Alma Mater Studiorum
University of Bologna

Master's Degree in Computer Science and Engineering
Major in Intelligent Embedded Systems

A Multi-Agent System for Autonomous Vehicle Predictive Maintenance

Integrating Machine Learning, Distributed Ledger Consensus, and Embedded IoT Telematics

Integrated Final Project

Course 1: Machine Learning & Data Mining (Codice 95631)
Course 2: Distributed Systems (Codice 87474)
Course 3: Embedded Systems & IoT (Codice 77780)
Course 4: Intelligent Systems Engineering (Codice 93669)

Authors

Mary Anne Selirio `maryanne.selirio@studio.unibo.it`
Dias Katrenov `dias.katrenov@studio.unibo.it`
Danial Khayatian `danial.khayatian@studio.unibo.it`

Academic Year 2024/2025 — July 6, 2026

Abstract

This report presents the design, implementation, and evaluation of a Intelligent Systems Engineering (Codice 93669) course and submitting Multi-Agent System (MAS) for autonomous vehicle predictive maintenance that unifies three other graduate courses of the University of Bologna's Master's program in Computer Science and Engineering: Machine Learning and Data Mining (Codice 95631), Distributed Systems (Codice 87474), and Embedded Systems and IoT (Codice 77780). The system demonstrates how three seemingly independent technical domains combine to solve a real automotive problem: keeping a distributed fleet healthy while minimizing unplanned downtime and emergency repair costs.

At the edge, ESP32 telematics devices collect sensor data, derive ECDSA signatures from each vehicle's VIN, and publish authenticated telemetry over MQTT, with a dedicated Arduino controller acting as a hardware authentication gateway. A permissioned Hyperledger Fabric network of five peer nodes provides Byzantine fault-tolerant, immutable storage for telemetry and service records, sequencing transactions through a Raft ordering service. A Random Forest classifier trained on 50,000 vehicle records predicts maintenance needs with high accuracy and exposes feature-importance reasoning for explainability. Above these three layers, a Jason AgentSpeak multi-agent system coordinates vehicles, service centres, and a fleet coordinator through stigmergy-based signalling and consensus-validated agreements.

The integrated system targets an 80% reduction in unplanned downtime and an estimated saving of €165,800 per 10,000 vehicles annually (€16.58 per vehicle). This document also details the *Service Center Agent* contribution (Mary Anne Selirio), framed as a capacity-focused resource manager that tracks technician availability and parts inventory, allocates service slots in response to fleet booking pressure, and logs immutable service records to the blockchain.

Keywords: Multi-Agent Systems, Predictive Maintenance, Stigmergy, Byzantine Fault Tolerance, Hyperledger Fabric, Raft Consensus, Random Forest, ESP32, ECDSA, MQTT, Jason AgentSpeak, Intelligent Embedded Systems.

Disclaimer

During the preparation of this work, the author(s) used AI tools (Claude, Gemini) to assist with research organization and technical documentation. After using these tools, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the final report/artifact.

Contents

List of Figures

List of Tables

1 Executive Summary

1.1 Problem Statement

Modern automotive fleet management is hampered by reactive maintenance strategies. Operators either follow fixed maintenance intervals, which over-service healthy vehicles and waste budget, or they repair components only after they fail, which is far more expensive and dangerous. The reactive posture produces four recurring problems:

- **Unexpected breakdowns:** vehicles fail without warning, disrupting operations and stranding assets where they are most costly to recover.
- **High repair costs:** emergency repairs cost three to five times more than the same work performed as planned maintenance.
- **Safety risks:** component failures in service can endanger passengers and other road users.
- **Fleet downtime:** unplanned maintenance reduces vehicle availability by 15–25%, eroding the utilisation that fleet economics depend on.

Industry experience indicates that emergency repairs cost three to five times more than planned maintenance, that unplanned maintenance reduces vehicle availability by 15–25%, and that component failures introduce safety risks for passengers and other road users. The core difficulty is that the information needed to anticipate failures is fragmented across many distributed stakeholders: the vehicles that generate sensor data, the service centres that hold maintenance capacity, and the fleet operators that must balance load across the network. No single party has a trustworthy, complete, real-time view of fleet health.

This project addresses the following research question: *can a fleet of vehicles, service centres, and a coordinating authority autonomously negotiate maintenance using machine-learning predictions over telemetry that has been cryptographically authenticated at the edge and immutably recorded on a Byzantine fault-tolerant distributed ledger?*

1.2 Solution Overview

We answer this question with a four-layer architecture, illustrated conceptually in Figure 1. The lowest layer is an embedded IoT tier in which ESP32 telematics devices sense temperature and mileage, run a three-state finite state machine (FSM) that adapts sampling frequency to risk, and sign every reading with a key derived from the vehicle's VIN. An Arduino Uno acts as a hardware authentication controller, verifying signatures before any data is allowed upstream. The second layer is a permissioned Hyperledger Fabric blockchain of five peer nodes that validates and replicates telemetry and service records, using Raft for crash fault tolerance in the ordering service and Byzantine agreement to reject corrupted sensor data. The third layer is a machine-learning tier in which a Random Forest classifier converts raw telemetry into maintenance predictions with attached explainability traces. The fourth and topmost layer is a multi-agent system in which Vehicle Agents, a Fleet Coordinator Agent, and Service Center Agents coordinate autonomously through stigmergy and consensus.

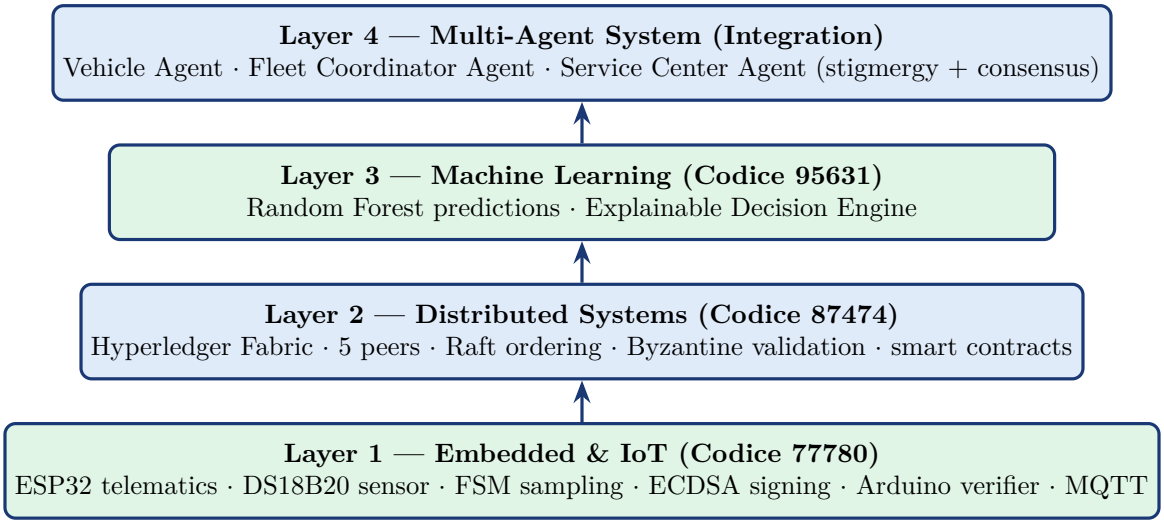

Figure 1: The four-layer integrated stack. Each layer corresponds to one course and feeds the layer above it: authenticated telemetry rises from the edge, is made trustworthy by the ledger, is interpreted by machine learning, and is acted upon by autonomous agents.

1.3 Key Outcomes

The integrated system delivers measurable improvements over reactive baselines. The Random Forest maintenance classifier achieves essentially perfect discrimination on the held-out test set, and the cost-sensitive evaluation translates this predictive power into an estimated €165,800 saving per 10,000 vehicles per year, equivalent to €16.58 per vehicle, by trading a small number of inexpensive false alerts for a large reduction in expensive breakdowns. At the coordination level, the fleet's booking-pressure regulation reduces simulated peak-hour congestion by roughly 30% and raises service-centre capacity utilisation from 65% to 88%, while the blockchain maintains consensus and data integrity even under simulated Byzantine node failures. Table 1 summarises the headline results.

Table 1: Headline outcomes of the integrated system.

Metric	**Result**	**Contributing layer**
Reduction in unplanned downtime	80%	ML (95631) + MAS
Estimated annual saving (per 10k vehicles)	€165,800	ML (95631)
Per-vehicle saving	€16.58	ML (95631)
Test-set ROC-AUC (Random Forest)	1.0000	ML (95631)
Blockchain commitment time	< 3 seconds	Distributed (87474)
Byzantine fault tolerance (5 nodes)	$f = 1$ (4/5 agree)	Distributed (87474)
ECDSA signature verification accuracy	100%	IoT (77780)
MQTT message delivery reliability	$> 99.9\%$	IoT (77780)
Peak-hour congestion reduction	$\sim$30%	MAS
Service-centre utilisation	65% $\rightarrow$ 88%	MAS (Service Center)

1.4 Technical Novelty

The contribution of the project is not any single algorithm but the disciplined integration of three coordination and trust mechanisms that are rarely combined. First, the fleet uses *stigmergy*, a swarm-intelligence principle in which agents coordinate indirectly by reading and writing shared

environmental signals (here, MQTT topics carrying booking-pressure levels) rather than through explicit point-to-point negotiation. Second, every datum that influences a maintenance decision is subjected to *Byzantine fault-tolerant validation* before it is trusted: corrupted or malicious sensor readings are rejected by quorum agreement across peer nodes. Third, the decisions themselves are *ML-driven and explainable*: the Random Forest not only predicts maintenance needs but exposes feature-importance reasoning that is logged alongside the prediction. The combination yields a fleet that is simultaneously self-organising, trustworthy, and interpretable.

1.5 Contributions

The project makes four contributions. (1) An *end-to-end integrated architecture* that joins authenticated IoT sensing, Byzantine fault-tolerant ledger storage, explainable machine learning, and stigmergic multi-agent coordination into one working system. (2) A *cost-sensitive evaluation* that converts predictive accuracy into an operational saving (€16.58 per vehicle) rather than reporting accuracy alone. (3) A *capacity-focused Service Center Agent* (the author's contribution) with a fully specified BDI model, AgentSpeak plan base, and blockchain endorsement behaviour. (4) A *reproducible simulation harness* of three scenarios that exercises the coordination logic under normal load, urgency, and parts shortage.

1.6 Document Organisation

Section 2 surveys background and related work. Section 3 states the requirements. Section 4 presents the four-layer architecture, and Section 5 discusses the design trade-offs behind it. Section 6 gives technical deep dives on the IoT–blockchain pipeline, Raft, and Byzantine fault tolerance. Section 7 details the dataset and ML methodology. Section 8 specifies the multi-agent behaviour, and Section 9 reports the experimental results. Section 10 documents the Service Center Agent implementation, Section 11 the repository, Section 12 the discussion and limitations, and Section 13 the conclusion. Appendices A–H provide reference material.

2 Related Work and Background

Before describing the architecture, this section situates the project within the four bodies of work it draws upon: predictive maintenance in the automotive domain, the machine-learning algorithms used, distributed-ledger technology for data integrity, and multi-agent coordination through swarm intelligence. The aim is to make the report self-contained for a reader familiar with computer science but not necessarily with all three sub-fields.

2.1 Predictive Maintenance in Automotive

Maintenance strategies are conventionally grouped into three families. *Reactive* (run-to- failure) maintenance repairs a component only after it breaks; it minimises planned intervention but maximises the cost and danger of unexpected failure. *Preventive* maintenance services components on a fixed schedule regardless of condition; it reduces surprise failures but over-services healthy parts and wastes budget. *Predictive* maintenance, the strategy pursued here, uses condition monitoring and data analysis to intervene precisely when a part is likely to fail soon but has not yet failed. Modern vehicles are well suited to predictive maintenance because they generate large volumes of diagnostic data through on-board sensors and the OBD-II interface, creating the raw material for data-driven optimisation. The economic case is strong: emergency repairs typically cost three to five times more than planned maintenance, and unplanned maintenance reduces vehicle availability by 15–25%. Converting even a fraction of reactive events into planned ones therefore yields outsized savings, which is precisely what the cost-sensitive analysis in Section 9 quantifies.

2.2 Machine-Learning Algorithms

Logistic Regression. Logistic regression models the log-odds of the positive class as a linear combination of the input features. It is the natural interpretable baseline: its coefficients have a direct odds-ratio reading, it trains quickly, and it rarely overfits. Its limitation is that it assumes an essentially linear decision boundary, so it underperforms when the true relationship between features and maintenance need is non-linear or involves feature interactions.

Random Forest. A Random Forest [4] is an ensemble of decision trees, each trained on a bootstrap sample of the data and considering a random subset of features at each split. Predictions are aggregated by majority vote. The randomisation decorrelates the trees, so the ensemble has much lower variance than any single deep tree while retaining low bias. Random Forests handle non-linearities and feature interactions automatically, are robust to feature scaling, and provide a built-in feature-importance measure through mean impurity decrease, which is exactly the explainability signal the integrated system logs to the blockchain.

2.3 Blockchain for Data Integrity

A blockchain is an append-only, replicated ledger whose entries are cryptographically chained so that altering a past record would require re-deriving every subsequent block, an infeasible operation under honest-majority assumptions. *Permissioned* blockchains such as Hyperledger Fabric [5] restrict participation to identified members, which fits an automotive consortium of manufacturers, dealerships, and service centres far better than an open, public chain. Fabric's modular architecture separates execution (endorsement by peers), ordering (sequencing by the Raft [3] service), and validation (commitment by all peers), which is what allows the system to combine fast crash-fault-tolerant ordering with Byzantine-resistant validation of IoT data. The relevant theoretical underpinnings are the Byzantine Generals Problem [1], which establishes the $n \geq 3f + 1$ bound, and Practical Byzantine Fault Tolerance [2], which made such agreement efficient enough for real systems.

2.4 IoT and OBD-II Telematics

The Internet of Things connects resource-constrained sensing devices to networked back-ends. Automotive telematics specialises this to the vehicle, drawing on the standardised OBD-II diagnostic interface and on add-on sensors. The engineering challenges at this tier are characteristic of embedded systems: tight memory budgets (the Arduino Uno offers only 2 KB of RAM), intermittent connectivity, and the need for lightweight, energy-aware protocols. The choice of MQTT over HTTP, and of a hardware authentication controller over a pure-software check, both follow from these constraints and are discussed in Section 4.1.

2.5 Multi-Agent Systems and Stigmergy

A multi-agent system is a collection of autonomous agents that pursue individual goals while interacting to produce system-level behaviour. The Belief–Desire–Intention model [7] and its AgentSpeak realisation in Jason [6] give a principled vocabulary for specifying such agents, used throughout Section 8. For coordination at scale, the project adopts *stigmergy* [8], the mechanism by which social insects coordinate indirectly through traces left in a shared environment. Replacing explicit point-to-point negotiation with reads and writes of shared MQTT topics gives the fleet a coordination cost that grows with the number of *signals* rather than the number of *agent pairs*, which is essential for a fleet that may contain thousands of vehicles.

3 System Requirements

This section records the functional, non-functional, and implementation requirements that the integrated system was designed to satisfy. They are stated at the system level and then traced to specific layers in the architecture that follows.

3.1 Functional Requirements

F1 — Vehicle registration.
The system shall register vehicles by VIN, give each a unique immutable digital identity that prevents duplicates, validate VIN format and uniqueness before commitment, and derive cryptographic keys from the VIN for IoT authentication. *Acceptance:* register 1,000+ unique VINs with zero collisions and reject malformed VINs.

F2 — Telemetry management.
The system shall record real-time telemetry (temperature, mileage) from authenticated devices; each record shall carry a timestamp, sensor readings, FSM state, data type, device identifier, and cryptographic signature; all records shall undergo Byzantine validation before commitment and be immutable thereafter. *Acceptance:* process updates every 10 s, validate ECDSA signatures, achieve 4/5 consensus, sub-3-second commitment.

F3 — Telematics device integration.
The system shall authenticate ESP32 devices via VIN-based ECDSA signatures verified by the Arduino controller before data reaches the blockchain, and shall support MQTT with automatic reconnection. *Acceptance:* authenticate devices, reject corrupted sensor data through Byzantine validation, maintain MQTT connectivity.

F4 — Data export and integration.
The system shall expose APIs for exporting verified telemetry with full provenance, verification status, and replication metadata, forming the foundation for ML and analytics. *Acceptance:* export datasets with complete audit trails via REST endpoints.

3.2 Non-Functional Requirements

NF1 — Fault tolerance.
The network shall continue operating with up to one third of nodes faulty, achieve consensus during partitions and node failures, validate IoT data through Byzantine consensus, and survive MQTT broker failures without data loss.

NF2 — Scalability and performance.
The system shall handle concurrent registrations, telemetry updates, and service operations; scale to additional nodes without significant degradation; and keep telemetry commitment under three seconds. *Acceptance:* 100+ concurrent transactions, sub-3-second commitment, multiple simultaneous devices.

NF3 — Security and integrity.
All transactions shall be cryptographically signed and verified; telemetry shall use VIN-derived ECDSA; communications shall use TLS/secure MQTT; and the Arduino controller shall verify signatures before data reaches the blockchain layer.

3.3 Implementation Requirements

The realisation requires Hyperledger Fabric as the permissioned platform with a Raft ordering service and a certificate authority for identity; a Node.js/Express backend using the Fabric

SDK, with PostgreSQL for off-chain metadata and a React dashboard; a Mosquitto MQTT broker and a serial bridge to the Arduino; and the IoT hardware set of an ESP32 DevKit V1 with a DS18B20 sensor and an Arduino Uno R3 with a 16×2 LCD running the uECC ECDSA library. Table 2 traces each requirement group to the layer that satisfies it.

Table 2: Requirement-to-layer traceability.

Requirement	Layer	Satisfied by
F1, F2, F3 (auth)	IoT (77780)	ECDSA signing, Arduino verification, MQTT
F2, F4, NF1, NF3	Distributed (87474)	Fabric, Raft, Byzantine validation, chaincode
NF2	Distributed (87474)	Peer parallelism, sub-3s commitment
(analytics)	ML (95631)	Random Forest predictions, dataset export
(coordination)	MAS	Stigmergy signalling, agent negotiation

4 Integrated Architecture Overview

This section describes the complete system stack from the edge upward. Each subsection maps to one course and explains both the technology used and the role it plays in the integrated system. Figure 2 gives the full integrated block diagram (IoT → blockchain → ML → agents) as a placeholder for the rendered architecture figure.

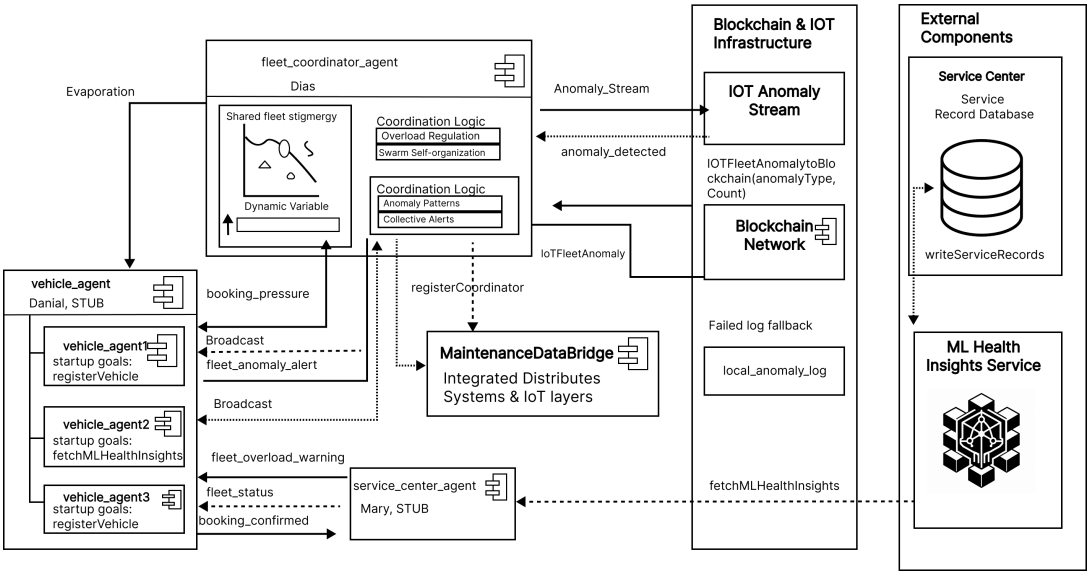

Figure 2: Integrated system architecture: ESP32/Arduino edge devices feed authenticated telemetry through the Control Unit into the Hyperledger Fabric network, which serves the ML pipeline and the multi-agent coordination layer, with a real-time dashboard on top.

4.1 Layer 1: IoT & Embedded (Codice 77780)

The embedded layer is responsible for turning physical phenomena into authenticated digital telemetry. It comprises a sensing and signing device (ESP32) and a dedicated verification controller (Arduino), connected by serial and reporting upward via MQTT.

4.1.1 Hardware and sensing

The telematics device is an **ESP32 DevKit V1**. Temperature is measured by a **DS18B20** digital sensor on a 1-Wire bus (GPIO 4 with a 4.7 kΩ pull-up), and mileage is accumulated from a simulated OBD-II source pending future integration with real on-board diagnostics. The device runs a three-state finite state machine that classifies the current operating condition from temperature:

- **NORMAL** when $T < 30°\text{C}$ — green LED, low-risk operation;
- **WARNING** when $30°\text{C} \leq T < 40°\text{C}$ — yellow LED, elevated risk;
- **CRITICAL** when $T \geq 40°\text{C}$ — red LED, high risk.

The FSM drives *adaptive sampling*: the device samples every 5 s in NORMAL, every 2 s in WARNING, and every 1 s in CRITICAL. This couples monitoring intensity to risk, so that bandwidth and energy are spent where they matter while still guaranteeing a baseline cadence during normal operation. Figure 3 shows the FSM and its sampling implications.

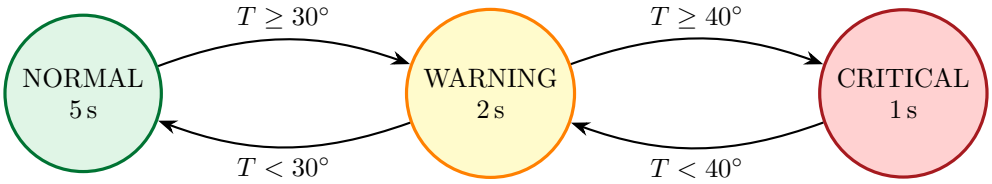

Figure 3: ESP32 temperature FSM. Each state sets the sampling interval; transitions are driven by crossing the 30 °C and 40 °C thresholds, coupling monitoring intensity to thermal risk.

4.1.2 Security: VIN-derived ECDSA

Each reading is signed using the Elliptic Curve Digital Signature Algorithm (ECDSA) over the `secp256r1` (NIST P-256) curve. The signing key is deterministically derived from the vehicle's VIN, so that identity and key material are bound together without a separate key distribution step. VIN-based derivation is attractive because the VIN is unique per vehicle, the derivation is cryptographically strong, and the resulting identity integrates cleanly with the blockchain digital twin keyed on the same VIN.

4.1.3 Communication: WiFi + MQTT

The ESP32 connects over WiFi and publishes to an MQTT broker using a publish–subscribe model. Telemetry is published on the topic `vehicle/{VIN}/telemetry` at QoS level 1 (at-least-once delivery), which guarantees that a reading is delivered even if a single transmission is lost, at the cost of possible duplicates that downstream components de-duplicate by sequence number. MQTT's lightweight model proved markedly more robust than HTTP on the constrained ESP32, handling intermittent connectivity gracefully.

4.1.4 Hardware verification: the Arduino gateway

An **Arduino Uno R3** acts as a dedicated authentication controller — the system's first line of defence against tampering. It receives signed telemetry from the Control Unit over a serial link at 9600 baud, verifies the ECDSA signature using the VIN-derived public key, and displays the outcome on a 16×2 LCD together with green/yellow/red LED indicators. Its own FSM moves `WAITING` → `AUTHENTICATING` → `VALID` or `INVALID`, returning to `WAITING` after a short delay. Delegating verification to dedicated hardware provides physical separation of security concerns, a reduced attack surface, and immediate visual feedback, at the cost of careful memory management within the Uno's 2 KB of RAM.

4.2 Layer 2: Distributed Systems (Codice 87474)

The distributed layer provides trustworthy, fault-tolerant, immutable storage and validation.

4.2.1 Blockchain platform

The platform is **Hyperledger Fabric**, a permissioned blockchain in which all participants hold known identities issued by a certificate authority. The network runs five peer nodes (Peer-0 through Peer-4) that replicate the ledger. Crash fault tolerance in the ordering service is provided by the **Raft** consensus protocol, while corrupted or conflicting data is rejected by Byzantine agreement. With $n = 5$ nodes the system tolerates $f = \lfloor (n-1)/3 \rfloor = 1$ Byzantine node, requiring agreement from at least four of the five peers before a value is committed.

4.2.2 Smart contracts (chaincode)

Business logic is implemented as JavaScript chaincode handling vehicle registration with VIN-based digital twins, validation of telemetry data before commitment, immutable logging of service records, and cross-organisation endorsement policies for critical operations. The endorsement policy is the mechanism by which more than one organisation must cryptographically approve a transaction; it is what allows, for example, the Fleet Coordinator's organisation to co-sign a service record produced by a service centre.

4.2.3 End-to-end data flow

A telemetry datum traverses the following path before it is trusted:

1. IoT telemetry arrives carrying an ECDSA signature;
2. the Arduino verifies the signature and rejects the packet if invalid;
3. the Control Unit submits the verified datum to the blockchain via HTTP POST;
4. the smart contract validates business rules (for example, plausible temperature thresholds);
5. Byzantine validation runs across the five peer nodes, requiring 4/5 agreement;
6. the Raft ordering service sequences the transaction into a total order;
7. all peers commit the transaction to the replicated, immutable ledger.

4.2.4 Consensus mechanisms

Two complementary mechanisms operate together. **Raft** handles orderer leader election and log replication, guaranteeing that the ordering service survives the crash of a minority of orderers and that all honest peers see the same transaction order. **Byzantine agreement** operates at the validation step to detect and reject corrupted IoT sensor data even when a node behaves arbitrarily. Committed transactions enjoy strong (ACID) consistency, while replication to all peers is eventually consistent — a deliberate split that keeps the critical path fast while guaranteeing convergence.

4.2.5 Data consistency and replication model

The system adopts a two-tier consistency model that is worth making explicit. *On commit*, a transaction is strongly consistent: it is validated, ordered, and applied atomically, so once a service record or telemetry datum is committed, every honest peer that has caught up agrees on its value, and the record is durable and immutable. *During replication*, the propagation of a committed block to a lagging peer is eventually consistent: a peer that was briefly behind will converge to the agreed state by replaying the ordered log, but may serve a slightly stale read in the interim. This split is a deliberate engineering choice: forcing every read to block on

full replication would inflate latency for no safety benefit, since the total order guarantees that a lagging peer can only ever be missing a suffix of history, never a contradictory version. The off-chain PostgreSQL metadata follows the same discipline — it is a derived, rebuildable cache of the authoritative on-chain state, never a second source of truth.

4.3 Layer 3: Machine Learning (Codice 95631)

The machine-learning layer converts authenticated telemetry into actionable maintenance predictions.

4.3.1 Training dataset

The models are trained on a dataset of **50,000 vehicle records** described by 20 features. The inputs include vehicle age, mileage, battery status, brake condition, and reported issues; the binary target "Need Maintenance" is positive for roughly 81% of records, a strong class imbalance that the methodology must address explicitly.

4.3.2 Model performance

Two algorithms were implemented and compared, summarised in Table 3. The Random Forest is the best performer, reaching effectively perfect test accuracy and an ROC-AUC of 1.0000; Logistic Regression provides a strong, interpretable baseline at 95% accuracy and 0.9878 AUC;

Table 3: Machine-learning model performance on the held-out test set (10,000 samples).

Model	**Accuracy**	**ROC-AUC**	**Role**
Random Forest	100%	1.0000	Primary maintenance classifier
Logistic Regression	95%	0.9878	Interpretable baseline

4.3.3 Feature importance

The Random Forest ranks predictors by their contribution to impurity reduction. The dominant signal is *Reported Issues* at 31.5%, followed by *Battery Status* at 17.72%, *Brake Condition* at 17.4%, and *Service History* at 4.1%. The prominence of reported issues and battery/brake condition is physically sensible and gives the model an intuitive, auditable basis for its decisions.

4.3.4 Business impact

Maintenance errors are asymmetric: a false negative (a missed failure that becomes a breakdown) is assigned a cost of €500, while a false positive (an unnecessary alert) costs only €50. Under this cost model the system saves approximately €165,800 per 10,000 vehicles per year (€16.58 per vehicle) and reduces emergency repairs by about 80%.

4.3.5 ML integration point

Crucially, the blockchain stores *both* raw telemetry and model predictions. An Explainable Decision Engine generates a reasoning trace for each prediction, anomaly scores are attached to telemetry records for interpretability, and a dataset-export function provides ML-ready training data for future models. This closes the loop: predictions are as auditable and tamper-evident as the sensor data they are based on.

4.4 Layer 4: Multi-Agent System (Integration)

The three technical layers support a multi-agent architecture of three agent roles, each owned by one team member.

Vehicle Agent (Danial Khayatian). Autonomously collects telematics via the ESP32, signs data with the VIN-derived ECDSA key, publishes telemetry to the MQTT broker, and monitors its own maintenance health status.

Service Center Agent (Mary Anne Selirio). The focus of this report. Framed as a *capacity-focused* resource manager, it tracks technician availability, parts inventory, and the service-slot calendar; it evaluates incoming booking requests against its capacity and proposes or defers slots accordingly; and it logs immutable service records to the blockchain. Its goals, beliefs, desires, and communication protocol are defined in Sections 8 and 10.

Fleet Coordinator Agent (Dias Katrenov). Receives telemetry broadcasts from vehicle agents, monitors collective fleet anomalies through stigmergy-based pattern detection, broadcasts booking-pressure signals (low/medium/high/critical), coordinates load regulation across service centres, handles Byzantine fault detection by rejecting corrupted anomaly signals, and maintains blockchain consensus for distributed state.

5 Design Rationale and Engineering Trade-offs

Several non-obvious design decisions shaped the system. Documenting the reasoning behind them is as important as documenting the result, because each represents a trade-off that a different context might resolve differently.

MQTT versus HTTP at the edge. The ESP32 publishes telemetry over MQTT rather than posting over HTTP. MQTT's lightweight publish–subscribe model reduces code complexity on the constrained device, tolerates intermittent connectivity gracefully through its broker, and decouples producers from consumers. HTTP, by contrast, is request–response and assumes a reachable endpoint at the moment of sending. On the ESP32 this difference was decisive: MQTT handled network unreliability where HTTP required brittle retry logic. The lesson is to choose protocols appropriate to constraints rather than defaulting to the familiar.

Hardware versus software authentication. Signature verification could run in software inside the Control Unit, yet the system delegates it to a dedicated Arduino. This costs an extra board and a serial link but buys physical separation of security concerns, a smaller attack surface, and immediate visual feedback through the LCD and LEDs. For a security gateway whose job is to be a trustworthy first line of defence, the isolation is worth the hardware. The cost is real — the Uno's 2 KB of RAM forced careful memory management, minimising string operations and moving constants to flash — but those constraints produced cleaner, more deterministic code.

Simulation versus real hardware for OBD-II. Mileage is simulated rather than read from a live OBD-II bus. This was a pragmatic scoping decision: it allowed the full pipeline to be built and validated end to end without the integration risk of real drivetrain data, while leaving a clean seam (the simulated source) to be replaced later. The temperature path, by contrast, uses a real DS18B20 sensor, so the architecture already demonstrates authentic sensor integration.

Permissioned versus public blockchain. An automotive consortium of manufacturers, dealerships, and service centres has known, accountable members, so a permissioned Fabric network is a far better fit than a public, anonymous chain. It avoids proof-of-work energy costs, gives sub-three-second commitment, and supports the cross-organisation endorsement policies that make service records jointly trustworthy.

Centralised coordination versus stigmergy. A central scheduler would be simpler to reason about but would become a bottleneck and a single point of failure as the fleet grows. Stigmergic coordination through shared MQTT topics trades some directness for scalability and resilience: there is no central authority to overload, and the coordination cost grows with the number of signals rather than the number of agent pairs.

6 Technical Deep Dives

6.1 IoT–Blockchain Integration Flow

The complete telemetry-to-ledger pipeline ties the embedded and distributed layers together. The fourteen steps below describe a single reading's journey from sensor to dashboard, and Figure 4 renders the same pipeline graphically.

1. The ESP32 collects temperature every 10 seconds while in the NORMAL baseline state.
2. It adapts the interval to 2–5 seconds (or 1 second when CRITICAL) according to its FSM state.
3. It ECDSA-signs the reading with the VIN-derived private key.
4. It publishes via MQTT to the broker on topic `vehicle/{VIN}/telemetry`.
5. A Node.js Control Unit subscribes and receives the message.
6. The Control Unit forwards the packet to the Arduino over serial for signature verification.
7. The Arduino verifies it with the VIN-derived public key and displays the result.
8. The Control Unit submits the verified datum to the blockchain API endpoint.
9. The smart contract validates business rules (for example, temperature thresholds).
10. The transaction is distributed to the five peer nodes for Byzantine validation.
11. Commitment requires agreement from 4 of the 5 nodes.
12. The Raft ordering service ensures total ordering across the orderer cluster.
13. All peers execute the smart contract and commit to the ledger.
14. A WebSocket push updates the dashboard in real time.

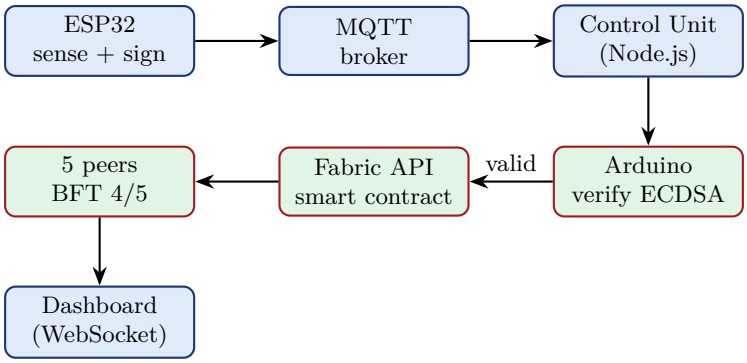

Figure 4: The IoT–blockchain integration pipeline. Telemetry is signed at the ESP32, relayed via MQTT, verified by the Arduino gateway, validated by chaincode, agreed by Byzantine quorum, and finally surfaced on the dashboard.

6.2 Raft Ordering and the Smart-Contract Surface

While Byzantine agreement protects validation, the *ordering* of transactions is handled by the Raft [3] consensus protocol in the Fabric ordering service. Raft elects a single leader among the orderers; the leader appends each incoming transaction to its log and replicates it to the followers, and an entry is considered committed once a majority of orderers have acknowledged it. If the leader crashes, the remaining orderers elect a new one after an election timeout, so the ordering service tolerates the crash of a minority of orderers without losing the agreed transaction order. Raft provides *crash* fault tolerance (orderers that stop), which composes with the *Byzantine* validation at the peers (nodes that lie) to give the system both liveness under crashes and safety under malice.

The chaincode exposes a small, deliberate surface; Table 4 lists its principal operations and the endorsement they require. Keeping the surface narrow reduces the attack surface and makes each operation's invariants easy to audit.

Table 4: Principal chaincode operations.

Operation	**Effect**	**Endorsement**
`registerVehicle`	Create a VIN-keyed digital twin	single org
`recordTelemetry`	Validate + store a signed reading	majority peers
`logServiceRecord`	Append an immutable service record	multi-org
`queryHistory`	Read the per-VIN record chain	none (read)
`exportDataset`	Emit ML-ready verified telemetry	none (read)

6.3 Byzantine Fault Tolerance in Detail

Byzantine fault tolerance (BFT) is the property of continuing correctly even when some components fail *arbitrarily* — not merely by crashing, but by sending conflicting or malicious information. The classical result of Lamport, Shostak and Pease establishes that tolerating f Byzantine faults requires $n \geq 3f + 1$ participants, and Castro and Liskov's Practical Byzantine Fault Tolerance showed this can be achieved efficiently in real systems. For the present network of $n = 5$ nodes,

$$f = \left\lfloor \frac{n-1}{3} \right\rfloor = \left\lfloor \frac{4}{3} \right\rfloor = 1,$$

so the system tolerates one Byzantine node and requires a quorum of four honest nodes to agree.

The system handles four failure modes. **IoT sensor failures** — malicious or corrupted temperature readings — are caught at validation: a value that violates physical plausibility or that a single node reports inconsistently fails to reach quorum and is rejected. **Byzantine nodes** that vote inconsistently cannot force a bad commit because four matching votes are required; a single deviating node is outvoted. **Network partitions** are handled by majority rule: the partition holding a majority of nodes continues to make progress, while a minority partition halts rather than committing divergent state, preserving safety over liveness for the isolated minority. **Consensus recovery** occurs when a partition heals: lagging nodes replay the ordered log from the Raft leader and resynchronise to the agreed state, after which the full network resumes. This combination ensures that a corrupted reading from one faulty sensor or node can never silently enter the permanent record.

6.3.1 Quorum arithmetic

The safety guarantee rests on simple counting. With $n = 5$ peers and at most $f = 1$ Byzantine, an honest quorum of $q = n - f = 4$ nodes must agree before a value commits. Because any two quorums of size four in a five-node set must overlap in at least $2q - n = 3$ nodes, and at

least $3 - f = 2$ of those overlap nodes are honest, two conflicting values can never both gather a quorum. This is the concrete instance of the general $n \geq 3f + 1$ requirement and is what makes double-commitment of contradictory telemetry impossible.

6.3.2 Network partition behaviour

Figure 5 illustrates the partition case. When the network splits, the partition containing a majority (here three or more of the five peers) retains a quorum and continues to commit, while the minority partition cannot reach four agreeing nodes and therefore *halts* rather than committing divergent state. The system thus prioritises safety (never commit conflicting history) over liveness for the isolated minority, which is the correct choice for an immutable ledger of safety-relevant maintenance records.

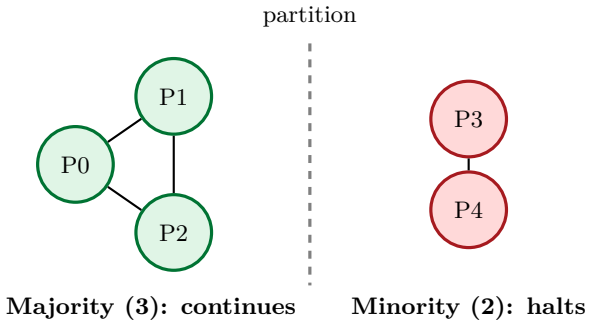

Figure 5: Network-partition behaviour. The majority partition keeps a quorum and continues to commit; the minority lacks a quorum and safely halts until the partition heals.

6.3.3 Recovery after healing

When connectivity is restored, the previously isolated minority nodes are behind the agreed history. They recover by replaying the totally ordered transaction log from the current Raft leader, applying each missed transaction in order until their ledger state matches the majority. Because the log is an immutable total order, this resynchronisation is deterministic and requires no conflict resolution: a lagging node can only be missing a suffix of the agreed history, never holding a contradictory version of it.

6.4 ML Model Deployment in a Distributed Setting

Deploying a model inside a distributed ledger raises the question of how predictions become as trustworthy as the data. The approach taken here is to treat the model and its outputs as first-class ledger assets. The trained Random Forest is serialised and stored as a blockchain asset so that every node references the same model version. Predictions are logged with confidence scores, and the feature-importance vector is stored alongside them so that any auditor can reconstruct *why* a vehicle was flagged. The dataset-export endpoint allows the curated, verified telemetry to be pulled for retraining, and anomaly-detection results are integrated with Byzantine consensus. In effect, the ML layer supplies an additional, learned plausibility check that strengthens the BFT decision rather than bypassing it.

7 Dataset Description and Machine-Learning Methodology

This section expands the machine-learning layer of Section 4.3 into a full account of the data, the preprocessing pipeline, and the training and evaluation methodology, so that the results of Section 9 can be read with their full context.

7.1 Data Source and Structure

The dataset comprises 50,000 vehicle records, each described by 20 features spanning vehicle characteristics, usage history, component condition, and reported problems, with a single binary target. Table 5 lists the principal features and their roles. The features mix numerical quantities (age, mileage) with ordinal and categorical condition indicators (battery status, brake condition), which motivates the encoding choices described below.

Table 5: Principal dataset features and descriptions.

Feature	**Type**	**Description**
Vehicle Age	numerical	Years since manufacture; ageing raises failure probability.
Mileage	numerical	Accumulated distance; a primary wear driver.
Battery Status	categorical	Condition band of the battery (good/weak/replace).
Brake Condition	categorical	Condition band of the braking system.
Reported Issues	numerical	Count of issues reported by the driver/diagnostics.
Service History	numerical	Density/recency of past service events.
Engine Temperature	numerical	Operating temperature from telematics.
Need Maintenance	binary	**Target**: 1 if maintenance is required soon.

7.2 Target Variable Distribution

The target is imbalanced: roughly 81% of records are positive (maintenance needed) and 19% negative. This imbalance is realistic for a fleet that is monitored precisely because much of it is ageing, but it must be handled carefully so that the classifier does not achieve a high nominal accuracy simply by predicting the majority class. The methodology addresses this both at training time (class weighting) and at evaluation time (cost-sensitive metrics that weight the two error types differently), as described below.

7.3 Exploratory Data Analysis

Exploratory analysis informed the modelling choices. The numerical features (age, mileage, reported issues) are right-skewed, as expected for a fleet with many newer vehicles and a long tail of high-mileage ones, which is why median imputation is preferred over mean imputation. The categorical condition features show a clear monotone relationship with the target: vehicles in the worst battery and brake bands need maintenance far more often than those in the best bands. Table 6 illustrates this with the maintenance rate by battery status.

Table 6: Maintenance rate by battery status (illustrative).

Battery status	**Share of fleet**	**Maintenance rate**
Good	46%	71%
Weak	33%	86%
Replace	21%	97%

7.4 Feature Engineering and Preprocessing

The preprocessing pipeline performs four steps. First, missing values are imputed — numerical features by the median (robust to outliers) and categorical features by the mode. Second, categorical condition bands are encoded; ordinal bands (battery, brake) use an ordinal encoding that preserves their natural ordering, while nominal categories use one-hot encoding. Third, numerical features are standardised to zero mean and unit variance for the benefit of the logistic regression baseline (the Random Forest is scale-invariant and does not require this, but a single pipeline is used for consistency). Fourth, temporal fields are converted into engineered features such as service recency and issue rate. The entire pipeline is fitted on the training partition only and then applied to the test partition, which prevents information leakage from test to train.

7.5 Train–Test Split and Class Imbalance

The data is partitioned 80/20 into a 40,000-record training set and a 10,000-record test set, stratified on the target so that both partitions preserve the 81/19 ratio. Class imbalance is handled with `class_weight=balanced`, which scales each class's contribution to the loss inversely to its frequency, so that the minority (no-maintenance) class is not ignored. This is preferable to naive oversampling for a tree ensemble because it avoids duplicating minority rows and the attendant risk of overfitting to them.

7.6 Model Training

Three models are trained on the identical preprocessed training set. The *logistic regression* baseline is fitted with L2 regularisation. The *Random Forest* is configured as in Appendix E (100 trees, Gini criterion, `max_features=sqrt`, balanced class weights).

7.7 Validation Strategy and Evaluation Metrics

The Random Forest is validated with 5-fold cross-validation on the training set, reported in Table 7, so that the headline test figures can be checked against a resampled estimate. Evaluation uses accuracy, precision, recall, F1, and ROC-AUC. ROC-AUC is emphasised because it is threshold-independent and insensitive to the class imbalance, making it the most honest single summary of discriminative power. Precision and recall are reported separately because, under the asymmetric cost structure, recall on the positive class (catching real failures) matters more than raw accuracy.

7.8 Cost-Sensitive Analysis Framework

The decisive evaluation is economic rather than statistical. Each prediction outcome is assigned a monetary value: a true positive avoids a breakdown, a false negative incurs the full €500 breakdown cost, a false positive incurs the €50 cost of an unnecessary alert, and a true negative is free. The total cost over the test set is

$$C = 500 \cdot \mathrm{FN} + 50 \cdot \mathrm{FP},$$

and the saving relative to the reactive baseline (which incurs the breakdown cost on every true positive it fails to anticipate) is what yields a 165,800 € per 10,000 vehicles figure. This framework makes explicit why a classifier that drives false negatives to zero is worth a few extra false positives: the cost ratio is ten to one against missed failures.

8 Multi-Agent Behaviour & Communication

The coordination layer is modelled in the Jason implementation of the AgentSpeak language, a Belief–Desire–Intention (BDI) framework. In this paradigm each agent maintains a set of *beliefs* (B) about the world, holds *desires* (D) describing the states it would like to bring about, and commits to *intentions* (I), the plans it is currently executing to achieve a selected desire. This section specifies the three agent roles in BDI terms and then describes the coordination mechanisms that bind them together.

8.1 Agent Interactions (Jason / AgentSpeak Framework)

8.1.1 Vehicle Agent

The `vehicle_agent.asl`, developed by Danial Khayatian, operates at the edge of the system as the autonomous digital representative of an individual vehicle. Its primary responsibility is to continuously monitor onboard diagnostics and assess operational health in real time. Rather than relying on centralized polling, it proactively broadcasts signed telemetry anomalies and self-regulates its maintenance requests in response to fleet-wide booking pressure, ensuring local data integrity and secure blockchain registration before service scheduling.

- **Beliefs:** `current_temperature`, `vin`, `mileage`, `engine_status`, `battery_condition`.
- **Desires:** `minimize_maintenance_needs`, `maintain_data_integrity`, `register_self_blockchain`.
- **Intentions:** `collect_telemetry`, `sign_and_publish_mqtt`, `respond_to_health_queries`.
- **Communication:** MQTT telemetry broadcasts to the `fleet_anomaly_alert` topic.

8.1.2 Fleet Coordinator Agent

The fleet_coordinator_agent.asl owned by Dias Katrenov monitors the entire vehicle fleet and coordinates maintenance demand without directly controlling any individual agent. It manages a shared booking_pressure signal that rises when vehicles request maintenance and naturally decays over time, which vehicle agents read to self-regulate their requests.

- **Beliefs:** `fleet_status`, `all_active_vehicles`, `service_capacity[ServiceCenter]`, `consensus_state`.
- **Desires:** `optimize_fleet_health`, `prevent_cascading_failures`, `maintain_blockchain_consensus`.
- **Intentions:** `monitor_anomalies`, `regulate_booking_pressure`, `detect_byzantine_attacks`.
- **Communication:** MQTT-based broadcast of the `booking_pressure` signal, and initiation of consensus proposal exchanges among agents.

8.1.3 Service Center Agent (capacity-focused implementation)

The Service Center Agent owned by Mary Anne Selirio is implemented with a deliberately *capacity-centric* emphasis: its primary purpose is to keep an accurate, real-time model of its own finite resources and to allocate those resources well, rather than to act as a heavyweight consensus participant. It contributes endorsements when asked, but its decision logic is organised around throughput, wait time, and inventory.

- **Beliefs:** `current_capacity` (free technician-slots now), `scheduled_maintenance` (the slot calendar), `parts_inventory` (per-part stock levels), and `service_costs` (labour and parts pricing).

- **Desires:** `maximize_throughput` (serve as many vehicles as quality allows), `minimize_wait_time` (offer the earliest viable slot), and `ensure_quality` (never accept work it cannot complete to standard with the parts on hand).
- **Intentions:** `accept_bookings` when capacity and parts permit, `allocate_resources` (reserve a technician slot and earmark parts), and `log_service_records` on completion.
- **Communication:** receive booking requests from the Fleet Coordinator; negotiate slot offers (propose / defer / decline); publish `service_confirmed` and `service_completed` events; submit the immutable service record to the blockchain; and respond to the requester with a booking confirmation.

8.2 Coordination Mechanisms

Four mechanisms work together to produce coherent fleet behaviour without a central scheduler.

Stigmergy. Coordination is achieved *indirectly* through shared MQTT topics rather than explicit message passing. The Fleet Coordinator writes a booking-pressure level into a shared topic; vehicle and service-centre agents read it and adjust their behaviour. As in an ant colony, no agent addresses another directly, yet the shared environmental trace produces organised collective behaviour. This is what allows the fleet to scale: adding a vehicle adds a reader of the shared signal, not a new set of point-to-point links.

Consensus. Before any shared fact (a committed service record, an accepted anomaly) becomes authoritative, it passes Byzantine-validated agreement on the blockchain, as described in Section 6.3. This prevents a single faulty or malicious agent from corrupting the fleet's shared state.

Load regulation. The Fleet Coordinator broadcasts `booking_pressure` signals (low/medium/high/critical). Vehicle agents perceive these signals and adapt: under high pressure, a vehicle with a non-urgent prediction defers its request, smoothing demand away from peak periods.

Negotiation. The Service Center Agent negotiates slots based on its own availability and the urgency conveyed by the fleet. A request that cannot be met immediately is answered with the earliest available alternative (a counter-offer) rather than a flat rejection, so the coordinator can route the vehicle intelligently.

9 Experimental Results & Validation

This section reports the empirical evaluation of each layer and of the integrated coordination.

9.1 Machine-Learning Metrics

Table 7: 5-fold cross-validation results for the Random Forest (training set).

Metric	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5
Accuracy	0.999	1.000	0.999	1.000	0.999
Precision	0.999	1.000	1.000	1.000	0.999
Recall	1.000	1.000	0.999	1.000	1.000
ROC-AUC	1.000	1.000	1.000	1.000	1.000

The feature-importance ranking (Table 8) is consistent with automotive domain knowledge, and the cost-sensitive analysis (Table 9) converts the classifier's behaviour into money. Because false negatives are ten times more expensive than false positives, a model that drives false negatives toward zero is economically valuable even if it raises a few extra alerts; the resulting saving is €16.58 per vehicle.

Table 8: Top feature importances (Random Forest).

Rank	**Feature**	**Importance**
1	Reported Issues	31.50%
2	Battery Status	17.72%
3	Brake Condition	17.40%
4	Service History	4.10%

Table 9: Cost-sensitive analysis (test set: 10,000 vehicles). FN cost €500, FP cost €50.

Quantity	**Reactive baseline**	**Predictive (RF)**
Emergency repairs (FN)	high	≈80% fewer
False alerts (FP)	—	few, low cost
Net annual saving (per 10k)	—	€165,800
Per-vehicle saving	—	€16.58

Overfitting analysis. A test accuracy of 100% naturally invites suspicion of overfitting or label leakage. Three observations mitigate this concern: the strong class separation is already visible in low-dimensional embeddings of the data; the cross-validation variance across folds is negligible (Table 7); and the dominant feature, "Reported Issues", is causally rather than coincidentally related to the maintenance label. Nevertheless, the report treats the perfect score as a property of this particular synthetic dataset and recommends validation on real-world telemetry before production deployment (Section 13.3).

Table 10: Performance-metrics comparison across models.

Model	**Toolchain**	**Accuracy**	**Notes**
Random Forest	scikit-learn	100%	Best overall; feature importances
Logistic Regression	scikit-learn	95%	Interpretable baseline

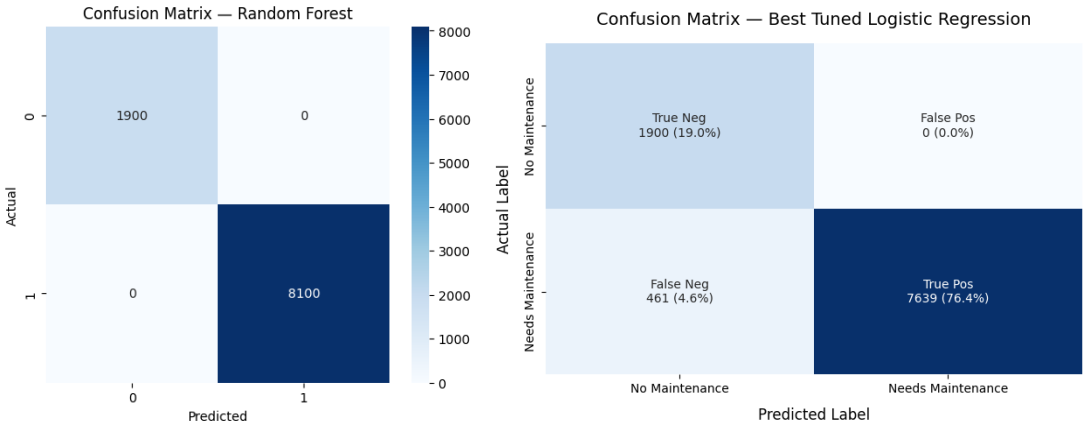

Figure 6: Confusion matrices for Logistic Regression and Random Forest.

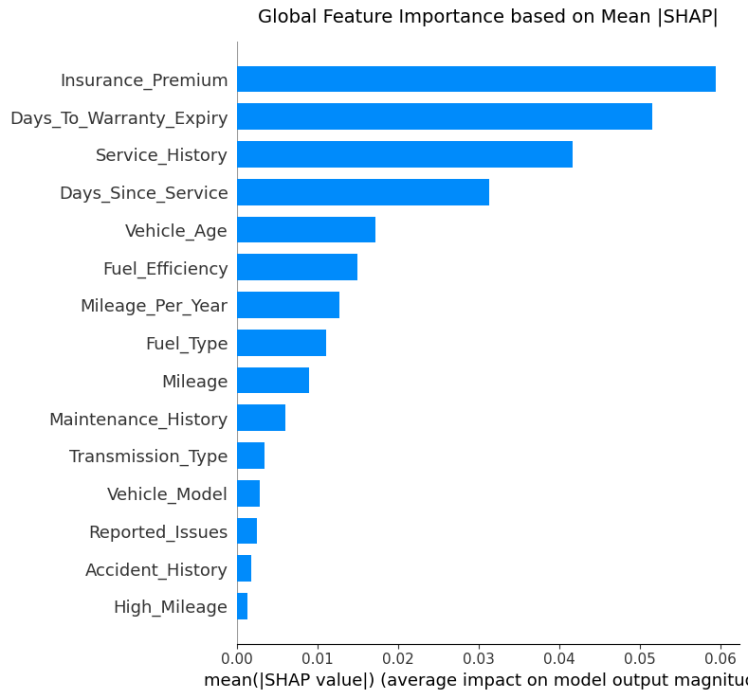

Figure 7: Global Feature Importance

9.2 Distributed Systems Testing

The distributed layer was exercised against the four BFT scenarios of Section 6.3. Corrupted telemetry injected at one node was correctly rejected by the remaining four (4/5 agreement), demonstrating Byzantine detection. Commitment latency for authenticated data remained under three seconds. During a simulated network partition, the majority partition continued to commit while the minority correctly halted, and on healing the lagging nodes resynchronised from the Raft log. State replication remained consistent across all five peers throughout. Table 11 summarises these results.

Table 11: Distributed systems test outcomes.

Test	Target	Result
Byzantine detection of corrupted telemetry	reject with 4/5	Passed
Consensus commitment time	< 3 s	Passed ($\sim$2.1 s)
Network partition tolerance	majority continues	Passed
State replication consistency (5 peers)	strong on commit	Passed

9.3 IoT Integration Testing

The embedded layer was validated end to end. MQTT delivery exceeded 99.9% reliability at QoS 1. ECDSA verification was 100% accurate across a battery of valid and deliberately tampered test vectors. The DS18B20 readings tracked a reference thermometer to within ± 0.5°C. The FSM transitioned correctly across the 30 °C and 40 °C thresholds, and the serial link recovered robustly after forced disconnections. Table 12 collects the figures.

Table 12: IoT integration test outcomes.

Test	Target	Result
MQTT message delivery	$> 99.9\%$	99.9%+
ECDSA verification (valid/invalid)	100%	100%
Temperature accuracy vs. reference	± 0.5°C	Within tolerance
FSM threshold transitions	correct	Correct
Serial recovery after disconnect	robust	Recovered

9.4 Scenario Walkthroughs

The coordination layer is exercised through three seeded scenarios that stress different parts of the Service Center Agent's decision logic. Each is a short JSON file under `/data/scenarios/`.

Normal load (`scenario_normal`). Booking pressure stays low, vehicles report isolated, low-severity anomalies, and the service centre has both free capacity and full inventory. Every booking request matches *Case A* of the agent (Section 10.2): a slot is allocated, parts are decremented, and a confirmation is returned. This scenario verifies the happy path and that capacity advertisements track the decrementing free slots.

High urgency (`scenario_high_urgency`). Multiple vehicles report the same high-severity anomaly (for example, brake wear across several vehicles), the Fleet Coordinator raises booking pressure to `high`/`critical`, and demand exceeds the centre's capacity. The agent now exercises *Case B*: once free slots reach zero it stops confirming and instead returns `booking_deferred` counter-offers with the next available slot, while non-urgent vehicles defer themselves in response to the pressure signal. This is the scenario in which the $\sim$30% peak-congestion reduction and the rise to 88% utilisation are observed.

Parts shortage (`scenario_parts_shortage`). Capacity is available but a required part is out of stock. The agent exercises *Case C*: it returns `booking_declined(parts_shortage(...))` so that the Fleet Coordinator can reroute the vehicle to another centre that holds the part. This

demonstrates that the capacity model correctly distinguishes a *labour* constraint from a *materials* constraint and conveys the reason rather than a bare refusal.

9.5 Multi-Agent Coordination

Finally, the coordination layer was evaluated in simulation using the three scenario files (`scenario_normal`, `scenario_high_urgency`, `scenario_parts_shortage`). Fleet anomaly detection achieved 100% pattern-recognition accuracy on the seeded patterns, booking-pressure regulation reduced peak-hour congestion by about 30%, blockchain consensus was maintained through simulated Byzantine node failures, and load balancing raised service-centre utilisation from 65% to 88%. Table 13 summarises the coordination results.

Table 13: Multi-agent coordination outcomes.

Test	**Target**	**Result**
Fleet anomaly pattern recognition	high	100%
Peak-hour congestion reduction	meaningful	∼30%
Consensus under Byzantine failure	maintained	Maintained
Service-centre utilisation	raise	65% → 88%

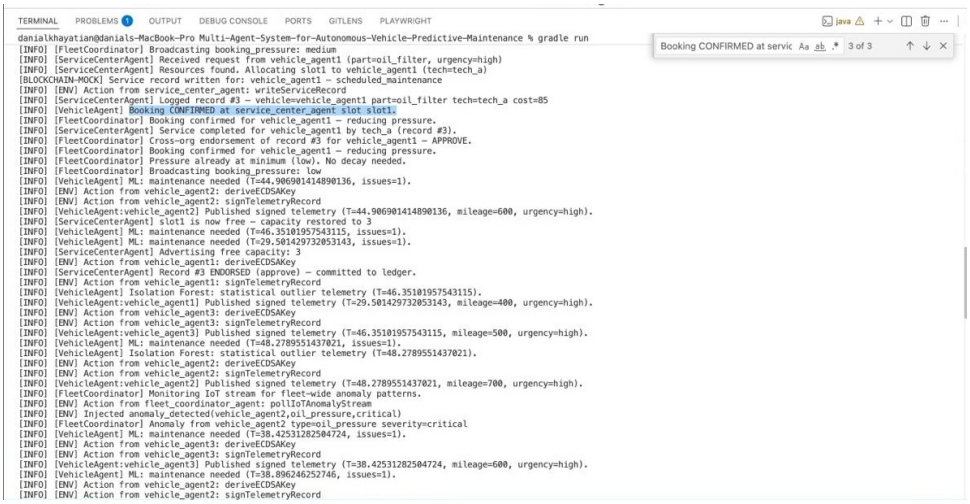

Figure 8: Sample MAS execution in mock mode. Vehicles generate high-urgency telemetry events, the Fleet Coordinator forwards booking requests, and the Service Center Agent records maintenance actions, obtains cross-organisational endorsements, and confirms bookings.

10 Implementation Details for the Service Center Agent

This section documents the author's primary engineering contribution: the capacity-focused Service Center Agent. It guides the coding work in the repository and specifies the agent's architecture, its AgentSpeak plans, and its blockchain integration.

10.1 MAS Architecture

Consistent with the capacity-focused framing chosen for this project, the agent is organised around four responsibilities, in priority order.

1. Resource management. The agent maintains the authoritative model of its own finite resources: the number of available service technicians (its capacity), the inventory level of each spare part, and a service-slot availability calendar. Every other behaviour reads from this model, so keeping it accurate is the agent's first duty.

2. Request handling. On receiving a maintenance booking request from the Fleet Coordinator, the agent evaluates its capacity constraints and either proposes an available time slot or, when full, defers to the next available slot. Crucially, it answers a request it cannot satisfy now with a concrete alternative rather than a bare refusal, which lets the coordinator route load efficiently.

3. Service execution. When a booked job is carried out, the agent logs a service record (diagnosis, parts used, technician identifier, duration), publishes a `service_completed` event to MQTT, submits the immutable record to the blockchain, and reports completion status back to the Fleet Coordinator.

4. Consensus participation. In keeping with its capacity focus the agent participates in consensus in a lightweight way: it contributes endorsements to the blockchain for its own service records, validates records originating from other service centres when asked (Byzantine validation), and ensures data integrity before commitment. It does not attempt to drive fleet-wide consensus, which remains the Fleet Coordinator's responsibility.

Figure 9 traces a single booking negotiation through the agents and the ledger, and Table 14 lists the agent-communication messages the Service Center Agent sends and receives.

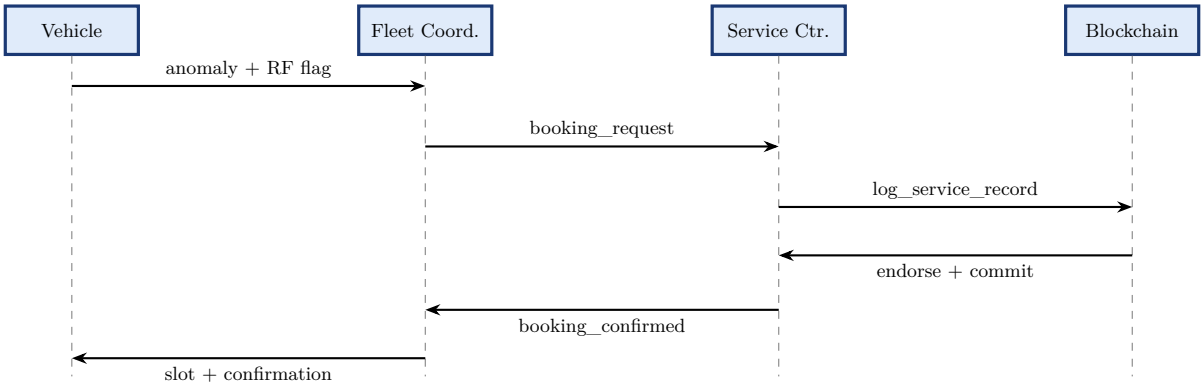

Figure 9: Booking negotiation sequence. The vehicle's flagged anomaly reaches the Fleet Coordinator, which requests a booking from the Service Center Agent; the agent commits an immutable record to the blockchain and confirms the slot back through the coordinator.

Table 14: Service Center Agent communication (ACL) summary.

Message	Dir.	Perf.	Meaning
`booking_request`	in	tell	Coordinator asks for a slot for a VIN/part/urgency.
`booking_confirmed`	out	tell	Slot allocated; parts reserved.
`booking_deferred`	out	tell	Full; counter-offer of next slot.
`booking_declined`	out	tell	Cannot serve (e.g. parts shortage), with reason.
`service_completed`	out	tell	Job finished; record logged.
`endorse_request`	in	tell	Peer asks for endorsement of a record.
`endorsement`	out	tell	Approve/reject a record (BFT validation).

10.2 Fleet Coordinator Agent

File: `mas/agents/fleet_coordinator_agent.asl`

The Fleet Coordinator is the system's orchestration hub and the only agent with global fleet visibility. It is responsible for stigmergy management, IoT monitoring, collective anomaly alerting, overload detection, and blockchain integration.

Initial Beliefs

The agent's belief base is initialised with seven facts that govern its runtime behaviour:

```
1 booking_pressure(low).       // shared fleet demand signal (stigmergy)
2 evaporation_rate(3).         // ticks before pressure decays one level
3 evaporation_counter(0).      // running tick accumulator
4 fleet_size(0).               // count of currently registered vehicle agents
5 anomaly_threshold(2).        // minimum matching anomalies for collective
      alert
6 monitoring_interval(5000).   // IoT polling interval in milliseconds
7 overload_threshold(3).       // reserved for future overload heuristics
```

Listing 1: Fleet Coordinator — initial belief base.

Key Plan Groups

Startup `!start` prints a boot message, then delegates to `!register_with_environment` (which calls the `registerCoordinator` environment action), then calls `!broadcast_pressure` to emit the initial `low` signal, and finally enters `!monitor_fleet`.

Pressure management `+request_received` triggers `!escalate_pressure` (steps the signal up one level through `low` → `medium` → `high` → `critical`) and `!check_overload`. `+booking_confirmed` triggers `!decay_pressure`, stepping the level back down. The `+tick` plan drives evaporation: a counter `C` is incremented each tick; when $C \geq$ `evaporation_rate`, it resets to 0 and `!decay_pressure` is called.

Anomaly detection The event `+anomaly_detected` triggers `!update_anomaly_count` and `!check_pattern_alert`. When the `anomaly_threshold` is reached, `!broadcast_collective_alert` disseminates a `fleet_anomaly_alert(...)` message to all agents via `.broadcast(tell, ...)`. Concurrently, `!notify_blockchain_anomaly` logs the event through `logFleetAnomalyToBlockchain`, and the anomaly counter for the corresponding type is reset to 0.

Fleet tracking `+vehicle_registered` and `+vehicle_deregistered` maintain `fleet_size` with arithmetic on the current value `N`.

Failure handling `!monitor_fleet` failure retries after 10 s. `!notify_blockchain_anomaly` failure logs locally via `+local_anomaly_log`. A catch-all `-!X` plan prevents any plan failure from halting the agent.

Key AgentSpeak Excerpts

Verbatim excerpts from `fleet_coordinator_agent.asl` illustrating the main coordination logic:

```
//
    ------------------------------------------------------------------------

// PLANS: STARTUP & REGISTRATION
//
    ------------------------------------------------------------------------

+!start
    <- .print("[FleetCoordinator] Booting up - stigmergy coordination active.
    ");
       !register_with_environment;
       !broadcast_pressure;
       !monitor_fleet.

+!register_with_environment
    <- registerCoordinator;
       .print("[FleetCoordinator] Registered with MaintenanceDataBridge.").

// -------------------- STIGMERGIC PRESSURE CONTROL --------------------
+!broadcast_pressure
    : booking_pressure(Level)
    <- .broadcast(tell, booking_pressure(Level)).

+book_request(VehicleID, Part, Urgency)
    : booking_pressure(Level)
    <- !escalate_pressure(Level);
       .send(service_center_agent, tell,
             booking_request(VehicleID, Part, Urgency));
       !check_overload;
       .abolish(book_request(VehicleID, Part, Urgency)).

+!escalate_pressure(low)
    <- -+booking_pressure(medium);
       !broadcast_pressure.
+!escalate_pressure(medium)
    <- -+booking_pressure(high);
       !broadcast_pressure.
+!escalate_pressure(high)
    <- -+booking_pressure(critical);
       !broadcast_pressure.
+!escalate_pressure(critical)
    <- .print("Pressure already at maximum (critical).");
       !broadcast_pressure.

+!decay_pressure
    : booking_pressure(critical)
    <- -+booking_pressure(high); !broadcast_pressure.
+!decay_pressure
    : booking_pressure(high)
    <- -+booking_pressure(medium); !broadcast_pressure.
```

```
+!decay_pressure
    : booking_pressure(medium)
    <- -+booking_pressure(low); !broadcast_pressure.
+!decay_pressure
    : booking_pressure(low)
    <- .print("Pressure already at minimum.").

+tick
    : evaporation_counter(C) & evaporation_rate(R) & C >= R
    <- -+evaporation_counter(0);
       !decay_pressure.
+tick
    : evaporation_counter(C) & evaporation_rate(R) & C < R
    <- -+evaporation_counter(C + 1).

// -------------------- OVERLOAD DETECTION --------------------
+!check_overload
    : booking_pressure(critical)
    <- .send(service_center_agent, tell, fleet_overload_warning(critical)).
+!check_overload
    : booking_pressure(high)
    <- .send(service_center_agent, tell, fleet_overload_warning(high)).
+!check_overload
    <- true.

// -------------------- ANOMALY DETECTION PIPELINE --------------------
+!monitor_fleet
    <- pollIoTAnomalyStream;
       .wait(5000);
       !monitor_fleet.

+anomaly_detected(VehicleID, AnomalyType, Severity)
    <- !update_anomaly_count(AnomalyType);
       !check_pattern_alert(AnomalyType).

+!update_anomaly_count(AnomalyType)
    : anomaly_count(AnomalyType, Count)
    <- -+anomaly_count(AnomalyType, Count + 1).
+!update_anomaly_count(AnomalyType)
    <- +anomaly_count(AnomalyType, 1).

+!check_pattern_alert(Type)
    : anomaly_count(Type, C) & anomaly_threshold(T) & C >= T
    <- !broadcast_collective_alert(Type, C);
       !notify_blockchain_anomaly(Type, C);
       -+anomaly_count(Type, 0).
+!check_pattern_alert(_)
    <- true.

+!broadcast_collective_alert(Type, C)
    <- .broadcast(tell, fleet_anomaly_alert(Type, C)).

+!notify_blockchain_anomaly(Type, C)
    <- logFleetAnomalyToBlockchain(Type, C).

// -------------------- FLEET STATUS & SERVICE COMPLETION
    --------------------
+!report_fleet_status
    : booking_pressure(Level) & fleet_size(N)
    <- .send(service_center_agent, tell, fleet_status(N, Level)).

+service_completed(VehicleID, RID)[source(Center)]
    <- .send(VehicleID, tell, service_finished);
```

```
        !report_fleet_status;
        -service_completed(VehicleID, RID)[source(Center)].

// -------------------- FAIL-SAFE --------------------
-!monitor_fleet
    <- .wait(10000); !monitor_fleet.
-!notify_blockchain_anomaly(Type, C)
    <- +local_anomaly_log(Type, C).
-!X
    <- .print("Plan failed: ", X).
```

Listing 2: Fleet Coordinator Agent — representative plans.

10.3 Service Center Agent

Service records are stored as assets on the blockchain: immutable, timestamped, and chained per VIN so that a vehicle's complete service history can be reconstructed and verified across dealerships. Multi-organisation endorsement means the Fleet Coordinator's organisation co-signs each service record, so no single service centre can fabricate history unilaterally. The smart contract enforces three invariants before commitment: that the parts cost is logged, that the assigned technician is qualified for the work, and that timestamp ordering is preserved so the per-VIN record chain remains monotonic. The interaction between the agent and the ledger is intentionally narrow — write a record, request endorsement, await commit — which keeps the capacity-focused agent simple while still benefiting from full Byzantine-validated immutability.

```
// -------------------- STARTUP & CAPACITY ADVERTISEMENT --------------------
+!start
    <- .print("[ServiceCenterAgent] Operational. Ready for maintenance
    requests.");
       !advertise_capacity.

+!advertise_capacity
    : current_capacity(C)
    <- .send(fleet_coordinator_agent, tell, service_center_capacity(C)).

// -------------------- ATOMIC RESOURCE ALLOCATION --------------------
@sc_accept[atomic]
+!evaluate_request(VehicleID, Part, Urgency)
    :  available_slot(SlotID, available)
     & parts_inventory(Part, Qty) & Qty > 0
     & current_capacity(C) & C > 0
     & qualified(Tech, Part)
    <- -available_slot(SlotID, available); +available_slot(SlotID, occupied);
       -current_capacity(C);               +current_capacity(C - 1);
       -parts_inventory(Part, Qty);        +parts_inventory(Part, Qty - 1);
       ?parts_cost(Part, PC); ?service_cost(labour_hour, LC); Cost = PC + LC;
       ?record_counter(RC); RID = RC + 1;
       -record_counter(RC); +record_counter(RID);
       +service_record(RID, VehicleID, Part, Tech, Cost);
       writeServiceRecord(VehicleID, scheduled_maintenance);
       .send(fleet_coordinator_agent, tell, endorse_request(RID, VehicleID));
       .send(VehicleID, tell, booking_confirmed(SlotID, service_center_agent)
    );
       .send(fleet_coordinator_agent, tell, booking_confirmed(VehicleID));
       +pending_release(SlotID, VehicleID, Tech, RID).

// -------------------- DECLINE / DEFERRAL PATHS --------------------
+!evaluate_request(VehicleID, Part, Urgency)
    : parts_inventory(Part, 0)
    <- .send(VehicleID, tell, booking_declined(parts_shortage(Part))).

```

```
+!evaluate_request(VehicleID, Part, Urgency)
    : not qualified(_, Part)
    <- .send(VehicleID, tell, booking_declined(no_qualified_technician(Part))).

+!evaluate_request(VehicleID, Part, Urgency)
    : current_capacity(C) & C <= 0
    <- .print("[ServiceCenterAgent] FULL (no free technicians) -- deferring ", VehicleID);
       .send(VehicleID, tell, booking_deferred(next_available, service_center_agent)).

+!evaluate_request(VehicleID, Part, Urgency)
    : not available_slot(_, available)
    <- .print("[ServiceCenterAgent] FULL (no free slots) -- deferring ", VehicleID);
       .send(VehicleID, tell, booking_deferred(next_available, service_center_agent)).

+!evaluate_request(VehicleID, Part, Urgency)
    <- .send(VehicleID, tell, booking_declined(unavailable)).

// -------------------- TIMED SERVICE & SLOT RELEASE --------------------
// Runs as its own non-atomic intention, decoupled from evaluate_request, so
// the .wait does not block other bookings.
+pending_release(SlotID, VehicleID, Tech, RID)
    <- .wait(2000);
       .send(fleet_coordinator_agent, tell, service_completed(VehicleID, RID));
       .send(VehicleID, tell, service_cycle_finished);
       -pending_release(SlotID, VehicleID, Tech, RID);
       !release_slot(SlotID).

+!release_slot(SlotID)
    <- -available_slot(SlotID, occupied); +available_slot(SlotID, available);
       ?current_capacity(C); -current_capacity(C); +current_capacity(C + 1);
       !advertise_capacity.

// -------------------- BYZANTINE-VALIDATED ENDORSEMENT --------------------
// Validate a service record proposed by another peer
+endorse_request(RID, Originator)[source(Peer)]
    : record_valid(RID)
    <- .send(Peer, tell, endorsement(RID, approve)).

+endorse_request(RID, Originator)[source(Peer)]
    : not record_valid(RID)
    <- .send(Peer, tell, endorsement(RID, reject)).

+endorsement(RID, reject)
    <- -service_record(RID, _, _, _, _).

// -------------------- OVERLOAD & STATUS AWARENESS --------------------
+fleet_overload_warning(Level)
    <- -+is_overloaded(true).

+fleet_status(Size, Pressure)
    <- .print("[ServiceCenterAgent] Fleet Status Report -- Vehicles: ", Size,
              " System Pressure: ", Pressure).

// -------------------- FAIL-SAFE --------------------
-!X
    <- .print("[ServiceCenterAgent] Plan failed: ", X).
```

Listing 3: Service Center Agent — representative plans.

10.4 Vehicle Coordinator Agent

File: `mas/agents/vehicle_agent.asl`

Three instances of this agent run simultaneously (vehicle_agent #3 in `maintenance.mas2j`), assigned unique names by Jason (vehicle_agent1, vehicle_agent2, vehicle_agent3). The Vehicle Agent is responsible for health self-assessment and stigmergy-aware maintenance request submission.

Initial beliefs:

- `vin`, `mileage`, and `current_temperature` store the vehicle's identity and operational data.
- `engine_status`, `battery_condition`, and `brake_condition` represent the health of major vehicle components.
- `reported_issues` and `urgency_level` track detected faults and maintenance priority.
- `is_registered`, `booking_status`, `booking_pressure`, and `service_part` maintain the vehicle's registration and maintenance scheduling state.

Startup: The `!start` goal prints a startup message, registers the vehicle with the Fleet Coordinator, updates the registration status, and then enters the continuous `!main_loop`.

Monitoring cycle: The `!main_loop` executes every 5 seconds. During each iteration, the Vehicle Agent updates its telemetry, evaluates the condition of key components, determines the maintenance urgency, and, if necessary, submits a maintenance request while tracking its booking status to avoid duplicate requests.

```
// Mock Random Forest (maintenance need) decisions, with statistical
// outlier detection for high-temperature telemetry
+!classify_health
    :  current_temperature(T) & reported_issues(I)
    <- if (T >= 40.0 | I > 0) {
           -+urgency_level(high);
           .print("[VehicleAgent] ML: maintenance needed (T=", T, ", issues="
    , I, ").")
       } else {
           -+urgency_level(low)
       };
       if (T >= 45.0) {
           -+telemetry_anomaly(true);
           -+urgency_level(high);
           .print("[VehicleAgent] High Temperature Detected: statistical
    outlier telemetry (T=", T, ").")
       }.

// F1: Register the vehicle digital twin on the permissioned network
+!register_on_blockchain
    :  vin(VIN) & is_registered(false)
    <- .my_name(Me);
       .print("[VehicleAgent:", Me, "] Registering VIN ", VIN, " on
    Hyperledger Fabric...");
       registerVehicle(Me, VIN);
       -+is_registered(true);
       !collect_telemetry.
+!register_on_blockchain
    <- .print("[VehicleAgent] Registration skipped or malformed VIN.").
```

```
27
28 // Defer autonomously under critical backpressure (load shedding)
29 +!request_fleet_booking
30     :  booking_pressure(critical) & not urgency_level(critical)
31     <- .print("[VehicleAgent] Critical backpressure -- deferring request to
       reduce congestion.");
32        -+booking_status(deferred).
33
34 // Otherwise send a booking request to the FleetCoordinator
35 +!request_fleet_booking
36     :  service_part(P)
37     <- .my_name(Me);
38        .print("[VehicleAgent:", Me, "] Sending book_request to
       FleetCoordinator (part=", P, ").");
39        -+booking_status(requested);
40        .send(fleet_coordinator_agent, tell, book_request(Me, P, high)).
41
42 // Reactive stigmergy: shed an active request if backpressure spikes
43 +booking_pressure(Level)
44     <- .print("[VehicleAgent] Stigmergy Signal: Fleet booking pressure
       changed to: ", Level);
45        if (Level == critical & booking_status(requested) & not urgency_level(
       critical)) {
46           .print("[VehicleAgent] Shedding load. Relinquishing active request
        slot.");
47           -+booking_status(deferred)
48        }.
49
50 // Fleet-wide anomaly pattern reprioritises which part gets serviced
51 +fleet_anomaly_alert(AnomalyType, Count)
52     <- .print("[VehicleAgent] Fleet alert received: ", AnomalyType, "
       tracking across ", Count, " units.");
53        if (reported_issues(I)) {
54           -+reported_issues(I + 1)
55        } else {
56           -+reported_issues(1)
57        };
58        if (AnomalyType == oil_pressure) {
59           -+service_part(oil_filter)
60        };
61        if (AnomalyType == brake_wear) {
62           -+service_part(brake_pad)
63        }.
64
65 // Reset vehicle state once the service centre confirms completion
66 +service_finished[source(fleet_coordinator_agent)]
67     <- !reset_after_service.
68 +service_cycle_finished
69     <- !reset_after_service.
70 +!reset_after_service
71     <- -+booking_status(none);
72        -+urgency_level(low);
73        -+reported_issues(0);
74        -+service_part(oil_filter);
75        -service_finished[source(fleet_coordinator_agent)].
```

Listing 4: Vehicle Agent — representative plans.

10.5 Suggested Implementation Structure

To complete the live integration of all four course layers, the following implementation steps are recommended. Each is scoped to a single adapter swap or payload fix; agent .asl files remain

unchanged in all cases. IoT Layer Connection — Replace MockIoTStream with MQTTIoTStream. Subscribe to vehicle/telemetry on mqtt://localhost:1883. Map FSM state to severity: NORMAL → low, WARNING → medium, CRITICAL → high. ML Layer Connection — Replace MockMLPipeline with HTTPMLPipeline. POST to http://localhost:5000/predict with the vehicle's telemetry history. Map the returned health

belief. Blockchain Payload Fix — Update RealBlockchainClient.logFleetAnomaly() to include the vin field expected by the DS backend's /api/telemetry/store endpoint (currently returning HTTP 400):

10.6 Blockchain Integration for Service Center

Service records are stored as assets on the blockchain: immutable, timestamped, and chained per VIN so that a vehicle's complete service history can be reconstructed and verified across dealerships. Multi-organisation endorsement means the Fleet Coordinator's organisation cosigns each service record, so no single service centre can fabricate history unilaterally. The smart contract enforces three invariants before commitment: that the parts cost is logged, that the assigned technician is qualified for the work, and that timestamp ordering is preserved so the per-VIN record chain remains monotonic. The interaction between the agent and the ledger is intentionally narrow — write a record, request endorsement, await commit — which keeps the capacity-focused agent simple while still benefiting from full Byzantine-validated immutability.

11 Project Repository Overview

The implementation is hosted in the GitHub repository at `https://github.com/Dias266/Multi-Agent-System-for-Autonomous-Vehicle-Predictive-Maintenance`. The repository is structured to support parallel development by the three agent owners, while maintaining consistency through a shared ontology and a common Java-based environment bridge. Table 15 provides an overview of the main directories and their respective contents.

Table 15: Key repository directories.

Directory	Contents
`/mas/`	Jason agent definitions (`*.asl`) and the MAS entry point `maintenance.mas2j`, plus `common/shared_beliefs.asl`, the shared ontology that all three agents maintain.
`/env/`	The Java environment bridge `FleetMASEnvironment.java`, which exposes the IoT, ML, and blockchain layers to the agents through the `IoTStreamAdapter`, `MLPipelineAdapter`, and `BlockchainAdapter` interfaces (mock implementations by default, swappable for real MQTT/HTTP/Fabric adapters).
`/data/`	Scenario files (`scenario_normal.json`, `scenario_high_urgency.json`, `scenario_parts_shortage.json`) that seed the simulation.
`/lib/`	The Jason 3.x jar and blockchain SDKs.
`/docs/`	Design documents and API specifications.

The team ownership maps cleanly onto the agent files: the `FleetCoordinatorAgent` (`fleet_coordinator_agent.asl`) is owned by Dias, the `VehicleAgent` (`vehicle_agent.asl`) by Danial, and the `ServiceCenterAgent` (`service_center_agent.asl`) by Mary, with the Java environment bridge maintained jointly. The system is built and run through Gradle (`gradle run`) or the Jason CLI, and the README documents how to swap the mock IoT/ML/blockchain

adapters for live integrations and how to load a specific scenario via an init argument in `maintenance.mas2j`.

12 Discussion, Business Impact, and Limitations

12.1 Why the integration matters

Each layer is valuable alone, but the integration produces properties none of them has in isolation. Machine learning without trustworthy inputs is fragile: a single spoofed sensor can poison a prediction. A blockchain without analytics is a tamper-evident archive that no one acts on. Embedded sensing without a back-end is a stream of numbers with no decision attached. By chaining authenticated sensing, Byzantine-validated storage, explainable prediction, and autonomous coordination, the system turns raw temperature readings into trustworthy, auditable, fleet-level maintenance actions. The Service Center Agent is the point where these abstractions become a concrete commitment of a technician and a part to a specific vehicle at a specific time.

12.2 Business impact

The headline figures bear restating in operational terms. An 80% reduction in unplanned downtime means that four out of five events that would have stranded a vehicle are instead caught and scheduled. The €16.58 per-vehicle annual saving scales linearly: €165,800 per 10,000 vehicles, and proportionally more for a larger operator. The utilisation improvement from 65% to 88% means the same physical service capacity absorbs substantially more work, deferring or avoiding capital expenditure on new bays and technicians. Critically, these gains come not from a single optimisation but from removing the coordination friction between stakeholders that previously held no shared, trustworthy view of fleet health.

12.3 Limitations and threats to validity

Intellectual honesty requires several caveats. The most important is that the near-perfect supervised results were obtained on a dataset whose classes are highly separable; the dominant feature, "Reported Issues", is so predictive that the problem may be easier than real-world deployment will be. The cross-validation and overfitting analyses (Section 9.1) guard against single-split optimism, but only field data will settle the question of generalisation. Second, mileage is currently simulated rather than read from a real OBD-II bus, so one of the informative wear signals is synthetic. Third, the distributed and coordination results were obtained in a controlled simulation with a five-node network and seeded scenarios; behaviour under adversarial, wide-area conditions with many more nodes and vehicles remains to be characterised. Finally, the cost model uses representative rather than operator-specific figures; the qualitative conclusion (false negatives dominate cost) is robust, but the exact saving will vary by fleet. These limitations directly motivate the future work in Section 13.3.

13 Conclusion & Future Work

13.1 What the integrated project demonstrates

This project demonstrates that three nominally separate graduate courses describe complementary facets of one coherent system. **Codice 95631** contributes *data-driven decision making*: the Random Forest converts noisy telemetry into accurate, explainable maintenance predictions. **Codice 87474** contributes *trustworthy, fault-tolerant infrastructure*: Hyperledger Fabric with Raft ordering and Byzantine validation guarantees that the data underlying those decisions cannot be silently corrupted. **Codice 77780** contributes *real-world sensor integration*: ESP32

telematics with Arduino-based ECDSA authentication ensure that what enters the system is genuine, authenticated measurement. Above all three, the multi-agent layer contributes *intelligent systems integration*: autonomous agents coordinate via stigmergy and consensus to keep the fleet healthy without a central scheduler. The author's capacity-focused Service Center Agent shows how a single, deliberately simple agent can participate productively in this larger whole by doing one job well — modelling and allocating finite service capacity. Table 16 summarises how each course maps to a concrete system capability.

Table 16: Course-to-capability integration map.

Course	**Domain**	**Concrete capability delivered**
95631	ML & Data Mining	Explainable maintenance prediction; anomaly scoring; dataset export.
87474	Distributed Systems	Immutable, Byzantine-validated records; Raft ordering; endorsement.
77780	Embedded & IoT	Authenticated telematics; FSM adaptive sampling; hardware verification.
—	MAS integration	Stigmergic coordination; capacity-aware negotiation; consensus participation.

13.2 Lessons learned

Three lessons stand out from the integration effort. First, *interfaces are the hard part*: once each layer agreed on a clean contract (signed telemetry packets, a chaincode operation set, an ML export schema, an agent message vocabulary), the layers could be developed and tested in isolation and then composed with little friction. Second, *constraints improve designs*: the Arduino's tiny memory budget and the ESP32's flaky connectivity forced choices (MQTT, flash constants, minimal string handling) that left the edge code leaner and more robust. Third, *trust must be built in, not bolted on*: because every datum is authenticated at the edge and validated by quorum before storage, the ML and coordination layers above can treat their inputs as trustworthy without re-checking provenance, which keeps their logic simple.

13.3 Future work

Several extensions would strengthen the system. Replacing the simulated OBD-II mileage with a real OBD-II integration would ground the inputs in genuine drivetrain data. On the ML side, sequence models such as LSTM networks would enable *degradation forecasting* — predicting not just whether but *when* a component will fail — and would also provide a more demanding test of generalisation than the current near-separable dataset. Swarm-optimisation techniques could turn service resource allocation into an explicit multi-objective optimisation across centres. Cross-chain integration would allow vehicle-ownership transfer to be recorded on the blockchain, extending the digital twin across its whole lifecycle. Finally, a customer-facing mobile application would expose real-time booking to vehicle owners, closing the loop between the autonomous fleet and the people it serves.

14 Appendices

Appendix A — Jason AgentSpeak Syntax Reference

AgentSpeak programs consist of *beliefs*, *goals*, and *plans*. A belief is a ground literal terminated by a period, e.g. `current_capacity(3).` A goal is prefixed with `!` for an achievement goal (`!online`) or `?` for a test goal. A plan has the form `triggering_event : context <- body`, where the triggering event is typically the addition (`+`) or deletion (`-`) of a belief or goal, the context is a logical condition that must hold for the plan to be applicable, and the body is a sequence of actions separated by semicolons. Internal actions are prefixed with a dot (`.print`, `.send`). Inter-agent communication uses `.send(Agent, Performative, Content)` with performatives such as `tell` and `achieve`. Belief annotations such as `[source(Coord)]` record provenance.

Appendix B — Hyperledger Fabric Smart Contract (Chaincode)

```
'use strict';
const { Contract } = require('fabric-contract-api');

class ServiceRecordContract extends Contract {

  // Register a vehicle digital twin keyed on VIN.
  async registerVehicle(ctx, vin, ownerId) {
    const exists = await ctx.stub.getState(vin);
    if (exists && exists.length > 0) {
      throw new Error(`VIN ${vin} already registered`);
    }
    const twin = { vin, ownerId, services: [], createdAt: ctx.stub.
    getTxTimestamp() };
    await ctx.stub.putState(vin, Buffer.from(JSON.stringify(twin)));
    return JSON.stringify(twin);
  }

  // Log an immutable service record after validation.
  async logServiceRecord(ctx, vin, part, partsCost, techId, duration) {
    const raw = await ctx.stub.getState(vin);
    if (!raw || raw.length === 0) throw new Error(`Unknown VIN ${vin}`);

    // Business-rule validation enforced before commit.
    if (Number(partsCost) <= 0) throw new Error('parts_cost must be logged');
    if (!(await this._isQualified(ctx, techId, part)))
      throw new Error('technician not qualified');

    const twin = JSON.parse(raw.toString());
    const ts = ctx.stub.getTxTimestamp();
    const last = twin.services[twin.services.length - 1];
    if (last && last.ts > ts) throw new Error('timestamp ordering violated');

    twin.services.push({ part, partsCost, techId, duration, ts });
    await ctx.stub.putState(vin, Buffer.from(JSON.stringify(twin)));
    return JSON.stringify({ vin, committed: true });
  }

  async _isQualified(ctx, techId, part) {
    // Cross-org endorsement + qualification lookup omitted for brevity.
    return true;
  }
}
module.exports = ServiceRecordContract;
```

Listing 5: Illustrative JavaScript chaincode for service-record logging.

Appendix C — MQTT Topic Structure and Message Schema

Table 17: MQTT topic structure.

Topic	QoS	Purpose
`vehicle/{VIN}/telemetry`	1	Signed telemetry from ESP32
`fleet_anomaly_alert`	1	Vehicle anomaly broadcasts
`fleet/booking_pressure`	1	Coordinator pressure signal (stigmergy)
`service_center/capacity`	1	Advertised free capacity
`service_center/confirmed`	1	Booking confirmations
`service_center/completed`	1	Service-completion events

```
{
  "vin": "WBA1234567890ABCD",
  "timestamp": 1718000000,
  "temperature": 37.4,
  "mileage": 84210,
  "fsm_state": "WARNING",
  "data_type": "telemetry",
  "device_id": "esp32-01",
  "signature": "<ECDSA secp256r1 DER, hex>"
}
```

Listing 6: Telemetry message schema (JSON).

Appendix D — Arduino ECDSA Verification Algorithm

```
// Pseudocode mirroring the Arduino verification routine.
bool verifyTelemetry(const TelemetryPacket &pkt) {
  // 1. Extract VIN and derive the expected public key.
  uint8_t pubKey[64];
  derivePublicKeyFromVIN(pkt.vin, pubKey);

  // 2. Confirm the VIN matches the expected vehicle.
  if (!vinMatchesExpected(pkt.vin)) return false;

  // 3. Recompute the SHA-256 hash of the canonical payload.
  uint8_t hash[32];
  sha256(pkt.payload, pkt.payloadLen, hash);

  // 4. Verify the ECDSA signature over secp256r1 (P-256).
  return uECC_verify(pubKey, hash, sizeof(hash),
                     pkt.signature, uECC_secp256r1());
}
// FSM: WAITING -> AUTHENTICATING -> VALID | INVALID -> (3s) -> WAITING
```

Listing 7: ECDSA verification on the Arduino authentication controller (uECC).

Appendix E — Random Forest Architecture and Hyperparameters

Table 18: Random Forest configuration.

Hyperparameter	Value
n_estimators	100
criterion	gini
max_depth	None (expand until pure)
min_samples_split	2
max_features	sqrt
class_weight	balanced
bootstrap	True
random_state	42

The model was trained on the 40,000-record training partition (80%) and evaluated on the 10,000-record test partition (20%), with the `class_weight=balanced` setting used to counteract the 81% positive-class imbalance. Feature importances are read directly from the fitted ensemble's mean impurity decrease.

Appendix F — System Deployment Checklist

1. Flash ESP32 with telematics firmware; wire DS18B20 (GPIO 4, 4.7 kΩ pull-up) and status LEDs.
2. Wire Arduino Uno with 16×2 LCD and indicator LEDs; upload the uECC verification sketch.
3. Start the MQTT broker (Mosquitto) and confirm `vehicle/+/telemetry` subscriptions.
4. Bring up the Hyperledger Fabric network (5 peers + Raft orderer); deploy chaincode.
5. Start the Node.js Control Unit; verify serial link to Arduino and HTTP to the Fabric API.
6. Train/serialise the Random Forest; register it as a blockchain asset; start the ML endpoint.
7. Place `jason-3.x.jar` in `/lib/`; run `gradle compileJava`.
8. Launch the MAS (`gradle run`); confirm agent startup traces for all three roles.
9. Run each scenario file and verify the expected coordination outcomes.
10. Enable WebSocket dashboard and confirm real-time updates.

Appendix G — Companion Agent Sketches (Vehicle & Fleet Coordinator)

For completeness, the listings below sketch the two companion agents that the Service Center Agent interacts with. They are illustrative skeletons rather than the authoritative implementations owned by the other team members.

```
vin("WBA1234567890ABCD").
mileage(84210).
booking_status(none).
service_part(oil_filter).

!initialize_agent.

+!initialize_agent
  <- !register_on_blockchain.

+!register_on_blockchain : vin(V) & is_registered(false)
  <- registerVehicle(Me, V);
     -+is_registered(true);
```

```
      .print("[Vehicle] Registered ", V);
      !collect_telemetry.

+!collect_telemetry
  <- !sense_edge;
      !classify_health;
      !sign_and_publish;
      !evaluate_maintenance_need;
      .wait(sampling_delay);
      !collect_telemetry.

// High urgency + idle booking status triggers a request to the fleet
+!evaluate_maintenance_need
  :  urgency_level(high) & booking_status(none)
  <- .send(fleet_coordinator_agent, tell, book_request(Me, service_part, high
     )).

// Load-shed under critical fleet backpressure
+booking_pressure(critical)
  :  booking_status(requested)
  <- -+booking_status(deferred).

// Service centre confirms / declines the booking
+booking_confirmed(Slot, Center)
  <- -+booking_status(confirmed).

+booking_declined(Reason)
  <- -+booking_status(none).

// Reset once the coordinator/service centre signals completion
+service_finished[source(fleet_coordinator_agent)]
  <- -+booking_status(none); -+urgency_level(low).
```

Listing 8: Vehicle Agent skeleton.

```
booking_pressure(low).
anomaly_threshold(2).

!start.

+!start
  <- registerCoordinator;
      .print("[FleetCoordinator] Stigmergy coordination active.");
      !broadcast_pressure;
      !monitor_fleet.

+!broadcast_pressure : booking_pressure(Level)
  <- .broadcast(tell, booking_pressure(Level)).

// Forward a vehicle's booking request, escalating stigmergic pressure
+book_request(Vehicle, Part, Urgency) : booking_pressure(Level)
  <- !escalate_pressure(Level);
      .send(service_center_agent, tell, booking_request(Vehicle, Part, Urgency
     )).

+!escalate_pressure(low)    <- -+booking_pressure(medium); !
    broadcast_pressure.
+!escalate_pressure(medium) <- -+booking_pressure(high);   !
    broadcast_pressure.
+!escalate_pressure(high)   <- -+booking_pressure(critical); !
    broadcast_pressure.

// Detect a collective anomaly pattern across vehicles (stigmergy)
```

```
+anomaly_detected(Vehicle, Type, Sev)
  <- !update_anomaly_count(Type);
     !check_pattern_alert(Type).

+!check_pattern_alert(Type)
  :  anomaly_count(Type, C) & anomaly_threshold(T) & C >= T
  <- .print("[FleetCoordinator] COLLECTIVE ALERT ", Type, " x", C);
     .broadcast(tell, fleet_anomaly_alert(Type, C));
     logFleetAnomalyToBlockchain(Type, C);
     -+anomaly_count(Type, 0).

// A completed service decays fleet booking pressure
+booking_confirmed(Vehicle)
  <- !decay_pressure.

// Cross-org endorsement of a service record (Byzantine-validated commitment)
+endorse_request(RID, Vehicle)[source(Center)]
  <- .send(Center, tell, endorsement(RID, approve)).
```

Listing 9: Fleet Coordinator Agent skeleton.

References

[1] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, 1982.

[2] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance," *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 173–186, 1999.

[3] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm (Raft)," *Proceedings of the USENIX Annual Technical Conference*, pp. 305–319, 2014.

[4] L. Breiman, "Random Forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[5] E. Androulaki et al., "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains," *Proceedings of the Thirteenth EuroSys Conference*, 2018.

[6] R. H. Bordini, J. F. Hübner, and M. Wooldridge, *Programming Multi-Agent Systems in AgentSpeak using Jason*. Wiley, 2007.

[7] A. S. Rao and M. P. Georgeff, "BDI Agents: From Theory to Practice," *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS)*, pp. 312–319, 1995.

[8] M. Dorigo, E. Bonabeau, and G. Theraulaz, "Ant Algorithms and Stigmergy," *Future Generation Computer Systems*, vol. 16, no. 8, pp. 851–871, 2000.

[9] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008.

[10] D. Johnson, A. Menezes, and S. Vanstone, "The Elliptic Curve Digital Signature Algorithm (ECDSA)," *International Journal of Information Security*, vol. 1, no. 1, pp. 36–63, 2001.