 50
5.5.8 Weather Factor . . . 52
5.5.9 Priority Calculation . . . 52
5.5.10 Oracle Functions . . . 54
5.5.11 Events . . . 55
5.6 TaskValidation . . . 55
5.6.1 Student and Robot Resources . . . 56
5.6.2 Administrative Control . . . 56
5.6.3 Student, Robot, and Preference Checks . . . 57
5.6.4 Distance, Execution Time, and Operational Cost . . . 57
5.6.5 Paymaster Access Logic . . . 58
5.6.6 Grouped Input Parameters . . . 59
5.6.7 Validation Pipeline . . . 60
5.6.8 Events . . . 60
5.6.9 Role in the Architecture . . . 61
5.7 Summary of the Smart Contract flow . . . 61

6 Oracle Integration with Chainlink CRE 63
6.1 The Oracle Problem . . . 64
6.2 Why Chainlink CRE . . . 65
6.3 Open Meteo as External Data Source . . . 66
6.4 Weather Factor Model . . . 68
6.5 CRE Workflow Architecture . . . 69
6.6 The TypeScript Workflow . . . 70
6.7 The Smart Contract Receiver . . . 72
6.8 Trust Chain and Data Handling . . . 73
6.9 Security and Trust Considerations . . . 74
6.10 CRE Access and Wallet Setup . . . 74
6.11 Sepolia Validation . . . 75

7 Case Study 77
7.1 Experimental Boundary and Account Roles . . . 77
7.2 Ranking and Selection Boundary . . . 78
7.3 Evaluation Method . . . 78
7.4 Meaning of Task Completion . . . 80
7.5 Worked Validation Trace . . . 80
7.6 Summary . . . 81

8 Conclusions and Future Work 83
8.1 Implemented Capabilities . . . 83
8.2 Evidence From the Case Study . . . 83
8.3 Limitations . . . 84
8.4 Future Work . . . 85
8.5 Final Remarks . . . 86

Bibliography 89

List of Tables

3.1 Comparison of related approaches and their role in the proposed thesis. . 21

4.1 Boundary between implemented on-chain logic, off-chain support, and non-implemented target features. 25

5.1 Approximation error of the rational dirtiness function for $k = 0.03$. 50
5.2 Input and state semantics for the priority calculation. 54

6.1 Open Meteo weather codes based on WMO weather interpretation codes. 67
6.2 CRE execution mode and observed experimental configuration. 76

7.1 Observed behaviour of the prototype validation scenarios. 79

List of Figures

1.1 Logical structure of the thesis. 12

4.1 System architecture of the proposed blockchain task-validation model . . 24
4.2 Target lifecycle of a candidate cleaning task 28
4.3 Oracle-based flow for external contextual information. 30

5.1 Remix interface 33
5.2 Alternative methods for obtaining Sepolia test ETH. 35
5.3 Incremental testing path from simulation to public testnet deployment. . 35
5.4 Remix VM 35
5.5 Sepolia Testnet 35
5.6 Dependency structure of the main smart contracts. 37
5.7 Factors used by `TaskPriority` to compute the priority score. 44

6.1 Oracle problem in the weather-factor integration. 64
6.2 Comparison between the Python prototype and the Chainlink CRE integration. 66
6.3 Architecture of the CRE weather update, separated into off-chain and on-chain components. 70
6.4 Configuration file used by the Chainlink CRE workflow 70
6.5 Sequence executed by the TypeScript CRE workflow. 71
6.6 Conceptual aggregation of weather values in the deployed CRE model. The number of nodes is illustrative and does not represent the local simulation mode used in the experiment. 72
6.7 Internal receiver flow used by `TaskPriority` to process a CRE weather report. 73
6.8 Validated Sepolia path from Open Meteo data to task validation. 76

7.1 Validation visualization through Streamlit dashboard 81
7.2 Priority queue visualization through Streamlit dashboard 82
7.3 Preferences visualization through Streamlit dashboard 82

Chapter 1

Introduction

Artificial intelligence systems are increasingly used in contexts where their decisions can affect human preferences, autonomy, and well-being, and, for this reason, the design of intelligent agents cannot only focus on efficiency or technical performance, but must also consider how decisions are justified to the people involved. This thesis starts from this perspective and investigates how blockchain technology, smart contracts, and oracles can support a transparent and verifiable infrastructure around candidate actions.

The reference scenario is inspired by Sweepy, a robotic cleaning agent introduced in the work of Dalmasso, Marcos-Vidal, and Pretus [13] designed to decide when it is appropriate to clean a room by considering both the current situation and the preferences of the users affected by its actions. In this thesis, Sweepy is used as a motivating decision making framework, not as a system that must be fully reproduced in Solidity: the focus is instead on the infrastructure that can surround a Sweepy-inspired agent once a candidate action has already been proposed.

The application domain considered in the prototype is a student residence, where the term *user* mainly refers to a registered student who lives in the residence and can be affected by the cleaning service. The current proof of concept assumes a simplified setting with one cleaning robot, or equivalently one robot assigned to a floor or area, and does not assume the existence of a deployed physical robot. The focus is on the decision process around a proposed cleaning action, not on the robotic functions used to perceive the environment, such as detecting dirt or navigating inside the building and these complex settings, such as multiple interacting robots or conflicts among several affected users, are left as future extensions.

Before presenting the technical contribution of this thesis, it is useful to introduce the ethical framework that motivates the scenario: contractualism. In Scanlon's formulation, contractualism is an ethical theory concerned with principles that no one could reasonably reject as a basis for mutual justification [4,32]. In this perspective, the focus is not simply on maximizing an overall outcome, but on whether an action can be justified to each affected individual and on the reasons that those individuals may have for rejecting it.

This idea is relevant for AI systems because an autonomous agent may act in situations where different users have different needs, preferences, and vulnerabilities. In such cases, a purely efficiency decision may not be sufficient, because the system should also

provide inspectable evidence about which rules, preferences, contextual data, and validation steps supported the acceptance or rejection of a proposed action. For this reason, contractualism provides a useful perspective for thinking about human-centered robotic decisions.

If on one hand, Sweepy provides a computational interpretation of contractual reasoning in a robotic cleaning scenario, on the other hand this thesis develops a blockchain infrastructure around that kind of scenario. Therefore, the thesis does not claim to implement contractualism itself on-chain, nor to reproduce the complete Sweepy learning mechanism, rather, it investigates a complementary engineering problem: how candidate actions can be validated, prioritized, executed, and recorded once they have been proposed contractualist-inspired agent.

The problem addressed by this thesis is that, even when an artificial agent proposes an action according to a human-centered ethical model, the subsequent steps may remain difficult to inspect. It may be unclear which user was affected, which preferences or constraints were checked, which contextual information was considered, why a task received a certain priority, and whether the final execution was recorded in a reliable way. This remains open in the Sweepy-inspired setting because the original model focuses on the agent's decision-making behavior, while the surrounding validation, provenance, and task-management infrastructure is not the main object of that work.

The goal of this thesis is therefore not to prove that contractualism is ethically superior to other moral theories, since that discussion belongs mainly to moral philosophy. Instead, the objective is technical: to propose and implement an architecture that can support a contractualist-inspired robotic system. The central research question can be formulated as follows: if a human-centered agent proposes an action, how can the subsequent validation, prioritization, and execution process be made transparent, traceable, and verifiable through blockchain technology?

In this work, blockchain is not presented as the component that performs moral reasoning, but rather, is used as an infrastructure around the agent. Smart contracts can encode explicit validation rules, manage users and preferences, compute task priority, check robot resources, and record the final outcome of the validation process. Moreover oracles can connect the system to external data, such as environmental or weather conditions, allowing the priority model to consider information that is not natively available on-chain.

This distinction is important because blockchain does not automatically make a decision fair, correct, or ethically justified, however, it can support auditability, make rules and state changes inspectable, and provide a tamper-resistant execution record. In this way, users or system administrators can verify which rules were applied, which data were considered, and why a task was accepted or rejected.

Hence, this thesis proposes a broader blockchain task-management architecture and implements a bounded proof of concept. The target architecture includes candidate admission, pending management, dynamic revalidation, priority ranking, task selection, execution, and audit recording. The implemented proof of concept focuses on validating an already selected task through Solidity smart contracts, priority computation, explicit

checks, emitted events, and Chainlink CRE external-context integration. The work explores how smart contracts can represent registered students, preferences, robot resources, priority rules, paymaster access logic, and external contextual data. In particular, the implemented prototype evaluates the feasibility of this architecture through Solidity smart contracts and an oracle workflow for weather information. The final objective is to provide a concrete technical foundation for a system in which robotic actions are not only selected or proposed by an intelligent agent, but also validated, recorded, and made auditable through a blockchain layer.

The thesis first introduces the technical background and the main definitions needed to understand the work in Chapter 2. Chapter 3 then reviews the state of the art and identifies the research gap addressed by the thesis. Building on these foundations, Chapter 4 presents the proposed model and clarifies the role of the blockchain layer in the overall architecture. The implementation is then described in Chapter 5, while Chapter 6 focuses on the integration of Chainlink CRE and Open Meteo as the external-context mechanism. Chapter 7 presents the prototype validation and discusses the evidence observed in the case study. Finally, Chapter 8 reflects on the limitations of the work and outlines possible future developments.

This organization is summarized in Figure 1.1.

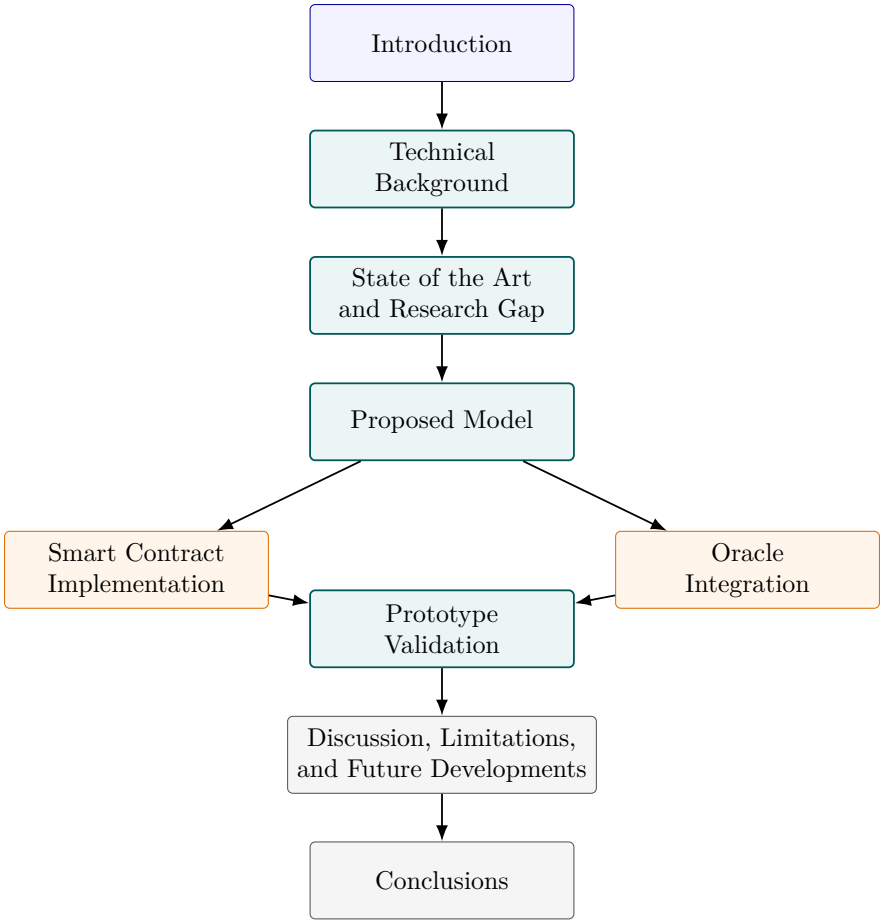


Figure 1.1: Logical structure of the thesis.

Chapter 2

Technical Background and Definitions

This chapter introduces the technical concepts needed in the rest of the thesis, it does not aim to provide a complete treatment of blockchain technology; rather, it defines the notions that are directly used in the proposed architecture: blockchain and distributed ledgers, consensus, Ethereum, smart contracts, gas, oracles, and Chainlink CRE. The purpose is to explain how these technologies can support the validation and traceability of a decision process inspired by the contractualist reasoning.

2.1 Blockchain and Distributed Ledger Technologies

A blockchain, according to ISO 22739:2024, is a distributed ledger composed of confirmed blocks organized in a sequential chain using cryptographic links [21]. In practical terms, data are not stored in a single central database, but are replicated across a network of nodes.

A ledger is a record of transactions or data entries, while a distributed ledger is shared and synchronized across multiple nodes, usually through a consensus mechanism. Blockchain is a particular type of distributed ledger in which information is grouped into blocks, and each block contains a cryptographic reference to the previous one, creating a chain structure where modifying an earlier block would also change its cryptographic link, making the alteration detectable.

For this thesis, the most relevant aspect is not the complete design of a blockchain protocol, but the fact that recorded operations can be inspected by different participants. A blockchain system can make the state of the system visible, make later manipulation difficult, and allow operations to be verified without relying only on an informal record controlled by one actor. In a user-centered robotic system, this is useful because the validation of a task can leave evidence of which rules were applied and which data were considered.

2.2 Consensus

Consensus is the mechanism through which the nodes of a distributed network agree on a common state and is needed because allows the protocol to determine which transactions are valid and which blocks can be added to the ledger.

Different systems use different consensus mechanisms, such as proof-of-work, proof-of-stake, or Byzantine Fault Tolerant based protocols. The latter is designed to address the problem of dishonest, faulty or offline nodes by allowing the network to reach a reliable result even when a limited number of participants behave incorrectly. The important point for this thesis is not to compare these mechanisms in detail, but to identify their general role: consensus allows a distributed system to maintain a common and verifiable state even when the participants do not rely on a single central database.

2.3 Ethereum

Ethereum is a blockchain platform designed not only to transfer value, but also to execute programs called smart contracts [6]. These programs are deployed on-chain, where they can store data, enforce rules, and update the state of the blockchain when users send transactions.

In this thesis, Ethereum is relevant because the implementation is based on Solidity smart contracts and on an Ethereum environment. During development, contracts can be tested in local or simulated environments, such as Remix VM, and later deployed to public test networks such as Sepolia. Test networks reproduce the main characteristics of Ethereum while using test ETH, the digital coin, which makes them suitable for experimentation before any real deployment.

2.4 Gas in Ethereum

In Ethereum, gas is the unit used to measure the computational effort required to execute operations on-chain. Every transaction consumes gas, paid by users in ETH, according to the complexity of the operations it performs.

Three concepts are sufficient for the following chapters: the gas used is the amount of computation actually consumed by a transaction. The gas price is the amount of ETH paid per unit of gas, while the gas limit is the maximum amount of gas that the user is willing to spend. These mechanisms prevent infinite or excessively expensive computations and make execution costs explicit.

In this thesis, the term gas must be used carefully, since Ethereum gas refers to the cost of executing transactions on-chain, while the project also uses an abstract notion of robot operational cost, sometimes represented in the code as a required resource for task execution. This operational cost is related to robot battery, task duration, and sensor level, and should not be confused with Ethereum gas.

2.5 Smart Contracts

As introduced briefly previously, a smart contract is a program whose code is deployed on a blockchain and executed by the network, can define rules, store variables, manage permissions, and automatically update its state when specific functions are called; in Ethereum, smart contracts are commonly written in Solidity and deployed through transactions.

Smart contracts are useful because they make rules explicit and executable, in particular, in the context of this thesis, this means that elements such as user registration, roles, preferences, priority thresholds, robot resources, and task validation can be represented directly in code. The decision process is therefore not hidden inside an informal procedure: the validation logic is encoded in functions that can be inspected, executed, and verified.

It is important to clarify that smart contracts should be deterministic, that is, given the same state and the same input, all nodes must obtain the same result. For this reason, smart contracts cannot directly perform arbitrary web requests or read data from external APIs, and when a contract needs information from the external world, this information must be provided through an oracle mechanism.

2.6 Oracles, the Oracle Problem, and Chainlink CRE

An oracle can be defined as an external component that observes data or events outside the blockchain and reports them to smart contracts, addressing the limitation of smart contract by providing external data [26]. In this way, smart contracts can react to real information while preserving on-chain execution and verification. This is very important in the proposed system, since the priority of a cleaning task may depend on contextual information, such as weather conditions, that is not naturally available inside Ethereum.

Oracles introduce an important challenge known as the oracle problem: even if the blockchain itself is decentralized and tamper-resistant, the external data source may still be centralized, wrong, unavailable, or manipulated, and if a smart contract depends on incorrect external data, the on-chain decision may also be incorrect.

This creates a trade-off between efficiency and decentralization, in fact a single oracle source may be simple and fast, but it introduces a point of trust. A decentralized oracle network can reduce this risk by using multiple nodes and aggregation mechanisms, although it may require more infrastructure and cost.

This issue is particularly relevant for this thesis because task priority can depend on external contextual data and, for this reason, the oracle component should not be described as a minor implementation detail but as a part of the trust model of the system.

In this thesis, the decentralized oracle network Chainlink is used to connect smart contracts with external data and off-chain computation [5], in particular the Chainlink CRE layer allows the system to connects the off-chain source with the on-chain smart contract 6.

2.7 AI and Smart Contracts

AI systems and smart contracts can be combined in different ways; on the one hand AI can support prediction, classification, recommendation, or adaptation, while smart contracts can provide rule enforcement, traceability, access control, and auditable execution. In this thesis, the smart contract is not used to implement an AI model or the full reasoning process of the agent. The blockchain component instead provides an infrastructure around the decision process: it records relevant data, enforces predefined rules, and makes the outcome verifiable. This is especially important in a contractualist-inspired setting, where a decision should be justifiable to the affected user.

The main limitation of this approach is the performance, since on-chain execution is slower and more expensive than ordinary off-chain computation, and blockchain systems may be difficult to scale for high-frequency interactions. For this reason, the proposed architecture is most suitable for situations where transparency, accountability, and verifiability are more important than raw speed.

Chapter 3

State of the Art and Research Gap

If the previous chapters introduced the motivation of the thesis and the technical background needed to understand the proposed system, this chapter moves from those foundations to the related literature and to the research gap addressed by the thesis. Its objective is not to give a general survey of all AI ethics or blockchain applications, but to focus on the areas that are directly connected to the thesis: justifiable artificial agents, transparent and accountable autonomous systems, smart contracts for auditable execution, blockchain robotic or cyber-physical systems, and oracle-based access to external information.

The chapter is organized around two research directions, the first concerns contractualist AI and the problem of making automated decisions justifiable to the users affected by them. The second concerns blockchain and smart contracts as technical tools for transparency, automation, provenance, and verifiable execution. The final section brings these directions together and formulates the research gap that motivates the proposed architecture.

3.1 Contractualist AI and Justifiable Decisions

Contractualism has been proposed as a way to think about artificial agents whose actions are not evaluated only by global utility or by fixed rules, but also by whether they can be justified to the people affected by them. This is particularly relevant in human-centered environments, where an automated system may need to balance efficiency, preferences, autonomy, and contextual constraints.

Human-centered AI research emphasizes that intelligent systems should not be designed only to maximize autonomy or technical performance. Shneiderman argues that reliable, safe, and trustworthy AI systems should combine high levels of automation with meaningful human control [33]. This perspective is relevant for the thesis because a cleaning robot in a shared residence affects users' preferences, autonomy, and well-being. Therefore, the problem is not only whether the robot can act efficiently, but also whether the process around its actions can remain understandable, inspectable, and accountable.

A complementary perspective is offered by Rahwan's society-in-the-loop framework. Rahwan argues that algorithmic systems with social impact should be governed through mechanisms that represent stakeholder values and monitor compliance with an algorithmic social contract [31]. This idea is useful for the student-residence scenario, where the robot's actions may affect different users and where rules, permissions, preferences, and governance choices must be explicit. However, the framework remains primarily conceptual and does not define a blockchain validation infrastructure. In this thesis, blockchain is therefore not treated as a necessary consequence of society-in-the-loop, but as the engineering approach investigated to make selected rules, permissions, validation steps, and recorded outcomes inspectable.

Dalmasso, Marcos-Vidal, and Pretus propose a contractualist model of moral decision-making embodied in a robotic cleaning agent called Sweepy [13]. In their model, the robot evaluates whether cleaning is appropriate by considering predefined norms, the current context, and user feedback, showing how contractualist reasoning can be interpreted computationally in a robotic scenario.

At the same time, the literature also shows that contractualism remains difficult to operationalize in AI systems. Recent reviews of AI ethics describe contractualist frameworks as promising, but less commonly implemented than other ethical approaches, partly because notions such as reasonable rejection, justification, and fairness are difficult to express as technical requirements [19]. This limitation is important for the thesis since, if contractualist reasoning remains only inside the agent, it may be difficult for an external observer to inspect the downstream validation path: which recorded rules were applied, which preferences were checked, which contextual data were used, and the reason why an outcome was finally accepted or rejected.

Systems-level approaches to AI ethics help address this issue, for example, Paglioni and Mortenson argue that ethical risks in AI should be treated as design problems distributed across the whole system, rather than as isolated properties of a model [29]. This perspective is useful for the present thesis because it suggests that a human-centered agent should be supported by the surrounding architecture, including access control, data sources, rule enforcement, logging, and verification. In this sense, the thesis does not try to replace Sweepy's decision-making model, but studies how the process around an action can be made more explicit and auditable.

The following frameworks should not be treated as interchangeable: human-centered AI provides the broader motivation for designing systems around affected users; society-in-the-loop emphasizes governance, monitoring, and stakeholder oversight; and Sweepy is the specific contractualist-inspired robotic model used as the motivating scenario. Blockchain is not derived as a necessary conclusion of these frameworks, but is the engineering approach investigated in this thesis to make the subsequent validation process more inspectable.

3.2 Transparency, Provenance, and Accountable AI

The need for transparency is also visible in broader AI research, existing works identify provenance, explainability, and accountability as recurring challenges in AI systems. For

this thesis, the most relevant point is not the internal explainability of a complex model, but the possibility of reconstructing the process around an automated decision: the input used, the rule applied, the changed state, and who was allowed to perform that change.

Recent works combine explainable machine learning with blockchain mechanisms to improve transparency, auditability, and trust in AI systems [25, 30]. These studies are relevant because they show that blockchain can support the verification of inputs, outputs, and decision traces, in particular work on generative-AI provenance also shows that smart contracts can be used to track sources and attribution [23]; however, this direction is only peripheral for the present thesis, since the focus is not on ownership of generated content, but on task validation in a robotic decision process.

Recent work on transparency and understandability in blockchain smart contracts shows that public blockchains can support visibility and traceability at lower system levels, but do not automatically provide higher-level accountability or user understanding [3]. This distinction is important for the present thesis, since if a model is opaque, storing its output on-chain does not explain its internal reasoning. The proposed smart contract layer is therefore complementary to explainable AI: it can make selected validation rules, inputs, state changes, and events auditable, but it does not by itself explain the reasoning of the off-chain agent or guarantee that governance choices are normatively justified.

3.3 Smart Contracts for Transparency and Automation

Smart contracts have been studied as building blocks for digital transformation because they can automate agreements and reduce dependence on intermediaries [22]. They can also improve transparency by making rules and transactions visible to participants, from an organizational perspective, previous work on electronic business contracts and smart contracts highlights both advantages and adoption barriers, including trust, legal uncertainty, and usability [36].

These properties are relevant for a contractualist-inspired architecture, since, in order to have a justifiable decision, the rules used to validate that decision should be explicit and inspectable. Smart contracts are useful because they can encode rules in a form that is executable and verifiable and, in the context of this thesis, they can represent user roles, preferences, robot resources, priority thresholds, and task validation rules. Literature on authentication and authorization in smart contracts is also relevant because the prototype requires a distinction between registered students and administrative actors [27].

3.4 Blockchain, AI, and Robotics

Several works investigate the combination of blockchain, AI, and robotic systems: Garrocho et al. propose an approach that combines smart contracts and edge AI in a multirobot system [17]. Their work is relevant because it shows that blockchain can support coordination, data integrity, and traceability in robotic environments.

Recent high-level work on blockchain technology for mobile multi-robot systems argues that blockchain and smart contracts can support transparency, security, trust, supervision, synchronized data storage, and system-wide rules in robotic environments [14, 18].

This perspective is useful for the present thesis because it shows that blockchain is increasingly discussed as an infrastructure for robotic systems where multiple agents interact, and not only as a financial technology. However, this literature mainly focuses on multi-robot coordination, supervision, fault detection, and auditing, whereas the present thesis addresses a narrower human-centered scenario: the validation and priority assessment of candidate cleaning tasks after they have been proposed by a contractualist-inspired agent.

3.5 Oracles and External Data in Smart Contract Systems

As previously discussed, 2.6, one limitation of smart contracts is that they cannot directly access external information, leading to an important lack in real-world systems, where decisions may depend on environmental or contextual data. The use of oracles in IoT and supply-chain scenarios demonstrates how blockchain systems can be connected to external events while preserving a verifiable execution layer [26].

In this thesis, the oracle component is essential because task priority can depend on data that come from outside the blockchain, even if is important to keep in mind that the oracle does not make external data automatically correct [2, 20] but it only defines a more structured and verifiable way to introduce data into the on-chain process.

3.6 Trust, Governance, and Privacy Boundaries

The prototype should not be described as fully trustless, even if blockchain reduces the need to rely on informal records, it does not eliminate institutional trust. In the proposed architecture, the administrator remains a trusted governance and a configuration actor that can register users, assign roles, and modify selected parameters. The value of blockchain is that these authorized operations and their effects and changes can be made explicit, recorded and inspected but the blockchain cannot determine whether the value is substantively fair.

A conventional database with auditable logs could also record task validations, user roles, and configuration changes, and it would usually be cheaper and easier to modify. The additional property sought in this thesis is independent inspection of the validation state and emitted events without relying exclusively on one application operator or on a private log controlled by the same institution. This is useful for administrators, auditors, or affected users who need to verify which rules and parameters were applied. At the same time, this benefit comes with costs: on-chain execution is less flexible, public data must be minimized for privacy, and the system still depends on trusted administrators, correct contract design, and reliable external data sources.

Another important issues is privacy: the current prototype uses preferences, health-related conditions, and other user attributes as synthetic or simulated data. This is acceptable for a proof of concept, but it would not be appropriate to directly store identifiable health information or sensitive personal data on a public blockchain in a real deployment. Indeed a production system would require an off-chain or privacy-preserving design, where only the minimum necessary proof, reference, or state is recorded on-chain.

Approach	**What it makes inspectable**	**Decision stage addressed**	**Remaining trust assumptions**
Contractualist AI and Sweepy	The reasons, norms, context, and feedback used by the agent to evaluate an action.	Candidate action selection.	The internal reasoning model and learning process remain off-chain and must be trusted or separately explained.
Explainable AI and provenance	Inputs, outputs, explanations, or provenance traces associated with an AI result.	Model interpretation and traceability.	Recording an output does not by itself prove the correctness of the internal reasoning.
Smart contracts	Rules, permissions, state changes, events, and validation outcomes.	Downstream validation after a candidate action is proposed.	The administrator, contract design, and input data sources remain part of the trust model.
Blockchain robotics	Coordination, incentives, data integrity, or cooperation among robotic agents.	Robotic coordination and execution support.	Often focuses on multi-agent coordination rather than user-centred task justification.
Oracles	The path through which external data enter the smart contract system.	External-context update before priority computation.	The original external source may still be wrong, delayed, or unavailable.

Table 3.1: Comparison of related approaches and their role in the proposed thesis.

3.7 Research Gap

The literature reviewed in this chapter shows that contractualism has been explored as an ethical framework for AI decision-making, especially in the Sweepy model [13]. It also shows that blockchain and smart contracts have been studied as tools for transparency, auditability, provenance, automation, external-data integration, and robotic coordination [17, 18, 23, 30].

These works provide important foundations, but they leave room for a more specific investigation: contractualist-inspired AI research mainly focuses on how an agent reasons about actions, feedback, and justification, while Blockchain-based robotics and AI often focus on coordination, incentives, data integrity, security, or automation. What remains less developed is the role of smart contract infrastructure around a human-centered or contractualist-inspired robotic decision process, after a particular action has already been proposed.

This thesis addresses this space by studying how smart contracts can support transparent validation, provenance, and task management in such a setting. The objective is not to claim that blockchain makes decisions ethically correct, nor that it replaces the contractualist reasoning of the agent. Rather, the thesis investigates how a blockchain layer can make the subsequent process more inspectable: who is allowed to interact with the system, which preferences are considered, which external data are used, which priority is assigned, and why a task is accepted or rejected.

The proposed architecture therefore does not replace the moral reasoning of the agent, but it studies the infrastructure around a candidate decision: the rules, inputs, permissions, oracle updates, priority score, validation checks, and records that can be inspected after the candidate action has been proposed. Therefore, the prototype uses smart contracts and oracles to make validation rules, state changes, and execution records explicit and auditable, providing a concrete technical basis for connecting a contractualist-inspired decision making scenario with a verifiable execution and task management layer.

Chapter 4

Proposed Model and Architecture

4.1 Introduction to the Model

This chapter moves from the foundations to the proposed model defining the engineering structure of the system: what the blockchain layer receives, checks, stores, ranks, and returns.

The reference scenario is a student residence with one primary cleaning robot, or one robot assigned to a floor or area, even though the prototype does not include a deployed robot, since the focus is on the decision process around a proposed cleaning action: how a candidate task enters the system, how it is checked against explicit rules, how it is ranked among other pending tasks, and finally how the final execution can be recorded.

In this thesis, as analyzed in chapter 1 a cleaning action should not be executed only because it is technically possible or globally efficient, but it should also pass explicit checks that make the decision more transparent for the person affected by it. For this reason, the model is based on a clear separation between the off-chain agent that proposes actions, the blockchain layer that validates and manages tasks, the oracle layer that introduces external context, and the robot layer that would likely execute the selected task.

4.2 System Boundary and Architecture Layers

The system is divided into four layers: the first layer is the *off-chain decision or agent layer* and represents a Sweepy-like or contractualist-inspired artificial agent, which observes the situation and proposes candidate cleaning actions. The detailed reasoning of the agent, including the learning mechanism or the evaluation of moral reasons, remains off-chain and is not reproduced in Solidity.

The second layer, and main focus of the thesis, is the *blockchain validation and task-management layer*, which receives selected candidate tasks, checks explicit constraints, stores relevant validation state, computes priority values, and records validation outcomes. A complete on-chain pending queue is part of the target architecture, but it is not implemented in the current prototype.

The third layer is the *oracle layer*, which introduces external contextual information

into the on-chain system, allowing the smart contract to interact with external information. In the prototype, the external context is represented by weather data obtained through Chainlink CRE and Open Meteo, but the same pattern could be used in other applications for sensor readings, environmental measurements, institutional state, or other external information.

The fourth layer is the *robot and execution layer* and would likely execute the selected and revalidated task. In the implemented proof of concept this layer is represented only abstractly through resource checks and validation records; physical cleaning is not independently confirmed but the idea is that the robot does not define the validation rules by itself, instead it receives the outcome of the task-management process and performs the corresponding physical action. After execution, the system can update the blockchain state so that the action remains traceable.

The resulting architecture is summarized in Figure 4.1, which shows the main system boundary and the interaction between the off-chain agent, the blockchain validation layer, the oracle layer, the users, and the robot execution layer.

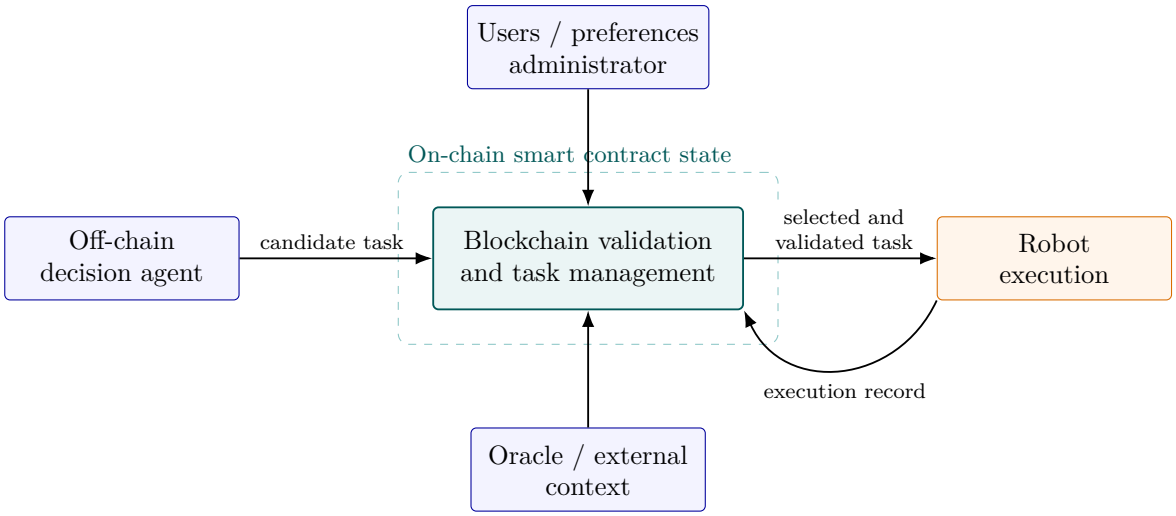


Figure 4.1: System architecture of the proposed blockchain task-validation model

The dashed boundary identifies only the on-chain smart contract state, while users, the administrator, the oracle workflow, the off-chain decision agent, and the execution layer remain external actors or services that interact with this state.

In particular registered students provide preferences and can be affected by tasks, while the administrator acts as a governance and configuration actor. The blockchain does not remove this governance role, but it can make authorized operations and state changes explicit and auditable.

Table 4.1 summarizes which parts of this architecture are implemented on-chain, which are handled off-chain, and which remain target features.

Element	Status	Clarification
User registration and roles	On-chain	Managed by `UserContract`. The contract stores role, reputation, activity status, and task counter.
Student time preferences	On-chain	Managed by `PreferenceContract`. `TaskValidation` checks whether the current blockchain hour is allowed.
Priority score	On-chain	Computed by `TaskPriority` from task data, health flags, weights, and stored `weatherFactor`.
Weather update	Off-chain + on-chain	The CRE workflow reads Open Meteo data off-chain. `TaskPriority` stores the resulting `weatherFactor`.
Task validation	On-chain	`TaskValidation` checks student validity, preferences, priority threshold, robot resources, and sponsorship/balance rules.
Priority ranking among tasks	Off-chain/dashboard	Multiple task scores can be computed and sorted outside the contract. The contracts do not maintain a pending queue.
Candidate task registry	Not implemented	The contracts do not store persistent task objects, task identifiers, or provenance references. Task data are passed directly to `validateTask`.
Physical cleaning completion	Not implemented	`taskCompleted` records successful validation or simulated completion, not independent evidence of real cleaning.

Table 4.1: Boundary between implemented on-chain logic, off-chain support, and non-implemented target features.

4.3 Candidate Task Interface

A candidate task contains the information needed by the current prototype to validate a proposed cleaning action for one affected user. In the implemented Solidity interface, the task is not stored as a persistent on-chain object, instead, the administrator submits the selected task to the validation contract through grouped input parameters.

Conceptually, the task submitted to the current prototype can be represented as:

$$T_i = (affectedUser, taskData, priorityData, healthData)$$

The field *affectedUser* identifies the primary user, in this prototyoe is a registered student, associated with the task. The array *taskData* contains the operational data used by `TaskValidation` to estimate the task duration, namely the base task time, the distance, and the time per meter. The array *priorityData* contains the numerical inputs used by `TaskPriority` to compute the priority score, such as dirtiness, days since cleaning, environmental risk, event priority, contextual factor, and deadline. The array *healthData* contains the health-related boolean inputs used by the priority contract.

This representation follows the current Solidity prototype: it does not include a persistent on-chain task identifier, a complete task registry, or a provenance reference to the off-chain agent output. These elements belong to the target architecture and could be added in a future version if the system were extended with an on-chain pending queue or a complete task-management registry.

This interface keeps the blockchain layer bounded, in fact, the full contractualist reasoning of the off-chain agent is not stored on-chain. Instead, the smart contracts validate the selected task using explicit inputs, user state, preferences, priority rules, robot resources, and the internal sponsorship or balance policy.

For the current prototype, each submitted task is associated with one primary affected user keeping the model manageable and matching the implemented validation function. This does not mean that the associated student is the only person who may be affected by the action, for example, a common-area cleaning task can affect several residents. In the current prototype, however, only the preferences of the student associated with the task are explicitly represented and checked on-chain. The model therefore verifies compatibility with one registered student's preferences, rather than guaranteeing that all affected people have been consulted or protected. This simplification is easiest to interpret for private-room tasks, where the associated student can reasonably be treated as the main affected user. For shared spaces, the same one-user check should be understood only as a prototype limitation. Situations in which the same cleaning action affects several users, or in which users have conflicting preferences, are important but are treated as future extensions.

4.4 Constraint Categories

The model distinguishes five categories of checks and information: the first is *user admissibility*, which states that a candidate task may enter the system only if it respects the declared preferences of the affected user. In the prototype, time preferences are treated

as hard admissibility constraints, consequently if a candidate task violates the affected student's declared preference, a high priority score should not override that violation.

The second category is *authorization and security*, which verify whether the caller is allowed to interact with the smart contracts and whether the affected user is registered, active, and associated with the correct role. This protects the system from unauthorized input and makes each request attributable to a recognized address.

The third category is *operational feasibility*: even if a task is admissible from the user's point of view, the robot may not be able to execute it at the current moment. For example, the robot may not have enough battery or available time, but this does not necessarily make the original request invalid; it only means that the task cannot be selected for execution now.

Then, the fourth category is *priority and ranking*, which interprets priority as an ordering mechanism among tasks that are already admissible and currently feasible. It helps the system decide which eligible pending task should be selected first.

Finally, the fifth category is *external context*: some variables are not naturally available on-chain and must enter the system through an oracle. In the prototype, weather is used as a simple demonstration case, however in other applications, the same mechanism could provide sensor measurements, environmental data, or institutional information.

4.5 Task Lifecycle and Pending-State Management

The lifecycle shown in Figure 4.2 represents the target architecture: in the current implementation, the pending queue and the complete on-chain selection of the next task are not implemented as Solidity mechanisms. They are treated as part of the broader architectural model and can be represented off-chain, for example through the dashboard. The implemented proof of concept focuses instead on validating an already selected task, checking explicit constraints, computing or verifying its priority, and recording the validation outcome on-chain.

In the target architecture, when a candidate task is submitted, the system first performs admissibility checks, then, if the affected user is valid and the declared preferences are respected, the task can enter a pending state. At this stage, the task is not necessarily executed immediately, but remains available for comparison with other candidate tasks, together with the relevant task data and provenance reference.

This pending state is important because the environment is dynamic and a task may be admissible but not executable at the moment of submission. For example, the robot may temporarily lack battery, available time, or position-related resources. In the target architecture, such a task would normally remain pending rather than be permanently rejected, and its feasibility could be checked again when the robot or controller requests the next task.

In a complete on-chain version of the architecture, the smart contract would inspect the pending tasks, revalidate dynamic conditions, recompute current priority values, exclude tasks that are currently infeasible, and select the highest-ranked eligible task. The score and the selection result could then be recorded so that the ordering itself becomes auditable. Therefore if the dashboard shows candidate or pending tasks, this should be

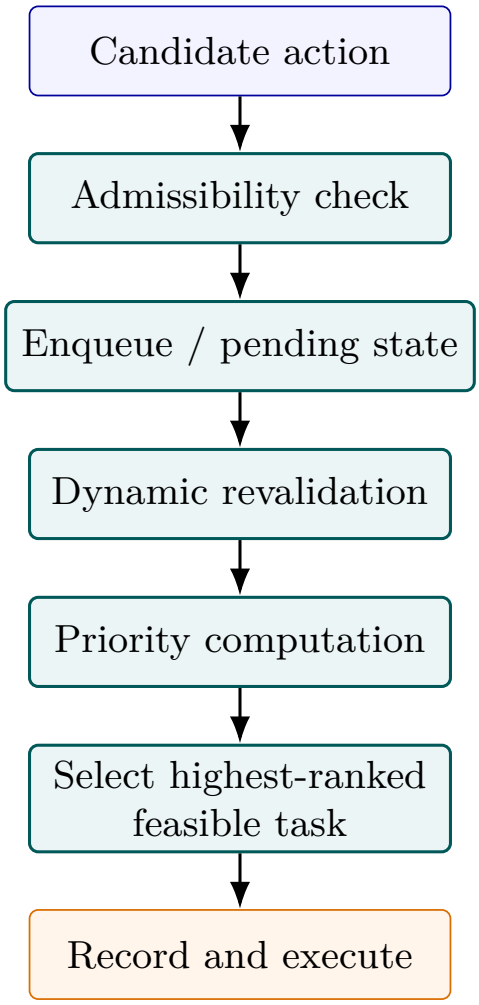


Figure 4.2: Target lifecycle of a candidate cleaning task

understood as an off-chain visualization of the broader architecture, not as evidence that the queue is fully managed by Solidity.

The current proof of concept implements a narrower version of this lifecycle: the Solidity contracts do not maintain the full pending queue or perform complete on-chain task selection, instead, the selected task is provided to the contracts, which verify the affected user, check preferences and robot resources, compute or check the priority score, apply the priority threshold, and record the validation result.

4.6 Priority Model

The weighted priority model used in the thesis is an illustrative engineering model that makes the ranking process explicit and inspectable, and does not define contractualist morality. In simplified form:

$$P = w_1 D + w_2 H + w_3 U + w_4 C + w_5 E$$

where D represents dirtiness, H health-related needs, U urgency, C contextual conditions, and E external context provided through an oracle. Each component is normalized on a scale from 0 to 100: a value close to 0 means that the factor has little influence on urgency, while a value close to 100 means that the factor strongly increases the priority of the task.

The weights $w_1, \ldots, w_5$ express the relative importance of the components. In the prototype, they are configurable parameters and should satisfy:

$$w_i \geq 0, \qquad \sum_{i=1}^{5} w_i = 100$$

With this convention, the priority score P ranges from 0 to 10000 if the weights are represented as integer percentages and each component ranges from 0 to 100, the scale is therefore implementation-oriented, matching the Solidity implementation, where all five weighted components are summed.

The numerical values used inside the models should be interpreted as configurable prototype parameters. For example, values assigned to asthma, allergies, flu, dirtiness levels, contextual factors, or weather conditions are not presented as medically or empirically validated quantitative relationships, but they are used to demonstrate how different kinds of information can be translated into a transparent ranking rule.

4.7 Priority Threshold in the Current Prototype

The current Solidity prototype includes a configurable `PRIORITY_THRESHOLD` used as a simplified validation filter after a priority score has been computed for the task selected. In particular, as already noticed in 4.5, the ranking of candidate tasks is not fully managed inside the smart contract; it can be performed by the off-chain agent or controller, while the contract verifies that the selected task satisfies the minimum priority condition before execution.

This choice keeps the prototype simple, but it should not be interpreted as the final conceptual role of priority. From the model perspective, priority is mainly a ranking criterion: it helps compare eligible tasks and identify which one should be executed first. A task with a lower score is not necessarily ethically invalid but it may simply be less urgent than another task at that moment.

For this reason, the threshold should be described as an implementation simplification rather than as a general ethical acceptance rule and a more complete version of the system could store candidate tasks in an on-chain pending queue, recompute their scores dynamically, and select the highest feasible task directly in the smart contract. In the current prototype, however, the smart contract focuses on validating and recording the selected task, while the broader ranking process remains off-chain.

4.8 Oracle and External Context

The weather factor is included to demonstrate how the architecture can bring off-chain contextual information into the smart contract framework. The important pattern is represented in 4.3

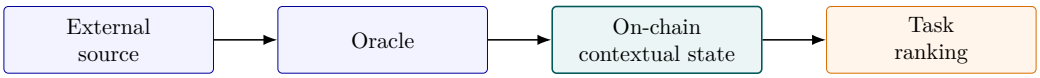


Figure 4.3: Oracle-based flow for external contextual information.

In the prototype, Open Meteo provides weather information, Chainlink CRE acts as the oracle workflow layer, and the smart contract stores the resulting contextual value. The purpose of this value is not to define a scientifically validated cleaning model based on weather, that is to show that a transparent task-management system can include external information without making the smart contract directly call an external API. This aspect is discussed in more detail in Chapter 6, where the oracle workflow and its interaction with the smart contract are presented.

4.9 Paymaster-Inspired Accessibility Feature

The model also includes a paymaster-inspired mechanism related to account abstraction in ERC-4337 [24]. This mechanism is useful because access to the system should not depend only on whether a student can directly pay transaction costs, indeed, in the prototype, the system can cover a limited number of task costs for each registered student according to predefined rules.

This feature should be understood as an accessibility mechanism which supports usability and access fairness, but it does not determine whether a task is justified or which task should be selected first.

4.10 Transparency and Verifiability

Since the robot is supposed to operate in a shared environment and makes decisions that affect users, its behaviour must be understandable and auditable, for this reason transparency and verifiability are fundamental requirements of the model. Transparency means that the rules governing the system are visible and that users can understand which factors influenced the process while verifiability means that decisions and state transitions can be checked after they are made.

In a blockchain architecture, this is achieved by storing relevant state changes and events on-chain, for example, the system can record that a candidate task was submitted, that admissibility checks were performed, that a task entered the pending state, that a priority score was recomputed and that a task was selected. Users will not need to rely only on the robot as a black box; they can inspect the smart contract logic and verify the validation steps exposed to the blockchain.

From a contractualist-inspired point of view, this is essential, in fact, a decision is not only accepted because the robot performs it correctly, but because the system can show why it was considered admissible, how it was ranked, and under which conditions it was selected. The blockchain component therefore supports accountability by preserving the link between rules, data, task state, and action.

Chapter 5

Smart Contract Implementation

5.1 Development Environment

After defining the model and the information managed by the system, this chapter explains how the architecture was implemented through smart contracts. The implementation was developed using Solidity language and Remix, Figure 5.1, an online development environment that allows writing, compiling, deploying, and testing Ethereum smart contracts directly from the browser.

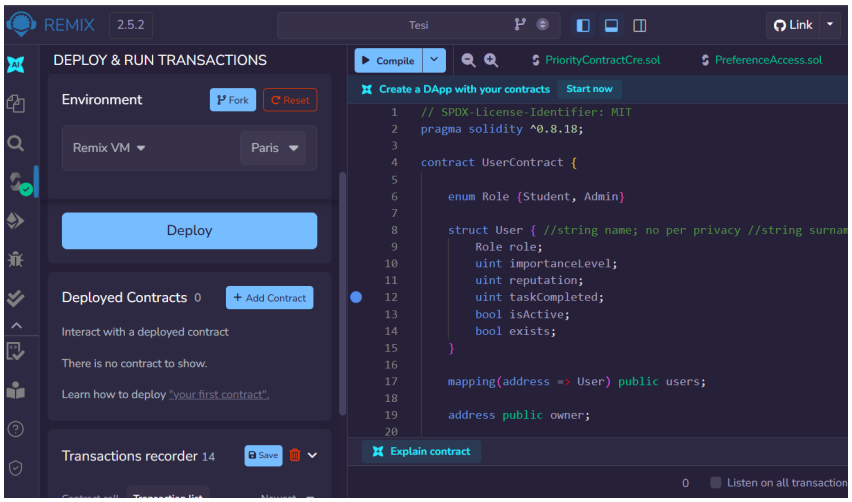


Figure 5.1: Remix interface

Remix is useful in the early stages of development because it provides a fast and controlled environment since a contract can be deployed and tested without immediately using a public blockchain. This is particularly important in the thesis because the smart contracts implement several validation rules, and each rule must be tested before moving to a more realistic network.

5.1.1 Remix VM

The first testing environment in Remix is the Remix Virtual Machine, Figure 5.4, a local blockchain simulation that runs inside the browser. It behaves like a temporary Ethereum environment where contracts can be deployed, functions can be called, and transactions can be simulated without using real ETH or test ETH.

Therefore this environment makes the system simpler as transactions can be executed immediately, predefined accounts are available, and no external wallet is required. However, the Remix VM is not persistent and when the browser session is closed or the environment is reset, the deployed contracts and their state disappear. For this reason, Remix VM is mainly used as a first development and debugging tool.

5.1.2 Ganache

Ganache provides another local testing environment which, unlike Remix VM, runs inside the browser, locally on the developer's machine, simulating a private Ethereum blockchain. This environment provides predefined accounts with test funds and allows contracts to be deployed and tested without relying on a public network.

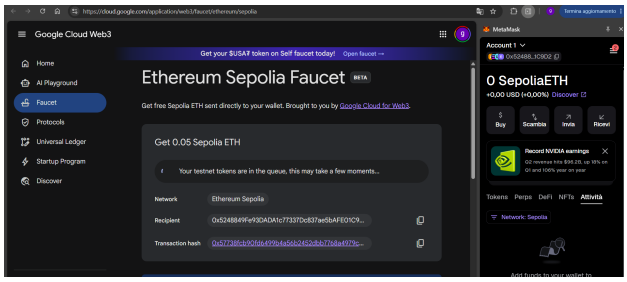
Using Ganache before a testnet deployment is useful because transactions are executed locally and blocks are created almost immediately, making debugging faster and avoiding the need to obtain test ETH from a faucet. In this project, Ganache represents an intermediate step between the browser tests in Remix VM and the later deployment on a public testnet.

5.1.3 Sepolia

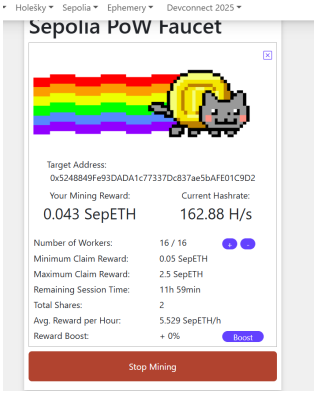
Sepolia, shown in Figure 5.5 is a public Ethereum test network which is able to reproduce the behaviour of a real blockchain network, but uses test ETH instead of real ETH. To deploy a contract on Sepolia, the developer must connect Remix to a wallet such as MetaMask and obtain test ETH from a faucet, that is a service that sends a small amount of testnet cryptocurrency to a wallet address. In this project, faucets were used to obtain Sepolia ETH, which is required to pay gas fees when deploying contracts and sending transactions on the Sepolia testnet. The request can usually be made by connecting a wallet or by pasting the wallet address into the faucet interface, some faucets distribute test ETH directly, while others require a short proof-of-work process before the reward can be claimed. Figures 5.2a, 5.2b show two examples used during the testing phase.

Compared with Remix VM and Ganache, Sepolia is slower because transactions must be propagated, included in blocks, and confirmed by the network. However, it is also more realistic as contracts deployed on Sepolia are persistent and can interact with external tools, wallets, frontends, and oracle workflows. For this reason, Sepolia is useful when the contract has already been tested locally and the goal is to verify its behaviour in a public Ethereum environment.

The three environments therefore represent an incremental validation path: the prototype is first tested in a fast simulated environment, then in a local persistent blockchain, and finally in a public testnet that is closer to a realistic deployment 5.3.



(a) Google Cloud Web3 faucet



(b) Proof-of-work faucet

Figure 5.2: Alternative methods for obtaining Sepolia test ETH.



Figure 5.3: Incremental testing path from simulation to public testnet deployment.

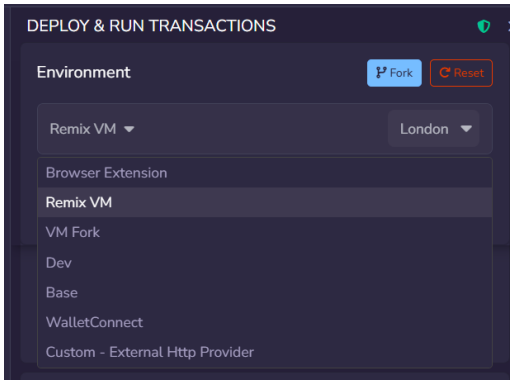


Figure 5.4: Remix VM

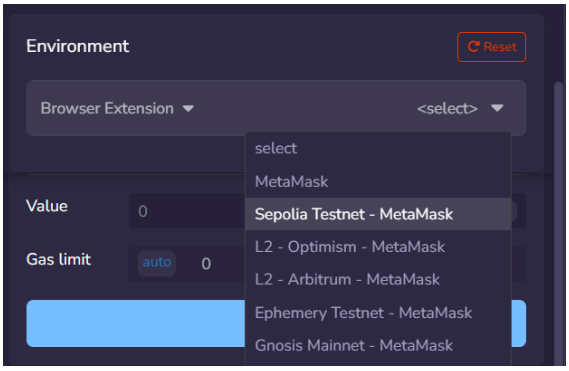


Figure 5.5: Sepolia Testnet

5.2 Smart Contract Architecture

As mentioned in the previous section, the development path follows a progressive approach, moving from Remix VM for initial testing, to Ganache for local blockchain validation, and finally to the Sepolia testnet for testing in a public blockchain environment. The system is organized into four main smart contracts having each a specific responsibility, following the principle of separation of concerns, in fact, instead of placing all the logic inside a single contract, the architecture separates user management, preferences, priority computation, and task validation.

- **User Contract**: manages users, roles, activity status, reputation, and basic registration information.
- **Preference Contract**: stores and checks student preferences, like the preferred time slots.
- **TaskPriority Contract**: computes the priority score of a cleaning task using contextual factors and external data.
- **TaskValidation Contract**: verifies whether a task can be accepted by checking the student, preferences, robot resources, priority threshold, and paymaster logic.

This modular design makes the system easier to understand and test and also reflects the model introduced in Section 4.4: stable rules are encoded in smart contracts, while dynamic informations are stored or provided as input when a task is evaluated. The contracts communicate through interfaces, so that one contract can call selected functions of another contract without needing to include all its internal code.

Figure 5.6 shows the dependency structure between the four contracts. `UserContract` is deployed first and provides the address used by both `PreferenceContract` and `TaskPriority`. Finally, `TaskValidation` receives the addresses of all three previous contracts and coordinates the validation process.

The complete Solidity implementation will be referenced through the project GitHub repository:
https://github.com/giuliabodrito/Tesi-Giulia-Bodrito-Politecnico-di-Torino
For this reason, this section only report the code fragments that are necessary to understand the model.

5.3 User Contract

The first contract is the `User Contract`, which has the role to define who can interact with the system. Since the architecture represents a closed environment, such as a student residence, the system must distinguish between authorized and unauthorized users and, as already discussed in 4.1, this is done by associating each user with a blockchain address.

The contract defines two roles:

- **Student**: a user who can be associated with preferences and cleaning requests.

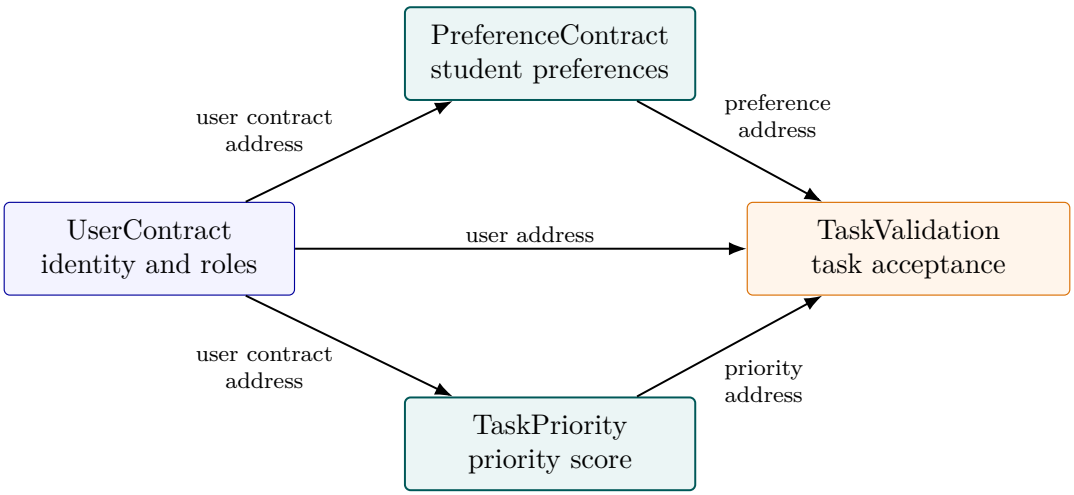


Figure 5.6: Dependency structure of the main smart contracts.

- **Admin**: a user who can register other users, update roles, and deactivate accounts.

These roles are represented through the Solidity enum `Role` which make the contract easier to extend, because additional roles could be added in the future if the system required them, for example a guest, a supervisor, or a building manager.

Each user is represented by a `User` struct, which stores the role, an importance level, the user's reputation, the number of completed tasks, an activity flag, and an existence flag. Personal data such as name and surname are intentionally not stored on-chain in this version: blockchain data are persistent and visible, so sensitive information should either be avoided or stored off-chain, possibly through a hash or another privacy mechanism.

The mapping:

```
mapping(address => User) public users
```

connects each blockchain address to its corresponding user information, and students can be identified by their address that is registered by the administrator.

5.3.1 Owner and Administrator Logic

When the contract is deployed, the address that deploys it becomes the `owner`, and the constructor also registers this address as an administrator. This is useful because the system needs at least one authorized account able to initialize and manage the list of users.

The contract uses the modifier `onlyAdmin` to protect administrative functions: a function marked with this modifier can be executed only by the owner or by an active user with the admin role, preventing ordinary students or external addresses from registering users or changing roles.

The contract also defines the modifier `onlyActiveUser` which checks that the caller is active and exists in the system. Although not every function needs this modifier, it expresses an important design principle: a user should not be able to interact with the system if they are not registered or if their account has been deactivated.

5.3.2 User Registration

The function `registerUser` is used to add a new user to the system and receives as input the user's address, the role, and the importance level. Then the contract creates a new `User` entry, initializes the user as active and existing and it also sets an initial reputation and initializes the number of completed tasks to zero.

This function is restricted to administrators, making them acting as entities that verifies whether a person belongs to the residence or to the controlled environment.

When a user is registered, the contract emits the event `UserRegistered`, in particular events are useful because they allow external applications, such as a frontend or an indexing service, to track relevant changes without continuously reading the entire contract state. In the context of this thesis, events support transparency because they create a trace of important operations.

5.3.3 Role Update and Deactivation

The function `setRole` allows an administrator to update the role of an existing user, but before changing the role, the contract first checks that the user is already registered. This explains the condition `User not registered`: the system should not assign a role to an address that does not yet exist in the user registry, avoinding in this way inconsistent states and making user management more controlled.

The function `deactivateUser,` which marks a user as inactive, is different from deleting the user. In fact keeping the entry while setting `isActive` to false preserves the history of the user and avoids losing information, while still preventing that user from participating in the system. This is useful in a student residence, where a student may leave the residence and should no longer be allowed to submit preferences or requests.

5.3.4 Reading User Information

Finally, the function `getUser` allows reading the information associated with a given address. Since the `users` mapping is public, Solidity already creates an automatic getter, but an explicit function can make the interface clearer when interacting with the contract from Remix or from an external application.

Overall, the `User Contract` provides the identity and access control layer of the architecture. It does not compute task priority and does not validate cleaning tasks directly, but, instead, provides the information that the other contracts need in order to check whether a student is valid, active, and authorized making the foundation of the closed and verifiable system described in the proposed model.

5.4 Preference Contract

The purpose of the second contract, the `Preference Contract`, is to store and check the preferences expressed by students. This contract is important because preferences are the element that makes the system more user centered and while without them the robot would follow only fixed rules, with preferences the system can adapt its behaviour to the needs and routines of the people affected by the cleaning task.

In the current implementation, preferences are mainly used to represent acceptable time slots, or example, a student may indicate that cleaning is acceptable during a certain hour and later, when the validation contract checks whether a task can be executed, it can ask the preference contract whether the current time is compatible with the student's preferences. This creates a direct connection between the contractualist idea of respecting the affected user and the technical validation process introduced in Section 4.5.

5.4.1 Connection with User Contract

`Preference Contract` receives the address of `User Contract` in the constructor and stores it through an interface called `IUserContract`. This interface exposes only the information needed by the preference contract, namely the user's role, reputation, active status, and existence flag.

This design keeps the architecture modular as preference contract does not need to duplicate the user registry, because user identity and authorization are already managed by the `User Contract`. The latter is indeed queried whenever there is the need to verify whether an address is allowed to submit preferences.

5.4.2 Preference Structure

Preferences are represented through a struct called `Preference` which contains four main fields:

```
struct Preference {
    PreferenceType preferenceType;
    uint8 hour;
    uint value;
    uint timestamp;
}
```

- `preferenceType`: the type of preference, for example a time related preference.
- `hour`: the hour of the day associated with the preference, from 0 to 23.
- `value`: the importance or value assigned to the preference.
- `timestamp`: the time at which the preference was submitted.

The contract stores preferences in a mapping from user address to an array of preferences:

```
mapping(address => Preference[]) private preferences
```

This means that each student can have multiple preferences associated with their address. Using a struct is more readable than storing only a list of numbers, because each value has a clear meaning and is easier to understand what each preference means and how it should be used by the rest of the system.

The mapping is marked as `private`, and the contract exposes preferences through an administrator through`getPreferences`. This restricts access through the Solidity interface, but it does not make the data confidential on a public blockchain [34] and contract storage can still be inspected by external observers. For this reason, the prototype assumes synthetic or non-sensitive preference data and does not implement a privacy-preserving storage mechanism.

5.4.3 Submitting Preferences

The function `addPreference` allows an authorized student to add new preferences, in particular, before storing the preference, the internal function `canSubmitPreferences` is used to check weather the caller is legitimate, making the system closed and protected from unauthorized inputs.

The authorization rule requires that the user satisfies the following conditions:

- exists in `User Contract`;
- is active;
- has the student role;
- has sufficient reputation.

Moreover the function also checks that the value is valid and that the hour is between 0 and 23. After the preference is added, the contract emits the event `PreferenceAdded` creating a trace of the operation and allowing external applications to observe that a new preference has been submitted.

5.4.4 Checking Time Preferences

The function `isTimePreferred` is used to check whether a specific hour is acceptable for a user, specifically it reads the list of preferences associated with the user and searches for a time preference matching the requested hour, in particular, if the user has not submitted any preference, the function returns `true`. This is a prototype policy of no time restriction: it should not be interpreted as affirmative consent by the student, but it only means that the contract has no stored time constraint to enforce. This design choice makes the system flexible: absence of preferences is interpreted as no restriction, rather than as a reason to reject every task. If preferences are present, the function returns `true` only when the requested hour matches one of the accepted time slots.

This function is used by the validation process to prevent the robot from acting at inconvenient times: for example, if a student has authorized cleaning only during specific hours, the task validation contract can reject or postpone tasks outside those hours. In this way, the preference contract adds a layer of privacy and personal comfort to the robotic cleaning system.

In the current prototype, the validation logic uses only two fields of a preference record: `preferenceType` and `hour`. A time slot is accepted if at least one stored preference has type `Time` and the same hour as the hour checked by the validation contract. The fields `value` and `timestamp` are stored for traceability and possible future extensions, but they do not currently affect the validation decision.

The contract does not implement an update or deleting mechanism for preferences. New records are, in fact, appended to the user's preference list and do not automatically supersede earlier records. Repeated records are therefore redundant, while multiple time preferences are interpreted as a union of allowed hours. Since the current representation does not include negative preferences, conflicts between allowed and disallowed time slots are not resolved on-chain.

The time-preference check concerns only the validation/start instant and does not guarantee that the entire cleaning interval remains inside an allowed time slot. This is a limitation of the current prototype; a production version should compare the estimated interval with the user's allowed time windows.

5.4.5 Access Helper

The contract also provides the function `checkMyAccess`, a helper function that allows a user or external application to check whether the current caller is allowed to submit preferences. It simply applies the authorization logic to `msg.sender`, which represents the address that is currently calling the function, for example the student address or the frontend wallet interacting with the contract.

The purpose of `checkMyAccess` is usability: instead of requiring a frontend to manually pass an address to check authorization, the user can directly verify their own access status. This is useful in a future application interface, where the system could show whether a student is allowed to submit preferences before displaying the preference form.

5.4.6 Preferences as Feedback for Future Adaptation

In the current implementation, preferences are used directly by the validation process 5.6, especially for time compatibility. However, preferences can also have a second role in future developments: they can become feedback signals for adapting the priority model.

For example, if many students repeatedly indicate that urgency should be more important than other factors, the system could use this feedback to adjust the weights of the priority function 5.12. This adaptation should not happen arbitrarily on-chain, instead a more scalable approach would be to analyze preference and feedback data off-chain, possibly through an AI or data-analysis component, and then update only selected parameters in the smart contract under controlled conditions.

An even more advanced version could maintain personalized weight profiles for each user where the priority computation would not only depend on general rules, but also on specific preferences of the user learned over time, making the system more adaptive. On the other hand this also introduces additional privacy and scalability challenges and, for this reason, the current prototype keeps the preference mechanism simple and transparent, while leaving personalized adaptive weighting as future work.

Overall, `Preference Contract` provides the user's preference layer of the architecture: connects preferences to registered blockchain addresses, checks whether students are authorized to submit them, and exposes a simple function to verify whether a time slot is acceptable. This contract therefore supports the contractualist objective of making robotic actions sensitive to the predilections of the people affected.

5.5 TaskPriority Contract

The third contract, `TaskPriority Contract`, computes the priority score of a cleaning task. This contract does not register users and does not store user preferences, since, as shown in the previous sections, those responsibilities belong to `UserContract` and `PreferenceContract`. Instead, `TaskPriority` focuses on the priority model: it receives task's inputs, combines them with the current weather factor, and returns a numerical score that can later be used by the validation contract. Moreover is also designed to work with the Chainlink CRE oracle workflow, but the details of this workflow will be

discussed in chapter 6. In this section, the focus is only on the priority logic implemented inside the smart contract.

The contract uses the interface `IUserContractForPriority` to interact with the user contract without duplicating the user registry. Through this interface, `TaskPriority` can read the role and status of an address and verify, inside the `onlyAdmin` modifier, whether the caller is an active admin.

5.5.1 Purpose of the Priority Contract

The priority contract is necessary because the robot may receive several cleaning requests, but it can not execute all of them at the same time. As already discussed previously, the system needs a transparent rule for deciding which task should be considered more urgent. In this implementation, priority is computed through a weighted formula, leading to an explicit decision process: the score depends on a defined set of factors and on weights that can be inspected on-chain.

The contract uses five weights:

$$w_1, w_2, w_3, w_4, w_5$$

These weights correspond to the main dimensions of the priority score:

- **health**: whether the student has conditions that make cleaning more important;
- **urgency**: whether there is an event or a deadline;
- **dirtiness**: how dirty the room or area is;
- **contextual**: a value representing contextual needs that are not captured by the other factors;
- **weather factor**: an external environmental factor updated through Chainlink CRE 6.5.

The relationship between these components and the priority contract is summarized in Figure 5.7.

The weights are not declared as constants because they are not permanently fixed at deployment time. Instead, they can be updated by an authorized administrator through `updateWeights`, which makes the system adaptable if later a factor is decided to have more or less importance. Moreover, the update is still transparent because the new weights are stored on-chain and the contract emits the event `WeightsUpdated`.

5.5.2 Access Control

Some functions of the priority contract are restricted to authorized accounts such as `updateWeights`, that can only be called by the owner or by an active admin registered in the user contract, preventing arbitrary users from changing the parameters of the priority model. The oracle access control, including the authorized forwarder used by Chainlink CRE, is discussed in the next chapter.

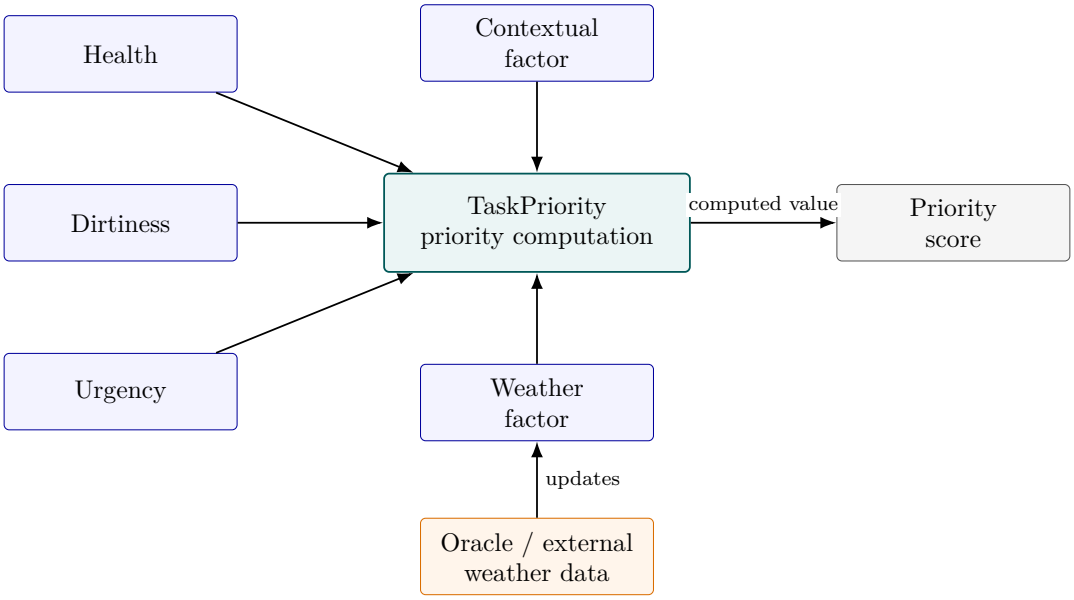


Figure 5.7: Factors used by `TaskPriority` to compute the priority score.

5.5.3 Weight Update

As introduced in 5.5.1 the function `updateWeights` receives five new values and assigns them to `w1`, `w2`, `w3`, `w4`, and `w5`. The contract requires that their sum is equal to 100:

$$w_1 + w_2 + w_3 + w_4 + w_5 = 100$$

This constraint makes the relative meaning of the weights clearer, in fact, if one factor becomes more important, another factor must become less important, or the total would exceed 100. In this way, the system avoids arbitrary weight inflation.

It is important to note that the current implementation does not divide the final score by 100, therefore, the result should be understood as a weighted priority score, not necessarily as a percentage between 0 and 100. The configurable threshold used by the validation contract is calibrated according to this scoring scale allowing the prototype to test both passing and failing priority cases without redeploying the validation contract.

As already discussed in 5.4.6, future work, the weights could be adapted using feedback from students.

5.5.4 Health Score

The first factor of Section 5.5.1, health score, is computed inside the smart contract by the function `calculateHealthScore`. In the initial design, the health component was modeled as a simple weighted sum of three boolean variables: asthma, allergies, and flu:

$$HealthScore = \min(100,\ 40A + 30L + 20F)$$

where:

- $A = 1$ if the student has asthma, otherwise $A = 0$;
- $L = 1$ if the student has allergies, otherwise $L = 0$;
- $F = 1$ if the student has flu, otherwise $F = 0$.

In this first version, the purpose was mainly to avoid passing an arbitrary precomputed health score to `computePriority`. The smart contract received the relevant health conditions and computed the score internally, making the calculation more transparent and verifiable.

In the current version, the health component has been refined, the contract still receives the three boolean variables:

$$A,\ L,\ F \in \{0,1\}$$

where A represents asthma, L represents allergies, and F represents flu. In addition, the model includes an environmental risk parameter $E \in [0,100]$, which describes the quality of the environment in the area to be cleaned, and may represent factors such as dust, humidity, mold, or poor ventilation.

The health risk score is defined as:

$$R_H = 35A + 20L + 25F + 15AL + 20AF + 5LF + \frac{E(10A + 5L + 5F)}{100}$$

The first three terms represent the individual contribution of each health condition, the interaction terms AL, AF, and LF increase the score when multiple conditions are present at the same time. Finally, the environmental term increases the risk proportionally to the environmental risk index E. The raw value can exceed 100: for example, if all three health indicators are active and the environmental-risk contribution is maximal, the raw expression may reach 140. Therefore, the implemented model uses a clipped health component:

$$H = \begin{cases} R_H, & \text{if } R_H \leq 100, \\ 100, & \text{if } R_H > 100. \end{cases}$$

This clipping keeps the health component in the same 0-100 range as the other priority components, preserving the nominal 0-10000 range of the weighted priority score.

The Solidity implementation is can be seen in:
https://github.com/giuliabodrito/Tesi-Giulia-Bodrito-Politecnico-di-Torino

This formulation does not claim to be a medical model, but rather, it provides a transparent way to represent health priority within the scope of the smart contract. Compared with the initial version, the model is more expressive because it considers not only isolated health conditions, but also their combined effects and the environmental context of the cleaning task.

Future versions could extend this component further by including additional needs, such as mobility disabilities or other accessibility constraints. These extensions should be designed carefully, so that the system supports students with specific needs without introducing discriminatory or unjustified priority rules.

5.5.5 Time and Urgency

The priority function also includes a temporal component, which represents the urgency of each task. Specifically, the contract compares the input `deadLine`, that represents the timestamp by which the task should become urgent, with the `block.timestamp`[1]

In the first version of the model, urgency was computed through a simple threshold's rule:

$$timeFactor = \begin{cases} 100 & \text{if } block.timestamp \geq deadLine \\ 50 & \text{if } deadLine - block.timestamp \leq 1 \text{ day} \\ 0 & \text{otherwise} \end{cases}$$

[1]Smart contracts cannot access the clock of a user's device or an external server. Instead, the blockchain provides its own time reference through the timestamp associated with each block. In Solidity, the hour, is derived from `block.timestamp`, a globally available Solidity variable that returns the timestamp of the current block [35]. The expression used in the prototype computes an hour modulo 24 from blockchain time and should not be interpreted automatically as the residence's local clock, especially when time zones or daylight-saving changes are relevant.

This approach is easy to implement and verify, but it has a limitation: all tasks within the same time interval receive the same temporal score. For example, a task whose deadline is one hour away and a task whose deadline is twenty-three hours away would both receive the same value.

For this reason, the temporal component was refined into a linear function of the remaining time: let t be the current timestamp, D the task deadline, and $T_{\max}$ the maximum time window within which the task starts becoming urgent. The updated urgency score is defined as:

$$U(t, D) = \begin{cases} 100, & \text{if } t \geq D, \\ 100 \cdot \max\left(0, 1 - \dfrac{D - t}{T_{\max}}\right), & \text{if } t < D. \end{cases} \tag{5.1}$$

where $D - t$ is the remaining time before the deadline. This formulation means that the urgency score is zero when the deadline is still far away, then increases progressively as the deadline approaches, and reaches 100 when the deadline is met or exceeded.

In this implementation, $T_{\max}$ is set to one day, therefore, a task with more than one day remaining receives no additional temporal urgency. A task with approximately twelve hours remaining receives a score close to 50, while a task whose deadline has already passed receives the maximum score of 100.

The final urgency component combines the temporal score, present in equation 5.1, with an event priority:

$$Urgency = eventPriority + U(t, D)$$

`eventPriority` represents external or social urgency, such as the presence of guests, an upcoming event, or a situation requiring faster cleaning. The result is restricted at 100 in the Solidity implementation, so that urgency remains a normalized component of the priority model.

As shown in the github code the Solidity implementation uses integer arithmetic, since floating point numbers are not natively supported by the Ethereum Virtual Machine.

5.5.6 Dirtiness Score

In the first version of the priority model, the dirtiness component, denoted by D, was computed with a simple linear rule:

$$D = sensorLevel + daysSinceCleaning$$

where `sensorLevel` represents the current dirtiness detected by the sensor, while `daysSinceCleaning` represents the time passed since the last cleaning operation. This formulation is easy to understand, but it has a limitation: the priority increases at a constant rate, regardless of how dirty the room already is.

For this reason, the model was later refined by introducing a non linear dirtiness score: the purpose is not to claim that dirtiness itself grows exponentially; rather, the model assumes that the priority associated with dirtiness should increase more rapidly when hygiene conditions become worse. This choice is motivated by the idea that accumulated

dirt may amplify health risks, for example by creating more favorable conditions for dust, mold, or bacterial proliferation.

The refined dirtiness score is modeled as a saturating exponential function:

$$D = 100 \cdot \left(1 - e^{-kd}\right) \tag{5.2}$$

Where d is the dirtiness index and is computed as the average of the sensor based dirtiness level and the number of days since the last cleaning:

$$d = \frac{sensorLevel + daysSinceCleaning}{2}. \tag{5.3}$$

This normalization keeps the index in the interval [0,100], assuming that both input variables are also expressed on a scale from 0 to 100. This makes the dirtiness component comparable with the other components of the priority model.

The parameter k in 5.2 controls how quickly the score increases: higher value of k makes the score grow faster, while a lower value produces a smoother increase. The dirtiness score increases with the dirtiness index, but it follows a saturating behaviour. The score, in fact, grows quickly for low and medium dirtiness levels and then approaches an upper bound, avoiding unbounded growth and keep the dirtiness component within the same range as the other priority factors.

This behaviour can be verified from the derivatives of the reference function. For $k > 0$, the first derivative is

$$D'(d) = 100ke^{-kd} > 0,$$

so the dirtiness score is increasing with respect to the dirtiness index. The second derivative is

$$D''(d) = -100k^2e^{-kd} < 0,$$

so the slope decreases as d grows. Therefore, the function is increasing and concave: higher dirtiness values still increase the score, but the increase gradually saturates rather than accelerating.

However, since the Ethereum Virtual Machine does not natively support floating point arithmetic or exponential functions, the formula 5.2 would make the contract more complex and more expensive in terms of gas. For this reason, the smart contract uses an integer approximation that preserves the same qualitative behavior: the score increases with dirtiness and remains bounded.

The approximation is inspired by the second order Taylor expansion of the exponential function:

$$e^z \approx 1 + z + \frac{z^2}{2} \tag{5.4}$$

The theoretical dirtiness score contains the term $1 - e^{-z}$, where the variable z is defined as:

$$z = k \cdot d \tag{5.5}$$

Using the identity $e^{-z} = 1/e^z$, the term can be written as:

$$1 - e^{-z} = 1 - \frac{1}{e^z}$$

Using the second order approximation of e^z in 5.4, this becomes:

$$1 - e^{-z} \approx 1 - \frac{1}{1 + z + \frac{z^2}{2}}$$

which can be rewritten as:

$$1 - e^{-z} \approx \frac{z + \frac{z^2}{2}}{1 + z + \frac{z^2}{2}} \tag{5.6}$$

Therefore, the theoretical dirtiness score in equation 5.2 can be approximated as:

$$D \approx 100 \cdot \frac{z + \frac{z^2}{2}}{1 + z + \frac{z^2}{2}}. \tag{5.7}$$

The parameter k in equation 5.2 is a positive real coefficient. In this model it is chosen as a small decimal value, because the dirtiness index ranges from 0 to 100 and the score should increase smoothly rather than immediately saturating. However, in Solidity, floating point values are avoided by introducing a scaling factor S, therefore the parameter k is represented as the ratio between two integer constrants, K and S:

$$k = \frac{K}{S} \tag{5.8}$$

where S is the scaling factor and K is the integer numerator. In the contract, these values are set as:

$$K = 30, \qquad S = 1000.$$

Therefore, the value of k used by the model is:

$$k = \frac{30}{1000} = 0.03. \tag{5.9}$$

As defined in Equation 5.3, the dirtiness index is the average of `sensorLevel` and `daysSinceCleaning`.

To avoid floating-point arithmetic, the contract works with a scaled version of the variable $z = k \cdot d$. This scaled variable is called x, and is defined as:

$$x = K \cdot d = K \cdot \frac{sensorLevel + daysSinceCleaning}{2}. \tag{5.10}$$

Since x represents the scaled version of z, namely $x = S \cdot z$, the approximation in 5.7 can be rewritten in integer form as:

$$D \approx 100 \cdot \frac{x + \frac{x^2}{2S}}{S + x + \frac{x^2}{2S}}. \tag{5.11}$$

The parameter $k = 0.03$ present in 5.9 determines the sensitivity of the dirtiness score and was selected to obtain a gradual increase of the score as the dirtiness index grows, without making the function saturate too quickly. Like the other numerical parameters of the model, k is not a universal constant, but a modifiable value that can be changed according to the real robot, the sensors, and the cleaning environment.

The implemented rational function approximates the exponential reference model preserving the desired monotonic and bounded behaviour, even if it is not numerically identical to the exponential function. For $k = 0.03$, the approximation underestimates the reference value by about 1.99 points at $d = 25$, 5.27 points at $d = 50$, and 6.79 points at $d = 100$ as shown in 5.1.

Since the dirtiness component is only one term of a heuristic priority score, this error is acceptable for the prototype, but it should be interpreted as an approximation rather than an exact implementation of the exponential model.

Index d	**Exponential** $D(d)$	**Rational** $\hat{D}(d)$	**Absolute error**
25	52.76	50.77	1.99
50	77.69	72.41	5.27
100	95.02	88.24	6.79

Table 5.1: Approximation error of the rational dirtiness function for $k = 0.03$.

Thus, the mathematical model describes the intended behavior of the dirtiness component, while the Solidity code provides a deterministic and gas compatible approximation suitable for on-chain execution.

5.5.7 Contextual Factor

The input previously named `culturalFactor` is now interpreted more broadly as a contextual factor. This change makes the model more precise, because the score refers to operational and environmental conditions of the residence, rather than to vague notions of culture.

The contextual factor captures aspects that are not already included in the dirtiness, health, urgency, or weather components. In a student residence, these aspects may include the type of area involved, stricter hygiene requirements, or temporary residence policies such as quiet hours or exam periods. In a future hospital scenario, the same idea could be extended to specific cleaning rules, hygiene protocols, or restricted access areas.

The contextual score is computed from two discrete variables:

$$ContextualFactor = AreaScore + HygieneScore$$

where `AreaScore` depends on the type of space and `HygieneScore` depends on the required hygiene level.

In the current prototype, `areaType` is defined according to the function of the indoor space. The corresponding values are:

$$AreaScore = \begin{cases} 5 & \text{private room} \\ 20 & \text{corridor or common passage} \\ 35 & \text{shared kitchen or high-use area} \\ 50 & \text{exposed or high-accumulation area} \end{cases}$$

Future versions could refine this parameter by also considering the surrounding environment, for instance, coastal, mountain, or volcanic areas may introduce different cleaning requirements due to sand, humidity, mud, ash, or dust. This would allow the contextual factor to represent not only the use of the indoor space, but also environmental conditions that influence dirt accumulation.

For `hygieneRequirement`, the values are:

$$HygieneScore = \begin{cases} 0 & \text{normal hygiene requirement} \\ 10 & \text{medium hygiene requirement} \\ 25 & \text{high hygiene requirement} \\ 40 & \text{critical hygiene requirement} \end{cases}$$

The values used to compute `AreaScore` and `HygieneScore` should be interpreted as configurable design parameters rather than universal constants. In this prototype, higher values are assigned to areas or situations with stricter cleaning requirements, so that shared or hygiene-sensitive spaces receive greater priority. These values were selected to create a simple and testable scoring model, but they could be modified by the system designer and adjusted based on institutional rules and expert input in a real deployment.

The final value is bounded to keep it comparable with the other normalized components:

$$ContextualFactor = \min(100,\ AreaScore + HygieneScore)$$

The Solidity implementation is shown in the github file:
https://github.com/giuliabodrito/Tesi-Giulia-Bodrito-Politecnico-di-Torino

This design makes the score easier to justify and audit, because each component is associated with an explicit operational condition.

In an intermediate version of the model, a `TimePolicy` factor was included among the contextual factors, and was later removed from the priority score because institutional time rules, such as quiet hours or exam periods, do not necessarily make a task more important; rather, they define when the robot is allowed or not allowed to operate.

Therefore, `TimePolicy` is better modeled as an execution constraint controlled by the residence administration, instead of as a positive priority factor. A future version of the system could introduce an controlled time policy module to represent these rules explicitly, such as `setResidenceTimePolicy(uint256 policyCode)`, accessible only through an `onlyAdmin` modifier.

5.5.8 Weather Factor

The weather factor is the external environmental component of the priority score and unlike the other factors, it does not come directly from the user or from manual task data, but from an external weather source. Since smart contracts cannot directly access web APIs, this information is provided through an oracle mechanism, implemented through Chainlink CRE, which retrieves weather data off-chain from the Open Meteo API and sends the relevant values to the smart contract.

In the current CRE-oriented version, the contract stores `lastUpdatedTemperature` and `weatherFactor`. The weather factor is updated by the internal function

$$updateWeatherFromTemperature$$

which receives both the temperature and a `weatherCode`. The smart contract assigns the weather factor according to an ordered set of conditions. In particular, precipitation-related weather codes have priority over temperature-based conditions:

$$weatherFactor = \begin{cases} c_1 & \text{if } weatherCode \geq 61, \\ c_2 & \text{if } weatherCode < 61 \text{ and } temperature > 30, \\ c_3 & \text{if } weatherCode < 61 \text{ and } 20 < temperature \leq 30, \\ c_4 & \text{if } weatherCode < 61 \text{ and } temperature \leq 20. \end{cases}$$

In the implemented prototype, the constants are set as $c_1 = 30$, $c_2 = 20$, $c_3 = 10$, and $c_4 = 5$.

For the purpose of this section, it is sufficient to note that weather codes from 61 upward are treated as rain or severe weather conditions, including freezing rain, snow, showers, and thunderstorms in the simplified prototype classification. This value receives the highest weather factor, independently of the temperature, while if the weather code is less than 61 the contract uses temperature thresholds to assign a lower or higher environmental influence. Higher temperatures are associated with a higher weather factor because they may make hygiene conditions more critical, for example by increasing discomfort, odors, or the need for more frequent cleaning.

Finally, the Chainlink CRE workflow and the Solidity functions used to receive weather data will be discussed in detail in the next chapter.

5.5.9 Priority Calculation

The main function of the priority contract is `computePriority` and represents the final step of the priority model: it combines the different components described in the previous sections and returns the final priority score of a task.

In the first version of the contract, `computePriority` received each input as a separate parameter. As the model became more expressive, however, the number of parameters increased and, for this reason, the current implementation groups the numerical inputs into an array called `priorityData` and the health related boolean inputs into an array called `healthData`, keeping the function interface more compact.

The array `priorityData` contains:

$$\texttt{priorityData} = [\texttt{sensorLevel},\ \texttt{daysSinceCleaning},\ \texttt{environmentalRisk},\ \texttt{eventPriority},\ \texttt{contextualFactor},\ \texttt{deadLine}]$$

The array `healthData` contains:

$$\texttt{healthData} = [\texttt{asthma}, \texttt{allergies}, \texttt{flu}]$$

Before computing the score, the contract checks that the numerical inputs are within the expected ranges:

```
require(sensorLevel <= 100, "Invalid sensor");
require(daysSinceCleaning <= 100, "Invalid days since cleaning");
require(environmentalRisk <= 100, "Invalid environmental risk");
require(contextualFactor <= 100, "Invalid contextual factor");
require(eventPriority <= 100, "Invalid event priority");
```

These requirements prevent unrealistic input values from increasing the score without control and make the expected input domain explicit.

The final score is computed as a weighted sum of the normalized components:

$$\begin{aligned}\text{PriorityScore} = {} & w_1 \cdot \text{DirtinessScore} + w_2 \cdot \text{HealthScore} \\ & + w_3 \cdot \text{Urgency} + w_4 \cdot \text{ContextualFactor} + w_5 \cdot \text{WeatherFactor}\end{aligned} \tag{5.12}$$

The corresponding Solidity function is:

```
function calculateWeightedPriority(
    uint256 dirtiness,
    uint256 healthScore,
    uint256 urgency,
    uint256 contextualFactor
) internal view returns (uint256) {
    return (
        w1 * dirtiness +
        w2 * healthScore +
        w3 * urgency +
        w4 * contextualFactor +
        w5 * weatherFactor
    );
}
```

Then the function `computePriority` calls the helper functions that compute each component of the model and then passes their results to `calculateWeightedPriority`. The implementation details are shown in the GitHub repository.

The modular structure was introduced for two reasons: first, it improves readability as each helper function corresponds to one component of the mathematical model, such as dirtiness, health, urgency, or contextual relevance. Second, it helps avoid the Solidity *stack too deep* error which occurs when a function has too many parameters and local variables at the same time. By moving the component calculations into separate functions

and by using arrays to group related inputs, the contract keeps `computePriority` smaller and easier to compile.

It is important to distinguish between values computed by the smart contract and values supplied to the contract. In the current prototype, `computePriority` receives `sensorLevel`, `daysSinceCleaning`, `environmentalRisk`, `eventPriority`, `contextualFactor`, and `deadLine` as input values. The contract checks that the bounded numerical inputs are in the expected range, but it does not independently verify that they correspond to real sensor measurements or real environmental facts and their correctness is therefore an assumption of the prototype. The contract computes the dirtiness score from `sensorLevel` and `daysSinceCleaning`, computes the health score from the three health indicators and `environmentalRisk`, and computes urgency from `eventPriority` and `block.timestamp`, while `contextualFactor` is already received as a numerical input by `computePriority`.

Field	**Domain**	**Use in the prototype**
`sensorLevel`	0–100	Dirtiness input; not directly measured.
`daysSinceCleaning`	0–100	Dirtiness input from off-chain records.
`environmentalRisk`	0–100	Input for the health-score calculation.
`healthData`	Boolean flags	Health-score input; medical truth is not verified.
`eventPriority`	0–100	Combined with time to compute urgency.
`contextualFactor`	0–100	Context input passed to `computePriority`.
`deadLine`	Unix timestamp	Compared with `block.timestamp`.
`weatherFactor`	5, 10, 20, 30	Oracle/local-test state used in priority.
`w1`–`w5`	Sum = 100	On-chain weights for the final score.

Table 5.2: Input and state semantics for the priority calculation.

Unless otherwise specified, the task inputs are provided through the dashboard or by the administrator during the prototype tests.

The function remains deterministic and transparent, in fact, only with respect to the same inputs, the same relevant contract state, and the same execution context it returns the same priority score. In particular, the value of `block.timestamp`, the stored weights, and the stored `weatherFactor` are part of the context that can affect the result, making the rule suitable for smart contract execution, since the decision process is visible, reproducible and auditable.

5.5.10 Oracle Functions

`TaskPriority Contract` also includes functions related to Chainlink CRE, such as `onReport`, `updateWeatherFromCRE`, `updateWeatherForLocalTest`, and `decodeWorkflowId` that are necessary to receive and validate weather updates from the CRE workflow.

They are not described in detail here because they belong to the oracle integration part of the thesis. For now, it is enough to clarify their role: they update the environmental factor used by the priority formula. The contract does not ask the weather API directly, instead, the off-chain CRE workflow reads Open Meteo, prepares the report, and

writes the result to the contract. Afterwards, the contract verifies that the caller is the authorized forwarder and updates `weatherFactor` as shown in chapter 6.

5.5.11 Events

The contract emits events when important state changes occur. In particular:

- `WeightsUpdated` records changes to the priority weights;
- `WeatherUpdated` records the temperature, weather factor, and workflow identifier;
- `ForwarderUpdated` records changes to the authorized CRE forwarder;
- `ExpectedWorkflowIdUpdated` records the workflow identifier expected by the contract.

Overall, `Task Priority` is the contract that translates contextual information into a priority score, its value is not only that it computes a number, but that it makes the computation explicit. The priority rule is written in Solidity, the weights are visible on-chain, the health score is computed by a public function, and the weather factor is updated through a controlled oracle path. All of this makes the task ordering logic compatible with the verifiability goals of the thesis.

5.6 TaskValidation

The role of the last contract of the architecture is to decide whether a cleaning task can actually be accepted and validated by the system. While `TaskPriority` computes how important a task is, `TaskValidation` checks whether the task can be executed in practice, acting as the final gate of the smart contract system.

This contract does not duplicate the logic already implemented in the previous contracts, instead it communicates with them through Solidity interfaces. It reads user information from `UserContract`, checks time preferences through `PreferenceContract`, and calls `TaskPriority` to obtain the priority score, through the following interfaces:

`IUserContract, IPreferenceContract and PriorityContract`

The first one is used to read the role, reputation, active status, and existence flag of a user, the second one is used to verify whether a time slot is compatible with the student's preferences and finally, the third one is used to call the priority function defined in `TaskPriority`. In this way, the validation process remains modular, where each contract has a specific responsibility, while the final decision is made by combining their outputs.

In this prototype, this contract records successful task validations or simulated completions inside the smart contract workflow but it is not independent evidence that a physical robot actually cleaned the space.

5.6.1 Student and Robot Resources

`TaskValidation` defines two main data structures, the first one is `StudentResource`:

```
struct StudentResource {
    uint256 gasBalance;
    uint256 paymasterUses;
    uint256 lastReset;
}
```

This structure stores the resources associated with a student inside the validation contract. The field `gasBalance` represents an internal balance that can be used when the paymaster mechanism is not available. The field `paymasterUses` records how many times the student has used the sponsored mechanism, while `lastReset` stores the timestamp of the last weekly reset.

The mapping:

```
mapping(address => StudentResource) private students
```

associates each student address with its internal resource data. The mapping is marked private at Solidity level, but this does not provide confidentiality on a public blockchain, it only restricts direct access from other contracts through the Solidity interface. The prototype therefore treats these values as synthetic testing data and avoids storing real sensitive student information on-chain. The contract provides the function `getStudent` to allow administrators to read this information in a controlled way. This is coherent with the access control logic described in Section 5.3: user information should not be modified or exposed without authorization.

The second structure is `RobotResource` and represents the current operational state of the robot.

```
struct RobotResource {
    uint256 battery;
    uint256 availableTime;
}
```

The field `battery` models the remaining battery or execution capacity, while `availableTime` represents the time still available to perform tasks. In the current prototype, the system considers one robot only, so the contract stores a single public variable:

```
RobotResource public robot
```

This choice keeps the prototype simple, but a future version could extend the same idea to a mapping of robots, where each robot address is associated with its own battery and available time. However, for the purpose of this thesis, one robot is sufficient to show how task validation can be represented through explicit smart contract conditions.

5.6.2 Administrative Control

Some functions in `TaskValidation` can only be called by an administrator through the use of `onlyAdmin` modifier, which reads the caller from `UserContract` and checks three conditions: the caller must exist, must be active, and must have the admin role.

This is necessary because the contract contains functions that update important resources; for example, `setRobot` updates the robot battery and available time, while `setStudentGas` assigns an internal balance to a student. If these functions were public without restrictions, any address could artificially increase its own resources or modify the robot state. Both functions emit events, making resource updates observable from outside the contract and supporting the transparency objective of the system.

5.6.3 Student, Robot, and Preference Checks

Before a task can be accepted, the contract checks whether the robot, the student, and the student's preferences are valid.

First, the function `validateRobot` checks whether the robot has enough resources; in the current version, the robot is considered available only if its battery is above a minimum threshold and if it still has available time. More precise battery models could be introduced in future versions, but the current implementation is sufficient to show how physical constraints can be translated into smart contract checks.

Then the function `validateStudent` verifies whether the student is allowed to receive a validated task, and checks that the student exists in the system, is active, has the student role, and has sufficient reputation. In the current prototype, role 0 corresponds to a student, while role 1 corresponds to an administrator. The reputation requirement is a governance policy of the prototype that controls participation in two places: a student with insufficient reputation cannot submit preferences and cannot be associated with a validated task. This requirement is used to model trust and abuse prevention in a closed residence system, but it is not a neutral measurement of the student's needs.

Finally, the function `checkPreferences` verifies whether the task respects the student's time preferences. The current hour is computed from `block.timestamp`, and the contract then asks `PreferenceContract` whether that hour is acceptable for the student, keeping, in this way, preferences separate from the priority score: a task may be important, but it should still be rejected if it violates the student's allowed time slots.

5.6.4 Distance, Execution Time, and Operational Cost

The validation contract also models the physical cost of executing a task, considering the time required to complete the task and the distance between the robot and the task location. Specifically, distance is not used as a priority factor by itself, instead, it contributes to the estimated execution time:

$$timeTask = baseTaskTime + distance \cdot timePerMeter$$

where `baseTaskTime` is the nominal time required to perform the cleaning task, `distance` represent the distance from the robot to the task location, and `timePerMeter` approximates the travel time required for each unit of distance. The estimated time is then used to calculate an abstract operational cost:

$$requiredGas = timeTask \cdot 5 + sensorLevel \tag{5.13}$$

The corresponding Solidity source code is available in the project GitHub repository: https://github.com/giuliabodrito/Tesi-Giulia-Bodrito-Politecnico-di-Torino.

In this section, the name `requiredGas` must be interpreted carefully, in fact, it does not represent the real EVM gas paid to execute a blockchain transaction, but it is an internal operational cost variable used to model how expensive a task is for the robot. This cost depends on the estimated task time and on the sensor dirtiness level, while Ethereum gas depends on the computational and storage operations executed by the smart contract, as explained in Section 2.4.

The coefficient 5 present in 5.13 was selected to make execution time the dominant component of the operational cost, while still considering the sensor of dirtiness level. In a real deployment, this coefficient can be changed using empirical measurements of the robot's battery consumption and task duration.

During validation, the contract checks two constraints:

$$robot.availableTime \geq timeTask$$

$$robot.battery \geq requiredGas$$

If one of these two conditions is false, the task cannot be accepted, even if the task has high priority. This is important because a decision is not only evaluated ethically or logically, but also operationally: the robot must actually be able to perform it.

5.6.5 Paymaster Access Logic

The contract includes a simplified mechanism inspired by the Paymaster concept in the ERC-4337 standard [24] which introduces account abstraction in Ethereum. In traditional Ethereum, a user normally sends a transaction from an externally owned account and pays the gas fee directly but with account abstraction, instead, the user can interact through a smart account, and the payment logic can be made more flexible.

In the ERC-4337 model, the user creates a `UserOperation`, which describes the action to be executed and the related gas-payment information; this operation is processed by a central smart contract called `EntryPoint`. A Paymaster can then decide to sponsor the transaction fee under predefined conditions, so that the user does not have to pay the gas directly.

This thesis does not implement the full ERC-4337 architecture, indeed there is no complete `EntryPoint` flow and no real account-abstraction wallet, instead, the prototype only adopts the idea of a controlled sponsoring policy. In this simplified version, a registered student can use the paymaster mechanism only a limited number of times within a weekly period.

The maximum number of sponsored uses is defined by:

```
uint256 public constant MAX_PAYMASTER_USES = 4;
```

The value `4` means that each student can use the sponsored mechanism up to four times in one week, limiting how many eligible task validations can be covered by the system during the weekly period, and does not restrict the number of preferences that

can be submitted. After the weekly limit is reached, the student can still request a task, but the contract checks whether the student has enough internal balance. Indeed once the current block timestamp is greater than `lastReset + 7 days`, the next eligible validation resets the counter and starts a new weekly period.

The function `canUsePaymaster` verifies that the student is registered, active, has the student role, and has not exceeded the weekly limit. If these conditions are satisfied, the internal function `usePaymaster` updates the counter, and if one week has passed since the last reset, the counter is reset before being incremented again. In particular the paymaster-inspired mechanism implemented in this prototype does not pay real Ethereum transaction fees. During the experiments, blockchain transaction fees are paid by the account that sends the transaction, typically the administrator account in Remix or the configured test wallet on Sepolia. The contract instead models an internal sponsorship policy: a student can execute a limited number of validated tasks without decreasing the internal `gasBalance`. After the weekly quota is exhausted, the task can still be validated only if the student has enough internal balance; otherwise, the transaction reverts.

In a real deployment, the sponsored cost could be covered by the organization that manages the service, such as the student residence, the university, or the building administrator. In this way, the mechanism supports accessibility policy: students are not excluded from task execution only because they cannot cover the internal cost at that moment. At the same time, the weekly limit prevents unlimited use and reduces the risk of abuse.

5.6.6 Grouped Input Parameters

Since the function `validateTask` receives many inputs, in order to keep the function manageable, the parameters are grouped into arrays.

The array `taskData` contains the parameters used to estimate the task duration:

$$\texttt{taskData} = [\texttt{baseTaskTime}, \texttt{distance}, \texttt{timePerMeter}]$$

The array `priorityData` contains the numerical inputs sent to the priority contract 5.5.9

$$\begin{aligned}\texttt{priorityData} = [&\texttt{sensorLevel},\ \texttt{daysSinceCleaning},\ \texttt{environmentalRisk},\\ &\texttt{eventPriority},\ \texttt{contextualFactor},\ \texttt{deadLine}]\end{aligned}$$

and the array `healthData` contains the health boolean values:

$$\texttt{healthData} = [\texttt{asthma}, \texttt{allergies}, \texttt{flu}]$$

This grouping is also useful from a Solidity point of view, in fact, as already explained in 5.5.9, a previous version of the function that used many separate parameters and local variables lead to the Solidity *stack too deep* error. The EVM uses indeed a stack as temporary execution space, and Solidity has limits on how many local values can be handled at the same time, while grouping related inputs reduces the number of direct parameters and makes the interface easier to use from Remix.

5.6.7 Validation Pipeline

The main function of the contract, `validateTask`, applies a sequence of checks, and the task is accepted only if all of them pass.

First, the function checks that the student is valid and that the student's preferences are satisfied. If these initial conditions pass, the function calls `TaskPriority` and requires the resulting priority score to be greater than or equal to `PRIORITY_THRESHOLD`.

The variable `PRIORITY_THRESHOLD` represents the minimum score required for accepting a task. In the current prototype, it is initialized to 4000 and can be updated by the administrator:

```
uint256 public PRIORITY_THRESHOLD = 4000;

function setPriorityThreshold(uint256 newThreshold) public onlyAdmin {
    PRIORITY_THRESHOLD = newThreshold;
}
```

This threshold works as a configurable filter and prevents the system from accepting tasks that are valid in principle but not important enough according to the current priority model. Since the priority score is a non-normalized weighted value, the threshold must be chosen according to the scale produced by `TaskPriority`. Making the threshold configurable allows the prototype to test both passing and failing priority cases without redeploying the validation contract.

Once the priority condition is satisfied, the function computes `timeTask` and `requiredGas` and the contract verifies that the robot has enough available time and battery. Finally, it applies the paymaster rule: if the student can use the paymaster, the counter is updated; otherwise, the student must have enough internal balance, and the required amount is deducted.

After a successful validation, the robot resources are updated:

$$robot.availableTime = robot.availableTime - timeTask$$

$$robot.battery = robot.battery - requiredGas$$

5.6.8 Events

The contract then emits the `TaskValidated` event that records who the task is associated with, whether the paymaster mechanism was used, the estimated task time, and the required operational cost.

Other events are used to make intermediate updates observable: `PriorityChecked` records the priority score returned by `TaskPriority`, while `PreferenceChecked` records whether the student's time preferences were satisfied. The events `RobotUpdated` and `StudentGasUpdated` record changes to the robot resources and to the student's internal balance.

These events do not execute the physical task by themselves, but creates an trace of the validation process, in this way external applications, dashboards, or monitoring tools can observe the events and reconstruct why a task was accepted.

In particular, the events emitted by `TaskValidation` provide an audit trail only for successful transactions that complete without reverting. If a validation condition fails, the function reverts and state changes and logs emitted during that execution are rolled back [16]. In that case, the evidence of rejection is not a persistent contract event, but the failed transaction receipt, the revert reason shown by the execution environment, and any off-chain dashboard preview or simulation performed before submitting the transaction.

Therefore, the current prototype records successful validations on-chain through events and state updates, while rejected tasks are mainly observable through transaction failure information and off-chain testing outputs. A future version could implement non-reverting rejection records if persistent on-chain rejection logs were required.

5.6.9 Role in the Architecture

Overall, `TaskValidation` is the contract that connects the different parts of the architecture: it does not decide user identity by itself, because this information comes from `UserContract`, neither does store preferences directly, since they are managed by `PreferenceContract` and does not even define the priority formula, because this is handled by `TaskPriority`. The TaskValidation's responsibility is to combine these elements and decide whether a task can pass the final validation step.

From the contractualist perspective introduced in Section 3.1, this contract is important because it transforms the idea of justifiable action into concrete checks. A task is indeed accepted only if the student is valid, the student's preferences are respected, the task has sufficient priority, and the robot has enough resources to execute it. Then the final decision is recorded through events, making the process verifiable.

5.7 Summary of the Smart Contract flow

In summary, this chapter described the implementation environment and the four smart contracts that work together to implement the validation flow of the proposed system. The contracts were developed in Solidity and tested in Remix, which provides a simple environment for compiling, deploying, and interacting with smart contracts. During the first development phase, Remix VM and local testing environments are useful because they allow fast experimentation without using a public blockchain. Then, as explained in the next chapter, the system is moved to Sepolia in order to connect the smart contracts with the oracle component.

The smart contract process starts with the registration of users in `UserContract` through their blockchain address, while administrators are allowed to manage users and update the system state, creating the access control layer of the architecture.

After registration, students can express preferences about cleaning time slots through `PreferenceContract` which are not used to compute priority directly, but they are used later as validation constraints: even if a task has high priority, it should not be accepted if it violates the student's preferences.

The `TaskPriority` contract combines several factors, including dirtiness, health needs, urgency, contextual information, and weather conditions, to compute the priority of a task. The weather factor is stored on-chain, but it depends on external data; for this

reason, the contract also includes functions designed to receive updates from a Chainlink CRE workflow.

Finally, `TaskValidation` combines the outputs of the other contracts, checks that the student is valid, the preferences are satisfied, the task priority is high enough, and the robot has sufficient battery and available time. `TaskValidation` also applies the paymaster access logic, so that a limited number of task validations can be sponsored under explicit conditions. Then, if all checks are satisfied, the task is validated and the result is recorded through an event.

The complete flow can therefore be summarized as follows:



This flow shows how the contractualist idea introduced in the previous chapters is translated into a concrete smart contract architecture where the system does not rely on an opaque decision process: each relevant condition is encoded, checked, and recorded. At the same time, one important part of the model still depends on information that is not naturally available on-chain, namely external contextual data such as weather conditions. The next chapter therefore focuses on the oracle problem and on the integration of Chainlink CRE and Open Meteo to update the weather factor used by the priority contract.

Chapter 6

Oracle Integration with Chainlink CRE

The previous chapter described the smart contract implementation and showed how `UserContract`, `PreferenceContract`, `TaskPriority`, and `TaskValidation` work together to validate a cleaning task. The validation process is mostly based on information that can be stored or checked directly on-chain, such as users, roles, preferences, priority parameters, and robot resources.

However, one part of the priority model depends on information that is not naturally available inside the blockchain: the weather conditions. As explained in Section 5.5.8, the `weatherFactor` is one of the components used by `TaskPriority` to compute the final priority score. If it is raining or if the temperature is high, a cleaning task may become more urgent, especially in shared spaces where external conditions can affect dirtiness, comfort, or hygiene.

However a smart contract cannot directly call an external API, read a website, or access real-time weather data by itself beacause smart contracts are deterministic programs: every node executing the contract must obtain the same result from the same on-chain state. External data, instead, belong to a non-deterministic environment and may change over time, depend on the provider, or become temporarily unavailable.

For this reason, an oracle component is needed; specifically, the oracle is used to connect the off-chain weather source with the on-chain priority contract. In this prototype, a Chainlink CRE workflow reads weather information from Open Meteo, extracts the temperature and weather code, and sends the result to the `TaskPriority` contract deployed on Sepolia. The smart contract then updates the `weatherFactor`, which becomes part of the priority computation used later by `TaskValidation`.

Therefore the purpose of this chapter is to explain the integration between the smart contract and the oracle component: first, the oracle problem is introduced, showing why external data represent a challenge for blockchain systems. Then, the chapter explains why Chainlink CRE was selected as the integration layer, how Open Meteo is used as a weather data source, how the weather factor is computed, and how the TypeScript workflow communicates with the Solidity contract deployed on Sepolia.

In this architecture, weather is treated as a simple example of a more general external-context interface: the objective is not to build a complete weather oracle, but to show how an external contextual variable can be introduced into the priority model through a controlled workflow. For this reason, the chapter also separates two different aspects: the integrity of the delivery path and the accuracy of the original weather observation.

6.1 The Oracle Problem

Blockchain systems are designed to provide a shared and verifiable state and once a transaction is accepted by the network, the resulting state transition can be checked by all participants. This property is useful for the thesis because the validation rules of the robotic system become explicit, reproducible, and difficult to modify without authorization.

Nevertheless, as discussed in oracle literature 2.6 the use of an oracle does not remove the external data source: an oracle can provide a structured path for delivering data to a smart contract, but the data may still be inaccurate, delayed or unavailable [2, 20]. This is the core of the oracle problem.

Figure 6.1 summarizes this problem in the specific case of the weather-factor integration. The figure shows the main path followed by the external data before it becomes part of the priority model.

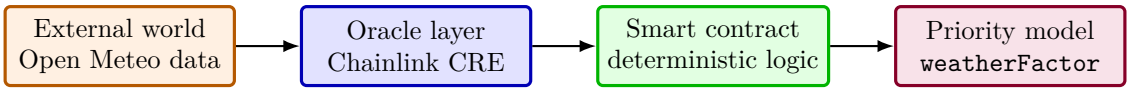


Figure 6.1: Oracle problem in the weather-factor integration.

In the context of this thesis, the oracle problem is directly connected to the task-priority model, which uses the weather factor as one of the weighted inputs of the function `computePriority` 5.12. Therefore, an incorrect weather update can affect the ordering and validation of cleaning tasks, for example, if the system receives an incorrect rain condition, the weather factor may become higher than necessary and increase the priority of some tasks. On the other hand, if rain is not detected when it is actually occurring, the system may underestimate the urgency of cleaning shared spaces.

This does not mean that oracles make the system unreliable; rather, it means that the oracle layer must be designed carefully. The smart contract can enforce transparent and deterministic rules, but it still needs a trustworthy mechanism to receive external data. In this project, Chainlink CRE is used to reduce this gap between off-chain information and on-chain execution. CRE does not eliminate every possible risk related to external data, but it provides a structured way to execute off-chain operations and deliver their result to a smart contract.

6.2 Why Chainlink CRE

Chainlink is a decentralized oracle infrastructure that connects smart contracts with data and computation coming from outside the blockchain [5] . As introduced at the end of the previous section, the selected Chainlink component used in this thesis is Chainlink Runtime Environment (CRE), an orchestration environment for workflows [9] that combine off-chain operations, such as HTTP requests, with on-chain interactions, such as EVM reads and writes [12].

In this thesis, three levels must be distinguished: Chainlink CRE is the infrastructure used to define and execute workflows. The workflow is the specific TypeScript program developed in the prototype to read Open Meteo data and prepare the contract update. The experiment used a local CRE workflow simulation with real Sepolia broadcast, therefore, the Sepolia transaction proves that the workflow logic could read Open Meteo data and write the resulting values to the deployed `TaskPriority` contract, but it does not by itself prove execution as a deployed workflow across a decentralized oracle network.

This structure is useful for the proposed system because the weather update is not only a simple variable assignment: the system has to read the current weather from an external API, extract only the values that are relevant for the priority model, compute or verify the corresponding weather factor, and then send the result to `TaskPriority`. CRE allows these operations to be expressed within a single workflow written in TypeScript, with the final result delivered to the Solidity contract on Sepolia [8, 11]

The role of CRE in this project can therefore be summarized as follows:

- it connects off-chain data with the on-chain smart contract;
- it uses an HTTP capability to request weather data from Open Meteo;
- it uses an EVM capability to write the result to the contract deployed on Sepolia;
- it allows the workflow to be tested and simulated before being used as part of the final integration;
- it provides a more structured architecture than the earlier prototype based on a local Python script or on a simpler oracle model

This last point is important for the development of the thesis, in fact, in the first versions of the project, the weather update was tested with a local Python mechanism, using Ganache as the local blockchain environment. The script could read weather information from an external API, transform the value into a weather factor, and then send a transaction to the smart contract or support manual testing in Remix. This approach was useful during development because it made it possible to test the priority formula before the oracle infrastructure was complete. It also helped verify that the contract could receive an external value and include it in `computePriority`.

However, the Python prototype was not sufficient for the final architecture: the main limitation was not the programming language itself, but the role of the script in the system. The update depended on the local computer, on the correct execution of a developer-controlled program, and in some versions on manually configured API keys.

From the blockchain point of view, this made the update less reproducible and less structured, because the off-chain logic was external to the oracle infrastructure.

Compared with the earlier Python prototype, the CRE version improves the structure of the integration because the trigger, HTTP request, data processing, EVM write, and configuration are expressed in an explicit workflow. However, in the reported experiment this improvement concerns workflow structure and Sepolia integration, not a demonstrated distributed fault-tolerance property.

Figure 6.2 summarizes the difference between the earlier Python prototype and the CRE integration.

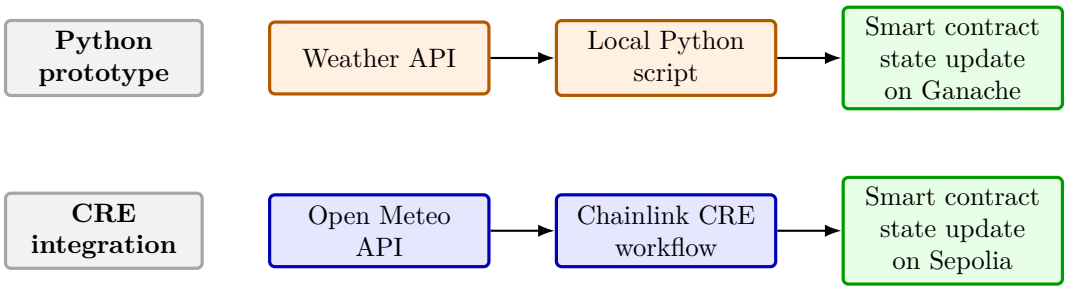


Figure 6.2: Comparison between the Python prototype and the Chainlink CRE integration.

In the Python prototype, the bridge is a local script controlled by the developer while in the CRE integration, the bridge is an explicit oracle workflow that connects the weather source at the priority logic introduced in Section 5.5.8.

An operational prerequisite of the implemented prototype is access to the Chainlink CRE environment. The workflow is executed through the CRE CLI 6.10, which requires authentication and access to the corresponding CRE services. Therefore, reproducing the complete oracle integration requires not only the workflow source code and the deployed smart contract, but also a properly configured CRE environment. This requirement concerns the execution infrastructure rather than the Solidity logic itself: the smart contracts deployed on Sepolia remain independently accessible and verifiable, while the CRE environment is required to execute the off-chain workflow that retrieves the external weather data and delivers the corresponding report on-chain.

6.3 Open Meteo as External Data Source

Open Meteo, the external data source used in this thesis, provides weather information through an HTTP API. In the implemented workflow, the API is queried to obtain the current weather conditions for a selected location; in this case the prototype uses Barcelona as the test location, but the same structure could be adapted to another city by changing the latitude, longitude, and timezone in the configuration file.

The main reason for using Open Meteo is that it provides a simple and accessible weather API for the type of data needed in this project. In particular, the workflow only requires two values:

- `temperature_2m`, which represents the air temperature at two meters above the ground;
- `weather_code`, which represents the current weather condition using WMO weather interpretation codes.

These two values are enough for the weather factor used by `TaskPriority`: the temperature is used to detect warm conditions, while the weather code is used to identify rain and other relevant weather states. Since the smart contract works with integer values[1] , the TypeScript workflow rounds the temperature before sending it to Solidity.

The weather code is a compact numerical representation of the current weather condition, Table 6.1 reports the main Open Meteo weather codes [28] used in the API response.

Table 6.1: Open Meteo weather codes based on WMO weather interpretation codes.

Code	Weather condition
0	Clear sky
1, 2, 3	Mainly clear, partly cloudy, and overcast
45, 48	Fog and depositing rime fog
51, 53, 55	Drizzle: light, moderate, and dense intensity
56, 57	Freezing drizzle: light and dense intensity
61, 63, 65	Rain: slight, moderate, and heavy intensity
66, 67	Freezing rain: light and heavy intensity
71, 73, 75	Snowfall: slight, moderate, and heavy intensity
77	Snow grains
80, 81, 82	Rain showers: slight, moderate, and violent
85, 86	Snow showers: slight and heavy
95	Thunderstorm: slight or moderate
96, 99	Thunderstorm with slight and heavy hail

In the current prototype, the weather factor does not distinguish all these categories separately, but the contract uses a simplified rule: codes from `61` upward are treated as rain or severe weather conditions and receive the highest weather factor. This is a design simplification, because these codes include rain, freezing rain, snow, showers, and thunderstorm conditions. The purpose is not to build a complete meteorological model, but to include external weather conditions in the priority score in a simple and verifiable way.

In the implemented configuration, the Open Meteo endpoint is queried using the coordinates of Barcelona, namely `latitude=41.3874` and `longitude=2.1686`, and the request also specifies the parameters:
`current=temperature_2m,weather_code` and `timezone=Europe/Madrid`.

The workflow may receive a larger JSON response, as shown in Listing 6.1, but it only uses the two fields inside the `current` object: `temperature_2m` and `weather_code`.

[1]Solidity does not provide native floating-point arithmetic. For this reason, smart contracts usually represent decimal quantities through integers, fixed scaling factors, or off-chain preprocessing.

Listing 6.1: Example of the JSON response returned by Open Meteo for the Barcelona configuration.

```
{
  "latitude": 41.375,
  "longitude": 2.125,
  "timezone": "Europe/Madrid",
  "current": {
    "time": "2026-07-14T16:00",
    "temperature_2m": 29.3,
    "weather_code": 1
  }
}
```

Open Meteo was preferred to the earlier AccuWeather prototype [1] because the latter requires API-key management and the key must be protected as a secret. This is correct from a security point of view, but it made the first prototype more fragile because the execution depended on key configuration, free-plan limitations, and possible expiration problems. Open Meteo made the workflow easier to test because the required weather endpoint can be called directly by the CRE HTTP capability for the basic use case.

It is important to clarify that Open Meteo is still an external provider, therefore, using Open Meteo does not remove the oracle problem completely: if the API is unavailable or returns an incorrect value, the workflow may still receive wrong data. The role of Chainlink CRE is not to make the data source perfect, but to provide a structured and verifiable path for bringing that data into the smart contract.

6.4 Weather Factor Model

The weather factor is the numerical value that translates external weather conditions into the priority model. As already introduced in Section 5.5.8, the factor is not meant to represent a complete meteorological evaluation; instead, it gives the smart contract a simple way to increase the priority of cleaning tasks when the external conditions can make cleaning more urgent.

The rule used in the prototype is:

$$weatherFactor = \begin{cases} c_1 & \text{if } weatherCode \geq 61, \\ c_2 & \text{if } weatherCode < 61 \text{ and } temperature > 30, \\ c_3 & \text{if } weatherCode < 61 \text{ and } 20 < temperature \leq 30, \\ c_4 & \text{if } weatherCode < 61 \text{ and } temperature \leq 20. \end{cases} \tag{6.1}$$

Where the constants are set as $c_1 = 30$, $c_2 = 20$, $c_3 = 10$, and $c_4 = 5$.

If the weather code is greater than or equal to `61`, the system treats the condition as rain or severe weather. In that case, the weather factor becomes `30` reflecting the idea that rain can increase the need for cleaning in shared spaces, because people may bring water, mud, or more dirt into common areas.

If the weather code does not indicate rain or severe weather, the contract considers the temperature: a temperature higher than `30` degrees Celsius produces a weather factor of `20`, while a temperature higher than `20` degrees Celsius produces a weather factor of `10`. In all other cases, the factor remains at the default value `5`. Therefore, rain has a stronger effect than temperature, while warm conditions still have a moderate effect on the priority score. Is important to notice that this values were selected to keep the model simple and readable and are not the result of a metereological calibration. The objective is indeed to create separate level of environmental influence that can be easily inspected on-chain and understood in the priority formula. Open Meteo current conditions are based on weather-model data, not necessarily on a direct local sensor observation [28]. Therefore, the prototype should treat the API response as external model-based weather data. Afterwards the workflow sends the rounded temperature and the weather code to the smart contract. After receiving these values, `TaskPriority` applies the rule defined in Equation 6.1 and updates `weatherFactor` on-chain.

For tasks evaluated under the same stored `weatherFactor` and the same weights, the weather term is identical for all tasks. If B_i and B_j denote the non-weather parts of two task scores, then:

$$P_i = B_i + w_5 E, \qquad P_j = B_j + w_5 E$$

and therefore:

$$P_i - P_j = B_i - B_j.$$

Thus, under these assumptions, weather changes the absolute score and can affect threshold decisions, but it does not change the relative ordering between tasks. Although weather still remains useful in the prototype because it demonstrates how external context can enter the priority model through the oracle path.

6.5 CRE Workflow Architecture

The previous sections introduced the oracle problem, Open Meteo as the data source, and the rule used to compute the weather factor. The next step is to clarify where the CRE workflow is placed in the system; in particular, the workflow should not be understood as a replacement for the smart contracts. CRE, is indeed only the off-chain component that collects the weather data and sends the relevant values to the priority contract, whereas `TaskPriority` remains the contract that stores the weather state and computes the priority score, and `TaskValidation` remains the contract that validates whether a task can be accepted, as described in Chapter 5. Chainlink CRE is placed before these contracts: it updates the external contextual input that `TaskPriority` will later use.

Figure 6.3 summarizes the complete path and the two inputs needed by the CRE workflow. The dashed areas show the boundary between the off-chain and on-chain parts of the integration. Open Meteo and the CRE workflow operate off-chain, where HTTP requests and external data processing are possible. The Solidity contracts operate on-chain, where the received values are stored and used by the priority logic.

The workflow configuration is part of the architecture because the workflow needs to know where to read data from and where to write the result. In the prototype, this

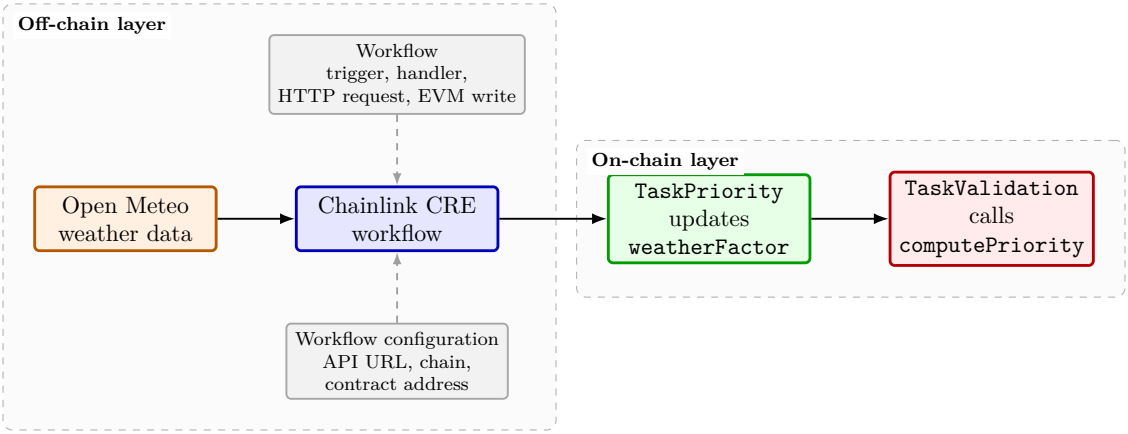


Figure 6.3: Architecture of the CRE weather update, separated into off-chain and on-chain components.

configuration is stored in the file `config.staging.json`, shown in Figure 6.4: the file contains the Open Meteo URL, the location name, the Sepolia chain selector, the gas limit, the `writeOnChain` flag, and the address of the deployed `TaskPriority` contract.

The field `taskPriorityAddress` is especially important: if `TaskPriority` is redeployed, this field must be replaced with the new contract address before running the workflow again. Otherwise, the workflow would write to a previous deployment instead of the current contract used by `TaskValidation`.

```
{
  "schedule": "*/30 * * * * *",
  "locationName": "Barcelona",
  "openMeteoUrl": "https://api.open-meteo.com/v1/forecast?latitude=41.3874&longitude=2.1686&current=temperature_2m,weather_code&timezone=Europe%2FMadrid",
  "chainSelectorName": "ethereum-testnet-sepolia",
  "taskPriorityAddress": "0x7373fF0F124AED4891Fb4B30bA6ba417cDF1EC5e",
  "gasLimit": "1000000",
  "writeOnChain": true
}
```

Figure 6.4: Configuration file used by the Chainlink CRE workflow

The field `writeOnChain` acts as a safety switch in the workflow configuration: when it is set to `false`, the workflow reads the Open Meteo data and computes the expected values, but it does not send any transaction to Sepolia. When it is set to `true`, the workflow calls the on-chain writing function and sends the rounded temperature and weather code to the configured `TaskPriority` contract. This separation was useful during development because the data-reading logic could be tested before enabling the actual on-chain update. The corresponding code is available in the github repository: https://github.com/giuliabodrito/Tesi-Giulia-Bodrito-Politecnico-di-Torino.

6.6 The TypeScript Workflow

As shown in Figure 6.3, the CRE workflow is implemented in the file `workflow.ts`, written in TypeScript and uses the Chainlink CRE SDK. In this context, the SDK provides the software tools needed to define the workflow and to use CRE capabilities. A capability

can be understood as a service made available to the workflow, for example a scheduled trigger, an HTTP request, or an EVM interaction.

The TypeScript workflow follows this sequence:

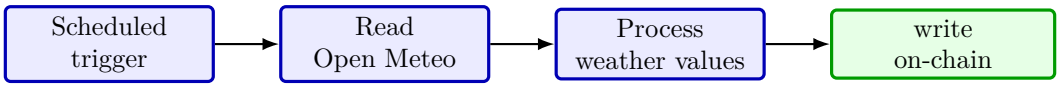


Figure 6.5: Sequence executed by the TypeScript CRE workflow.

The entry point of this sequence is `onCronTrigger`, defined as a constant that contains the handler executed when the scheduled trigger starts the workflow. In this context, a handler is the function that is called in response to a specific trigger and its role is to coordinate the following execution:

1. it checks that the scheduled execution time is present;
2. it calls `getWeatherInfo` to obtain the weather data;
3. it logs the location, rounded temperature, weather code, and expected weather factor;
4. if `writeOnChain` is enabled, it calls `writeWeatherOnChain`;
5. it returns a final message summarizing the weather update.

The scheduled execution is configured by `initWorkflow`: this function registers a `CronCapability` and connects it to `onCronTrigger`. In simple terms, the cron capability decides when the workflow starts, while `onCronTrigger` defines what the workflow does when it starts.

The weather-reading logic is divided into two parts: the function `fetchWeatherInfo` contains the request logic, it reads the Open Meteo URL from the configuration file, checks that the HTTP response is successful, parses the JSON response, and verifies that `current.temperature_2m` and `current.weather_code` are present. It then rounds the temperature and treats the weather code as an integer, because these are the values that can be sent to the Solidity contract.

The function `getWeatherInfo` connects the request logic to the CRE HTTP capability, creates an `HTTPClient`, and executes `fetchWeatherInfo` through the CRE environment.

In the general deployed CRE model, a workflow result may be produced by multiple oracle-node executions. Each execution can query the same Open Meteo endpoint and return the weather values used by the workflow. The aggregation step can then combine these results into a single value, as shown in Figure 6.6, where the number of nodes is illustrative.

In the experiment reported in this thesis, however, the workflow was executed with the CRE CLI in local simulation mode using the broadcast option. Therefore, the figure should be read as a conceptual representation of the deployed CRE model, not as direct evidence of the number of oracle nodes involved in the experiment.

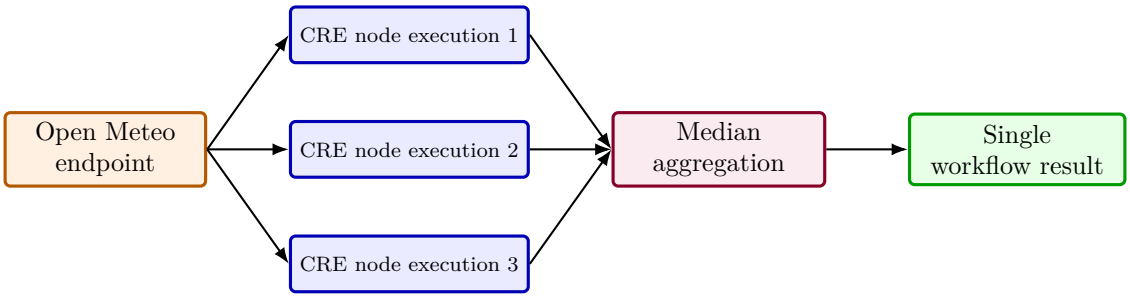


Figure 6.6: Conceptual aggregation of weather values in the deployed CRE model. The number of nodes is illustrative and does not represent the local simulation mode used in the experiment.

In the implemented workflow, the median is used for the rounded temperature and, as an implementation simplification, for the weather code. The median of temperature is meaningful as a numerical aggregation rule, however, the median of `weather_code`, should not be interpreted as a meteorologically meaningful aggregation, because weather codes are categorical labels rather than continuous meteorological quantities. A production version should use an aggregation rule designed for categorical weather states. This aggregation only concerns the workflow execution and should not be confused with the accuracy of the original weather source: Open Meteo remains the provider of the external observation.

The on-chain part is handled by the function: `writeWeatherOnChain` which selects the Sepolia EVM network from the configured chain selector, creates an `EVMClient`, and creates a TypeScript object representing the deployed `TaskPriority` contract. To send the update on-chain, the workflow uses the TypeScript binding generated from the contract ABI. This binding provides a helper method that prepares the CRE report and encodes the call to `updateWeatherFromCRE(int256,uint256)`. The encoded report contains the two values that must be delivered to the contract: the rounded temperature and the weather code. After receiving them, `TaskPriority` applies the rule in Equation 6.1 and updates `weatherFactor` on-chain.

6.7 The Smart Contract Receiver

The receiver of the CRE update is the `TaskPriority` contract introduced in Chapter 5 as the contract that stores the priority-related state. In this section, the focus is only on the part of the contract that receives weather values from the CRE workflow.

The update reaches the contract through the `onReport` function, that is the generic entry point used to receive a CRE report. In simple terms, it receives two kinds of information: information about the workflow execution and the encoded data prepared by the workflow. In this prototype, the encoded data contains the rounded temperature and the weather code.

Since `weatherFactor` influences the priority score, the contract must not accept

weather updates from any address. For this reason, `onReport` is protected by the **modifier** `onlyForwarder`, where the `forwarder` is the authorized delivery address allowed to submit CRE reports [7] to `TaskPriority` and if the caller is not this address, the update is rejected.

Figure 6.7 summarizes the receiver logic before the individual checks are described in more detail.

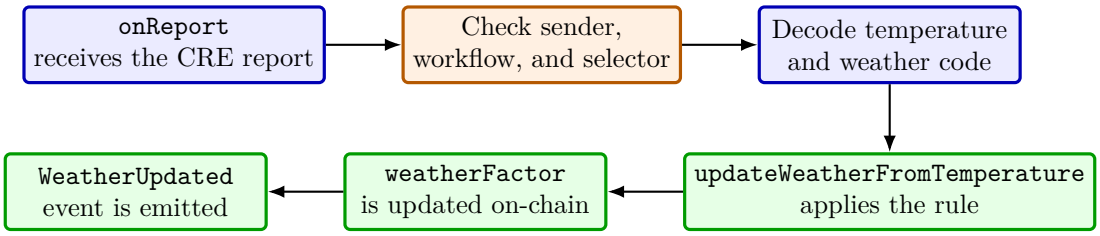


Figure 6.7: Internal receiver flow used by `TaskPriority` to process a CRE weather report.

After checking the sender, `onReport` verifies the workflow information: the workflow identifier indicates which CRE workflow produced the report. In the contract, this value can be compared with `expectedWorkflowId`, even though in the reported experiment, `expectedWorkflowId` was left equal to zero. Enabling a non-zero `expectedWorkflowId` would further restrict accepted reports to a specific workflow and is left as a configuration improvement for a future deployment.

The contract also checks the function selector contained in the encoded report, in Ethereum, a function call is encoded as calldata, and the first four bytes of this calldata identify the function being called. In this prototype, these first four bytes must correspond to `updateWeatherFromCRE`, ensuring that the report is intended to perform the weather update, and not another contract operation.

If these checks are successful, `onReport` decodes the weather values from the report: `temperature` and `weatherCode`. The contract then calls the internal function `updateWeatherFromTemperature`, which stores `lastUpdatedTemperature`, applies the rule in Equation 6.1, updates `weatherFactor`, and emits the event `WeatherUpdated`.

The contract also contains `updateWeatherFromCRE(int256,uint256)`, which represents the specific weather-update call prepared by the workflow and is also restricted by `onlyForwarder`. Finally, `updateWeatherForLocalTest` is not part of the CRE delivery path, but can be useful for manual testing in Remix, because it allows an admin to test the weather-factor logic without executing the CRE workflow.

6.8 Trust Chain and Data Handling

After the workflow and the receiver contract have been introduced, the trust chain can be described more precisely, the integration can be described in four levels:

1. **Source level**: Open Meteo is trusted as the provider of the original weather observation.

2. **Workflow level**: the CRE workflow is trusted to request the configured fields, check the response, round the values, and prepare the expected contract update.

3. **Delivery level**: the contract accepts updates only through the configured delivery address and, when enabled, the expected workflow identifier.

4. **Contract level**: `TaskPriority` does not simply accept a final weather factor from outside; it recalculates the factor on-chain from temperature and weather code.

These levels should not be confused: on one hand CRE improves the execution and delivery path, but it does not prove that the weather observation itself is true. If Open Meteo returns an incorrect value, the workflow may still process that value correctly and the contract may still update its state correctly. Moreover the workflow reads the current Open Meteo endpoint when it is executed, but the contract does not store an explicit observation timestamp and does not enforce a maximum age for the weather value. If the workflow stops running, or if new updates fail, `computePriority` continues to use the last `weatherFactor` stored on-chain. This is acceptable for a prototype, but a production version should store timestamps and reject stale values.

Failure handling is mostly implemented in the workflow, in fact, if Open Meteo returns an unsuccessful HTTP status, or if the expected fields are missing, or, moreover, the EVM write fails, the workflow throws an error. In these cases, the contract state is not updated, and the previous `weatherFactor` remains active.

6.9 Security and Trust Considerations

The security of this prototype is based on a simple principle: the off-chain workflow can collect data, but the on-chain contract must remain the place where the priority-relevant state is updated according to visible rules.

The first control is `onlyForwarder`, which restricts weather updates to the configured delivery address. The second control is `expectedWorkflowId`, which can restrict accepted reports to a specific workflow. The third control is the function selector check inside `onReport`, which verifies that the report corresponds to `updateWeatherFromCRE`. Together, these controls reduce the risk that an unrelated sender or unrelated report updates the priority contract, protecting the update path. However the controls do not validate the meteorological truth of the data. This is why the integration should be described as a preliminary oracle integration for external context, not as a complete production-grade weather oracle.

Future versions could query more than one weather provider, store the observation timestamp on-chain, reject stale updates, add nonce-based replay protection, or define different weather-factor rules for different buildings and locations.

6.10 CRE Access and Wallet Setup

Before executing the Chainlink CRE workflow, some preliminary configuration was required. In the experimental setup used for this thesis, access to the Chainlink environment

was associated with a blockchain wallet: MetaMask was used for this purpose and the wallet was connected to the Chainlink platform after accepting the corresponding Terms of Service.

The same wallet environment was also used to interact with the Ethereum Sepolia test network. Since the prototype was deployed and tested on Sepolia, test tokens were required to perform blockchain operations without using real ETH. The Chainlink faucet was therefore used to request testnet assets for the connected address.

It is important to distinguish this wallet configuration from the authorization mechanism used by the CRE weather update. Connecting a wallet to the Chainlink environment does not mean that this wallet directly sends every weather update to TaskPriority. The CRE workflow retrieves and processes the external weather data, while the smart contract accepts the resulting report through the configured CRE forwarder, protected by the onlyForwarder modifier.

In the prototype, the wallet used during the setup was also used for administrative and deployment operations on Sepolia. However, these are conceptually separate roles: the admin role is defined by the Solidity application logic, whereas the CRE forwarder is the authorized component responsible for delivering CRE reports to the weather receiver contract.

Therefore, the complete experimental setup requires three elements to be distinguished: access to the CRE execution environment, a wallet configured for interaction with the Sepolia test network, and the authorized CRE forwarder configured in TaskPriority

6.11 Sepolia Validation

The oracle integration was validated on Sepolia, the Ethereum test network used in the project. This experiment shows that a locally simulated CRE workflow can read external weather data and broadcast a real transaction to the deployed Sepolia contract.

The validation followed an incremental process: first, the workflow was tested without broadcasting a transaction. In this mode, it read the Open Meteo response for Barcelona, extracted the temperature and weather code, and computed the expected weather factor. This confirmed that the HTTP request and the TypeScript data-processing logic were working correctly.

Then, the on-chain write was enabled, the workflow sent the weather update to `TaskPriority` through the CRE EVM capability. A successful broadcast produced a Sepolia transaction hash, and the transaction receipt confirmed that the transaction was accepted. Afterwards, the contract getters showed that `lastUpdatedTemperature` and `weatherFactor` had been updated. In one successful validation run, the workflow logged Barcelona weather data with a rounded temperature of `30` degrees Celsius, weather code `3`, and weather factor `10`; the corresponding Sepolia transaction was `0xa9ff0d8062462bf52acb0f7cfd810bc1c073618d95d1cdd44feed313cb678ae9`. Therefore, the Sepolia transaction demonstrates that the workflow logic was able to read Open Meteo data, prepare the weather update, and write it to the deployed `TaskPriority` contract. However, by itself, this result does not demonstrate that the workflow was

deployed as a production workflow across a decentralized oracle network.

The target state changed by the transaction is limited and explicit, the contract updates `lastUpdatedTemperature` with the rounded temperature received from the workflow and updates `weatherFactor` according to the on-chain rule. These values are then available to the priority formula described in Chapter 5. The transaction does not directly validate a task; it only updates the contextual state that later influences task priority.

Figure 6.8 summarizes the validated path from the external weather source to the task-validation logic.

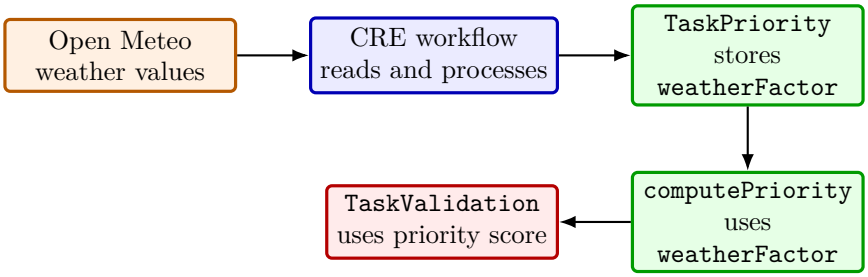


Figure 6.8: Validated Sepolia path from Open Meteo data to task validation.

This result connects the oracle chapter back to the previous smart contract chapter. If Chapter 5 explained the internal validation logic of the contracts, this chapter shows how one external variable, the weather condition, can enter that logic through a controlled oracle workflow. The final decision remains on-chain: Chainlink CRE collects and delivers the external context, `TaskPriority` stores the updated weather state, and `TaskValidation` can later use the resulting priority score. In the reported experiment, the CRE workflow was executed in local simulation mode with real Sepolia broadcast [10]. Therefore, the result validates the end-to-end path from Open Meteo reading to on-chain state update, but it does not demonstrate a production deployment across a decentralized oracle network.

Item	**Observed configuration**
CRE execution mode	Local workflow simulation with real Sepolia broadcast.
Command used	`cre workflow simulate ... -broadcast ...`
Workflow deployment status	Not deployed as a production decentralized workflow.
Target network	Ethereum Sepolia.
Forwarder configuration	The contract accepted reports only from the configured CRE forwarder address.
Observed result	The transaction updated `lastUpdatedTemperature` and `weatherFactor`.

Table 6.2: CRE execution mode and observed experimental configuration.

Chapter 7

Case Study

This case study evaluates the implemented prototype in the context of robotic cleaning-task validation in a student-residence scenario. The objective is not to demonstrate a complete autonomous cleaning system, but to show how a selected cleaning task can be checked through smart contracts, priority scoring, user preferences, robot-resource constraints, internal sponsorship rules, and oracle-updated weather state.

In the prototype, students are represented by blockchain addresses that act as simulated student identities, these addresses are registered on-chain by an administrator through `UserContract`. This allows the experiment to test user validation, preference checks, priority assessment, and task-validation outcomes without requiring a production identity-management system or a deployed physical robot.

7.1 Experimental Boundary and Account Roles

The case study uses the four deployed contracts: `UserContract` stores registered users, roles, activity status, reputation, and the task counter. `PreferenceContract` stores time preferences submitted by students. `TaskPriority` computes the priority score using the input components and the stored `weatherFactor`. Finally, `TaskValidation` combines user validity, preferences, priority threshold, robot resources, and the internal sponsorship mechanism before accepting or rejecting a selected task.

Different account roles are distinguished: administrative operations, such as user registration, robot-resource configuration, threshold updates, and task validation, are executed by an administrator account. Preference submission, instead, requires the caller to be an authorized student, because `addPreference` checks `msg.sender`. Therefore, when preferences are submitted during testing, the interaction must be executed from the corresponding simulated student account. The administrator can inspect stored preferences through the administrator-only getter, but this is different from submitting preferences on behalf of a student.

The Streamlit dashboard is used as a visualization and preview layer, it can read contract state, display user and robot information, compute local previews, and call view functions. Read-only calls, such as score previews or state inspection, may be executed as `eth_call` [15]; these calls do not submit a transaction and do not create

an on-chain audit record. Persistent on-chain records are produced only by successful transactions, such as preference submission, oracle weather updates, resource updates, or task validation. The dashboard source code is available in the project repository: https://github.com/giuliabodrito/Tesi-Giulia-Bodrito-Politecnico-di-Torino or at http://localhost:8502/.

7.2 Ranking and Selection Boundary

The current implementation does not include a fully on-chain priority queue; candidate tasks are indeed prepared off-chain, for example in the dashboard or by an external controller. Their priority scores can be computed by calling `TaskPriority`, and the resulting values can be displayed and sorted off-chain. However, the Solidity contracts do not store a persistent pending queue and do not enforce that only the highest-ranked task is submitted to validation.

For this reason, the case study should be interpreted as an evaluation of task validation and priority assessment, not as an evaluation of an on-chain queue-management system. The submitted task is selected off-chain and then passed to `TaskValidation`, which checks whether it satisfies the implemented validation rules. This boundary is consistent with the prototype status described in Chapter 4: a complete dynamic queue remains future work.

7.3 Evaluation Method

The case study evaluates the behaviour observed during prototype testing in Remix and through the Streamlit interface. The objective is not to provide a benchmark, but to verify whether the implemented validation rules accept and reject tasks consistently with the contract logic.

For each scenario, the relevant initial state, input values, expected outcome, and observed result are recorded. Successful cases are identified by accepted transactions or by the validation path shown in the interface. Failed cases are identified by reverted transactions, revert reasons, or by the rejection reason displayed in Streamlit. Events emitted inside reverted executions do not persist as ordinary on-chain logs. Table 7.1 summarizes the observed behaviour.

These observations are functional validation results, not a performance benchmark and not a complete formal test suite. They show that the implemented prototype distinguishes accepted and rejected cases according to the main validation conditions: user registration and activation, preferences, priority threshold, robot resources, internal gas balance, and paymaster-inspired weekly quota. The Streamlit interface also makes the reason for acceptance or rejection explicit, which supports the transparency objective of the prototype.

Moreover when multiple candidate tasks are considered, the prototype can rank them by their computed priority score before validation. This ranking step is performed off-chain, for example through the interface or supporting script logic. The current smart

Scenario	Condition tested	Observed behaviour
User state	Student registration, activation, and deactivation.	A registered and active student can be used in validation. After deactivation, the student is rejected by the validation logic.
Preferences	Satisfied preference, unsatisfied preference, or no stored preferences.	If the stored preference is satisfied, validation continues. If it is not satisfied, the transaction fails or Streamlit reports rejection. If no preference is stored, validation can proceed because the empty list is treated as no restriction.
Priority threshold	Computed priority score above or below `PRIORITY_THRESHOLD`.	Validation succeeds only when the score reaches the threshold. If the score is below the threshold, the transaction fails and Streamlit reports the priority-related rejection reason.
Robot resources	Battery and available time sufficient or insufficient for the task.	Validation can continue when both values are sufficient. If battery or available time is too low, the transaction fails.
Internal balance	Administrator assigns internal gas balance to the student.	The internal balance is stored and can be used as fallback when the paymaster-inspired quota is exhausted.
Paymaster-inspired path	Available quota, exhausted quota with balance, exhausted quota without balance, and weekly reset.	If quota is available, validation uses the sponsored path. If quota is exhausted but balance is sufficient, validation can still succeed through the fallback balance. If neither is available, validation fails. After one week, the paymaster-use counter is reset and the sponsored path becomes available again.
Streamlit preview	Validation parameters inspected through the Streamlit interface.	The interface shows whether the task is accepted or rejected and reports the reason, making the validation path easier to inspect.

Table 7.1: Observed behaviour of the prototype validation scenarios.

contract does not implement a dynamic on-chain queue: it validates the selected task submitted to `validateTask`.

7.4 Meaning of Task Completion

A successful call to `validateTask` represents a successful validation of the submitted cleaning task inside the smart contract workflow. It does not prove that a physical robot actually cleaned a room or common area since the prototype does not include independent physical sensing, navigation logs, or robot-side confirmation.

When validation succeeds, the contract updates the relevant internal state. Robot battery and available time are reduced according to the estimated operational requirement. If the paymaster-inspired quota is available, the corresponding internal counter is updated. If the quota is not available but the student has enough internal gas balance, that balance is used as a fallback. If neither condition is satisfied, validation fails.

Therefore, in this thesis, task completion should be interpreted as validation-level or simulated completion. It is evidence that the task passed the implemented contractual checks, not evidence of physical execution in the real world.

7.5 Worked Validation Trace

A complete validation trace can be reconstructed as follows: the administrator registers a student address and ensures that the student is active, has role `Student`, and has sufficient reputation. Then the robot state is configured with enough battery and available time. The associated student either has no stored time preferences, which the prototype interprets as no time restriction, or has a stored time preference matching the blockchain hour used by `checkPreferences`.

The selected task is then submitted to `validateTask` through three grouped inputs: `taskData`, `priorityData`, and `healthData`. `TaskValidation` first checks robot availability and student validity, then calls `PreferenceContract.isTimePreferred` for the associated student. If this check passes, the contract calls `TaskPriority.computePriority`. The resulting score is then compared with the current `PRIORITY_THRESHOLD`, if the score is high enough, the contract computes the estimated task time and internal operational cost, verifies robot resources, applies the sponsorship or fallback-balance rule, updates robot resources, increments `taskCompleted`, and emits `TaskValidated`.

The surviving record of a successful validation therefore consists of the transaction receipt, emitted events, updated robot state, updated sponsorship or internal-balance state, and the updated task counter. If one of the checks fails, the transaction reverts and the surviving evidence is the failed transaction information, the revert reason shown by the execution environment, and any off-chain preview made before submission.

In the successful validation trace, the student address was first registered and active, with the Student role and sufficient reputation. The preference check was satisfied for the tested time slot and the robot state before validation showed sufficient battery and available time. The priority inputs were then submitted to TaskPriority, including dirtiness-related values, health flags, urgency, contextual factor, and the stored weather

factor. The resulting priority score was compared with the configured threshold and, since the score was above the threshold and the robot had enough resources, TaskValidation accepted the task. The transaction updated the robot resource state, updated the internal sponsorship or balance state depending on the paymaster path, incremented the validation-related state, and emitted the TaskValidated event.

7.6 Summary

The case study shows that the prototype can make selected task validation inspectable through explicit smart contract checks. The priority score can support off-chain ranking and task assessment, while the final submitted task is validated on-chain against user, preference, priority, resource, and sponsorship conditions. The result is a bounded but concrete proof of concept: it does not implement a full on-chain queue or prove physical cleaning completion, but it demonstrates how the relevant validation state and successful outcomes can be recorded and inspected. The following screenshot provides a representative view of the auxiliary Streamlit dashboard used to inspect the case-study outputs. The dashboard contains additional sections for exploring task data, validation outcomes, and contextual variables; however, only one representative view is included here to avoid shifting the focus away from the smart-contract validation logic.

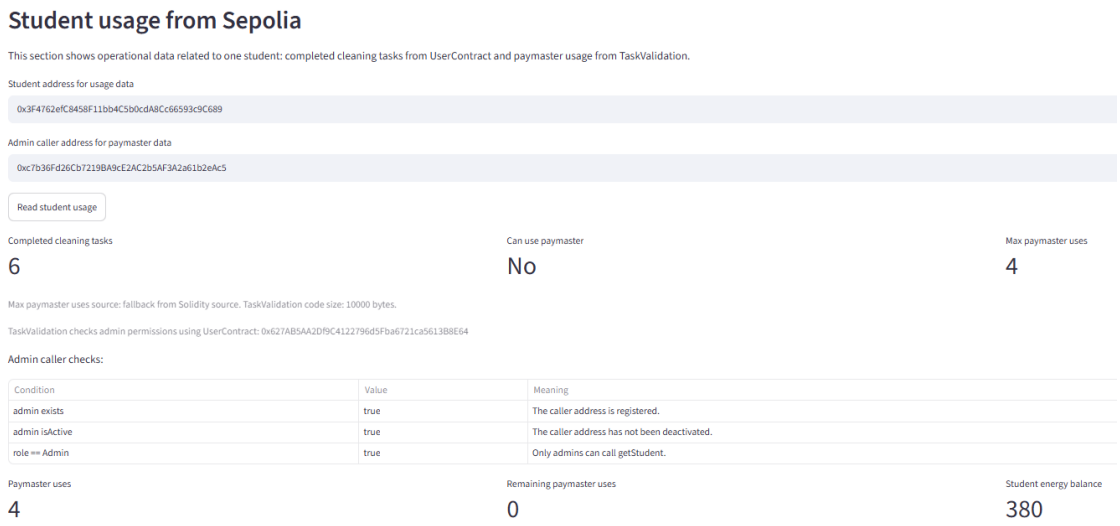


Figure 7.1: Validation visualization through Streamlit dashboard

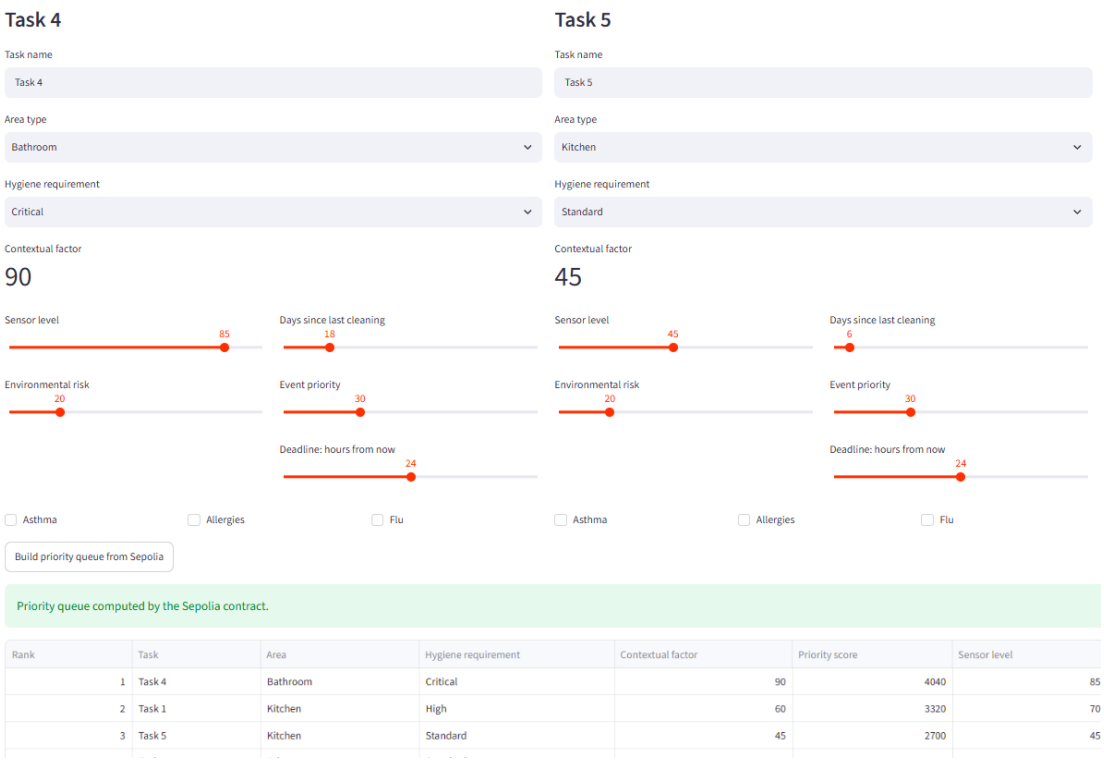


Figure 7.2: Priority queue visualization through Streamlit dashboard

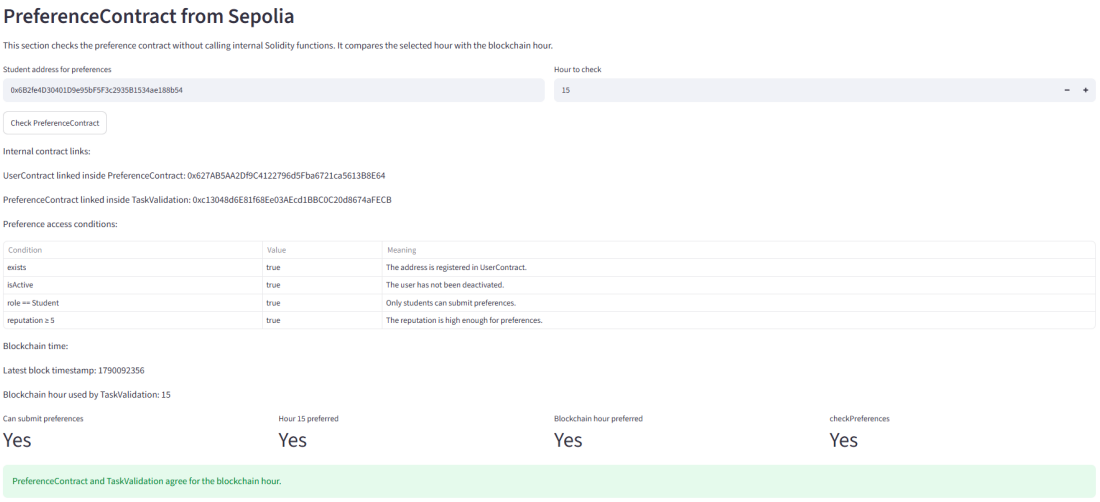


Figure 7.3: Preferences visualization through Streamlit dashboard

Chapter 8

Conclusions and Future Work

This thesis addressed the following research question: if a human-centered or contractualist-inspired artificial agent proposes an action, how can the subsequent validation, prioritization, and execution process be made transparent, traceable, and verifiable through blockchain technology? The answer developed in this work is not that blockchain can evaluate the moral quality of the agent's reasoning, nor that it can prove that a task is fair or understandable to users. Rather, the thesis shows that a smart contract layer can make selected downstream checks explicit and inspectable once a candidate task has entered the task-management process.

8.1 Implemented Capabilities

The implemented prototype focuses on a cleaning-task scenario in a student-residence environment, it does not include a deployed physical robot and does not reproduce the full Sweepy decision-making model. Instead, it implements the blockchain-supported infrastructure around a proposed cleaning task.

The final implementation includes four main smart contract components: `UserContract`, `PreferenceContract`, `TaskPriority` and `TaskValidation`.

The prototype also integrates an external contextual variable through Chainlink CRE. The workflow reads weather data from Open Meteo, extracts the fields needed by the model, and sends temperature and weather-code values to `TaskPriority`. The smart contract then applies its own deterministic rule to update the stored `weatherFactor`. In this architecture, the oracle integration is used as one example of how external context can enter the on-chain priority model; it is not presented as a complete general-purpose oracle system.

8.2 Evidence From the Case Study

The main evidence provided by the thesis is the behaviour observed in the prototype implementation and in the incremental testing path described in Chapters 5 and 6. The smart contracts were developed and tested through Remix VM, Ganache, and Sepolia

demonstrating that the contracts can be deployed, configured, called, and inspected in an Ethereum-compatible environment.

The case study makes several checks inspectable: a reviewer can inspect whether a student address was registered, whether the address was active, which role and reputation value were associated with it, which time preferences were stored, which robot-resource values were available, which priority parameters were submitted, which threshold was used, and whether the validation call emitted the corresponding event. If a task is accepted, the resulting state changes are also visible: robot resources are reduced, the internal student gas balance may be consumed or sponsored according to the simplified paymaster logic, the validation counter is updated, and the `TaskValidated` event records relevant information about the accepted task.

The oracle integration was also validated in a bounded way; the CRE workflow was first tested without broadcasting a transaction, confirming that the Open Meteo response could be read and processed. In the on-chain test, the workflow sent a weather update to `TaskPriority` on Sepolia, and the transaction receipt and contract getters showed that `lastUpdatedTemperature` and `weatherFactor` were updated. This observation supports the narrower claim that one external contextual value can be delivered to the contract and made available to the priority computation.

These results answer the research question at the level of technical traceability, in particular the case study shows that selected parts of the validation and prioritization process can be represented by explicit contract functions, state variables, access checks, and events. However, this evidence should not be interpreted as a user study, a fairness evaluation, or a validation of the off-chain agent's moral reasoning. Inspectable rules may support accountability, but they do not by themselves prove that users understand the system, that the rules are ethically sufficient, or that the selected task is fair in a broader social sense.

8.3 Limitations

The prototype remains a proof of concept, where robot resources are represented through abstract values such as battery, available time, and an internal operational cost. The system records successful validation or simulated completion inside the smart contract workflow, but it does not provide independent evidence that a physical robot actually cleaned a space.

Second, the task queue is not fully managed on-chain: the contracts can compute priority scores and validate a selected task, but they do not maintain a compulsory on-chain pending queue, automatically select the highest-ranked task, or dynamically revalidate all pending tasks when context changes. Queue ordering can be explored off-chain or in the dashboard, while the implemented contract layer focuses on score computation and final validation.

Third, the priority model uses heuristic parameters; in fact, the weights, coefficients, threshold, and weather-factor values are design choices used to test the architecture. They are not empirically calibrated, clinically validated, or derived from a participatory process with real users.

Fourth, the paymaster-inspired mechanism should not be treated as a fairness evaluation. It models an internal sponsorship policy and makes its effects observable, but it does not prove that the resulting allocation is fair, inclusive, or acceptable to affected users.

Fifth, the oracle path introduces trust assumptions since Chainlink CRE structures the delivery of the external value and the contract restricts who can update the weather state, but the original weather observation still comes from Open Meteo. The current contract does not enforce a maximum age for the weather value, and replay or freshness protection remains limited.

Finally, privacy remains a central boundary as the prototype avoids storing names and surnames on-chain, but a real deployment involving health conditions, disability-related needs, or behavioural data would require stronger privacy-preserving mechanisms. Transparency and auditability must therefore be balanced against the risk of exposing sensitive information permanently.

8.4 Future Work

Future work could extend the prototype in several directions. One direction would be to implement a full on-chain pending queue: candidate tasks could be stored on-chain, ranked according to the priority score, and selected directly by the contract. This would also require dynamic selection and revalidation, so that tasks can be recomputed when preferences, robot resources, weather state, or priority weights change.

A second direction would be to add a provenance and physical-completion interface, in fact the current prototype records validation and simulated completion, but not independent execution evidence. A future system could connect robot logs, sensor readings, signed execution reports, or human confirmation mechanisms to the smart contract, while still avoiding the direct exposure of sensitive data.

Another future work would be to integrate a more realistic robotic layer, including physical sensor data, navigation constraints, and multiple robots. The scenario could also be expanded to coordinated robots or swarm robotics, making it possible to study task allocation, resource sharing, and conflicts among several autonomous agents operating in the same environment.

Another development concerns artificial intelligence: AI could be introduced as an extension of the current rule-based prototype, for example to estimate dirtiness, predict urgency, learn from feedback, or suggest priority weights. It could also be used as a comparison baseline: future experiments could compare a rule-based smart contract approach, an AI-based prioritization model, and a hybrid architecture in which AI proposes or adapts decisions while smart contracts record, constrain, and validate selected outcomes.

Moreover a reward mechanism for users, robots, or system participants could be introduced. Such a mechanism could incentivize reporting, feedback, outcome validation, or participation in governance. However, the incentive design would need careful evaluation, because rewards can improve participation but can also distort behaviour.

Future work could model accessibility requirements more carefully making the system

explicitly suitable for people with disabilities or specific accessibility needs, for example by prioritizing tasks that affect wheelchair users, people with respiratory conditions, visually impaired users, or users who need unobstructed corridors and predictable environmental conditions. This extension would require direct attention to privacy, consent, and the risk of exposing sensitive attributes.

Further work is also needed on privacy-preserving design, empirical calibration, and governance. Sensitive information should be stored off-chain when possible, with on-chain hashes, commitments, or zero-knowledge techniques used only when appropriate. Priority weights and thresholds should be justified through domain knowledge, user studies, institutional requirements, or data-driven calibration. Governance rules should define who can update weights, register users, modify validation rules, authorize oracle sources, or approve changes proposed by an AI component.

8.5 Final Remarks

The broader human-centered value of the architecture should therefore be understood as a motivation and a prospect, not as a fully measured result. The thesis does not prove that the system is fair, that users understand the decision process, or that a robot completed a physical task. What it does show is more specific: selected validation rules, priority inputs, oracle-updated context, access checks, and state changes can be encoded in smart contracts and inspected after execution.

This is the main contribution of the work; by separating the off-chain proposal of a candidate action from the on-chain validation and recording layer, the prototype provides a concrete example of how blockchain can support traceability around human-centered robotic decision making. The result is not a complete autonomous ethical system, but a technical foundation on which more complete, privacy-aware, empirically evaluated, and user-centered systems could be developed.

AI use disclosure

Generative AI tools were used only as writing-support tools for language refinement, grammar checking, and improving the clarity of selected passages. All technical content, implementation choices, results, and final responsibility remain with the author.

Bibliography

[1] AccuWeather. Accuweather apis. Developer Documentation. https://developer.accuweather.com/documentation/overview. Accessed: 2026-09-17.

[2] Hamda Al-Breiki, Muhammad Habib Ur Rehman, Khaled Salah, and Davor Svetinovic. Trustworthy blockchain oracles: Review, comparison, and open research challenges. *IEEE Access*, 8:85675–85685, 2020.

[3] Hanouf Al Ghanmi, Sabreen Ahmadjee, and Rami Bahsoon. A systematisation of transparency, accountability and understandability in blockchain smart contracts. *ACM Transactions on Software Engineering and Methodology*, 2026. Just Accepted, accepted 7 April 2026, online 4 May 2026.

[4] Elizabeth Ashford and Tim Mulgan. Contractualism. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, summer 2018 edition, 2018.

[5] Lorenz Breidenbach, Christian Cachin, Benedict Chan, Alex Coventry, Steve Ellis, Ari Juels, Farinaz Koushanfar, Andrew Miller, Brendan Magauran, Daniel Moroz, Sergey Nazarov, Alexandru Topliceanu, Florian Tramèr, and Fan Zhang. Chainlink 2.0: Next steps in the evolution of decentralized oracle networks. White paper. https://research.chain.link/whitepaper-v2.pdf, April 2021. Accessed: 2026-09-17.

[6] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. White paper. https://ethereum.org/whitepaper/, 2014. Accessed: 2026-09-17.

[7] Chainlink. Building consumer contracts. Chainlink CRE Documentation. https://docs.chain.link/cre/guides/workflow/using-evm-client/onchain-write/building-consumer-contracts. Accessed: 2026-09-17.

[8] Chainlink. Chainlink functions: Getting started. Chainlink Documentation. https://docs.chain.link/chainlink-functions/getting-started. Accessed: 2026-09-17.

[9] Chainlink. Chainlink runtime environment (CRE). Chainlink Documentation. https://docs.chain.link/cre. Accessed: 2026-09-17.

[10] Chainlink. Simulating workflows. Chainlink CRE Documentation. https://docs.chain.link/cre/guides/operations/simulating-workflows. Accessed: 2026-09-17.

[11] Chainlink. What is a decentralized oracle network (DON)? Chainlink. https://chain.link/article/what-is-a-decentralized-oracle-network-don. Accessed: 2026-09-17.

[12] Chainlink. Chainlink runtime environment documentation. Chainlink Documentation. https://docs.chain.link/cre, 2026. Accessed: 2026-09-17.

[13] Giovanni Dalmasso, Luis Marcos-Vidal, and Clara Pretus. Modelling moral decision-making in a contractualist artificial agent. In Nardine Osman and Luc Steels, editors, *Value Engineering in Artificial Intelligence – 2nd International Workshop, VALE 2024, Revised Selected Papers*, volume 15356 of *Lecture Notes in Computer Science*, pages 155–175. Springer Nature Switzerland, 2025.

[14] Marco Dorigo, Alexandre Pacheco, Andreagiovanni Reina, and Volker Strobel. Blockchain technology for mobile multi-robot systems. *Nature Reviews Electrical Engineering*, 1:264–274, 2024.

[15] Ethereum. JSON-RPC API. Ethereum Documentation. https://ethereum.org/developers/docs/apis/json-rpc/. Accessed: 2026-09-17.

[16] Ethereum Improvement Proposals. EIP-140: Revert instruction. https://eips.ethereum.org/EIPS/eip-140, 2017. Accessed: 2026-09-17.

[17] Charles T. B. Garrocho, Frederico L. M. de Sousa, Mateus C. Silva, and Ricardo A. Rabelo. Blockchain-based smart contract and edge AI applied in a multirobot system: An approach. *IEEE Robotics & Automation Magazine*, 31(3):106–114, 2024.

[18] Jonathan Grey, Oshani Seneviratne, and Isuru Godage. Blockchain-based mechanism for robotic cooperation through incentives: Prototype application in warehouse automation. In *2021 IEEE International Conference on Blockchain (Blockchain)*, pages 597–604. IEEE, 2021.

[19] Jack Harris and Veljko Dubljević. Navigating the ethics of artificial intelligence. *Encyclopedia*, 5(4):201, 2025.

[20] Ammar Hassan, Imran Makhdoom, Waseem Iqbal, Awais Ahmad, and Asad Raza. From trust to truth: Advancements in mitigating the blockchain oracle problem. *Journal of Network and Computer Applications*, 217:103672, 2023.

[21] ISO 22739:2024: Blockchain and distributed ledger technologies – vocabulary. https://www.iso.org/standard/82208.html, 2024. Accessed: 2026-09-17.

[22] Shailak Jani. Smart contracts: Building blocks for digital transformation. Technical report, Indira Gandhi National Open University, 2020.

[23] Nilesh Joshi, Omkar R. Barve, and Manisha Bharati. A provenance solution for ethical generative AI and fair attribution. In *2025 IEEE Pune Section International Conference (PuneCon)*. IEEE, 2025.

[24] Zibin Lin, Taotao Wang, Chonghe Zhao, Shengli Zhang, Qing Yang, and Long Shi. A measurement investigation of ERC-4337 smart contracts on ethereum blockchain. In *2024 International Conference on Computing, Networking and Communications (ICNC)*, pages 1164–1170. IEEE, 2024.

[25] Dev Mehta, Akshat Jingar, Janam Patel, Rajesh Gupta, Pronaya Bhattacharya, Sudeep Tanwar, Chinmay Trivedi, and Maniklal Das. Blockchain and explainable ML-based secure network attack classification in smart homes. In *2024 IEEE Globecom Workshops (GC Wkshps)*, pages 1–6. IEEE, 2024.

[26] Hajar Moudoud, Soumaya Cherkaoui, and Lyes Khoukhi. An IoT blockchain architecture using oracles and smart contracts: The use-case of a food supply chain. In *2019 IEEE 30th Annual International Symposium on Personal, Indoor and Mobile*

Radio Communications (PIMRC), pages 1–6. IEEE, 2019.

[27] Muhammad Bin Saif, Sara Migliorini, and Fausto Spoto. Blockchain-based multi-role authentication and authorization in smart contracts with a hierarchical factory pattern. In *2024 6th International Conference on Blockchain Computing and Applications (BCCA)*, pages 22–29. IEEE, 2024.

[28] Open-Meteo. Open-meteo weather forecast api documentation. https://open-meteo.com/en/docs. Accessed: 2026-09-17.

[29] Vincent P. Paglioni and Torrey Mortenson. A systems-level approach to address risks and ethics in artificial intelligence systems. *Systems Engineering*, page e70062, 2026.

[30] Digantkumar N. Parmar, Sudharshan Putha, Renato R. Maaliw, Bhavani Prasad Kasaraneni, Faizan A. Reegu, and Hyeon Byeon. Blockchain-enhanced AI security and trust through auditable decision-making and data integrity in modern industry. In *2024 IEEE 2nd International Conference on Innovations in High Speed Communication and Signal Processing (IHCSP)*. IEEE, 2024.

[31] Iyad Rahwan. Society-in-the-loop: programming the algorithmic social contract. *Ethics and Information Technology*, 20(1):5–14, 2017.

[32] Thomas M. Scanlon. *What We Owe to Each Other*. Harvard University Press, Cambridge, MA, 1998.

[33] Ben Shneiderman. Human-centered artificial intelligence: Reliable, safe & trustworthy. *International Journal of Human-Computer Interaction*, 36(6):495–504, 2020.

[34] Solidity. Contracts. Solidity Documentation. https://docs.soliditylang.org/en/latest/contracts.html. Accessed: 2026-09-17.

[35] Solidity. Units and globally available variables. Solidity Documentation. https://docs.soliditylang.org/en/latest/units-and-global-variables.html. Accessed: 2026-09-17.

[36] Valentina Vinšalek Stipić, Zlatko Ćesić, and Mile Vičić. Analysis of the perception of managers of small and medium-sized enterprises on the advantages and disadvantages of electronic business contracts and smart contracts. In *2024 47th MIPRO ICT and Electronics Convention (MIPRO)*, pages 983–987. IEEE, 2024.

Glossary

ABI Application Binary Interface, an interface that defines how compiled software components interact at the binary level, including calling conventions, data types, and binary formats. 72

API (Application Programming Interface): An interface that defines a set of rules and protocols that enable different software components or applications to communicate and exchange data. 15, 65–68

capability A modular component that provides a specific functionality within the Chainlink ecosystem, enabling workflows to perform specialized operations, such as interacting with EVM-compatible blockchains through an EVM Capability or making HTTP requests to external services through an HTTP Capability. 65, 68, 71

event a Solidity mechanism used to record information in the transaction log when a relevant action occurs. Events can be read by external applications, such as frontends or indexing services, without continuously querying the full contract state. 41, 43, 55, 57, 60, 73

faucet a web service that distributes small amounts of testnet cryptocurrency to a wallet address. In Ethereum test networks, faucets are used to obtain test ETH needed to pay transaction fees during development and experimentation, without using real funds. 34

Ganache a local blockchain development environment that simulates a private Ethereum network. It is used to deploy and test smart contracts locally, with predefined accounts and test funds, before moving to a public test network 5.1.2. 65

hash A fixed-size value produced by a hash function from input data. It acts as a digital fingerprint and is commonly used to verify data integrity. 75

hash function A function that transforms input data of arbitrary size into a fixed-size value called a hash. Cryptographic hash functions are designed to be deterministic and resistant to collisions. 91

HTTP Hypertext Transfer Protocol, an application-layer protocol used for communication and data exchange between clients and servers on the Web. 66, 69, 71, 74

interface a contract definition that specifies which functions another contract can call, without including the internal implementation of those functions. 36, 39, 43, 55

SDK Software Development Kit, a collection of tools, libraries, documentation, and resources used to develop applications for a specific platform, framework, or system. 71

Sepolia Ethereum test network used to deploy and test smart contracts without using real Ether. In this thesis, Sepolia is used to test the interaction between the smart contracts and the Chainlink CRE workflow in a public blockchain environment. 63, 65

smart contract Programs deployed on a blockchain that execute predefined logic. 24, 27, 29, 30, 33, 42, 63

TypeScript A programming language that extends JavaScript by adding static typing and other language features. TypeScript code is compiled into JavaScript, allowing type-related errors to be detected during development and improving the maintainability of large codebases. 65, 67, 71, 72

WMO World Meteorological Organization, a specialized agency of the United Nations responsible for promoting international cooperation in meteorology, climatology, hydrology, and related geophysical sciences. 6, 67