

Nunnarri: An AI-Enabled Universal IoT Management Platform Integrating MQTT, Local Language Models, and Model Context Protocol

Harshar A T¹, Jebakumar R², Pranav K³, Balaji V⁴

B.Tech. in Artificial Intelligence and Data Science, Sengunthar Engineering College, Tamil Nadu, India

Abstract- The growing use of heterogeneous Internet of Things (IoT) devices creates operational challenges in device onboarding, telemetry collection, control, data persistence, and integration with intelligent services. This paper presents Nunnarri, a modular IoT management platform designed to bring these functions into a single deployable system. The public implementation combines a React and Vite dashboard, Node.js and Express API services, an MQTT/WebSocket gateway, two PostgreSQL database services, an Ollama-connected AI backend, and a Model Context Protocol (MCP) server. The architecture separates identity and access data from IoT telemetry, provides a virtual-pin abstraction for device interaction, and exposes selected device-management operations to compatible AI clients through MCP tools, resources, and prompts. The system is packaged as a multi-container Docker Compose application and includes firmware integration material for ESP8266-class devices. The contribution of this work is an implementation-oriented architecture that connects conventional IoT messaging and web management with locally hosted language-model assistance and a standardized tool interface. The repository documents the intended system components; however, quantitative claims about latency, throughput, anomaly-detection quality, reliability, and resource use require controlled evaluation. A repeatable evaluation plan is therefore defined for message delivery, end-to-end command delay, service recovery, AI-assisted task completion, and security controls

Keywords— Internet of Things, MQTT, edge AI, local large language models, Ollama, Model Context Protocol, cyber-physical systems, Docker, device management

I. INTRODUCTION

IoT deployments commonly combine microcontrollers, sensors, actuators, communication protocols, dashboards, databases, and automation services. As a deployment grows, operators may need to coordinate multiple tools to register devices, inspect state, review telemetry, issue commands, and diagnose failures. A platform that centralizes these workflows can reduce operational fragmentation, while still preserving clear boundaries between messaging, application logic, persistence, and user interaction.

Nunnarri is an open-source implementation intended to address this integration problem. Its repository describes a management dashboard, an API backend, an MQTT and WebSocket gateway, separated PostgreSQL services for user and IoT data,

an AI backend connected to a locally hosted model through Ollama, and an MCP server that exposes selected functions to compatible AI applications. A virtual-pin model maps device state and control values into a consistent interface, and Docker Compose is used to coordinate the application services.

The paper focuses on architecture and implementation rather than claiming a novel learning algorithm. The main research question is: how can a modular IoT management platform combine protocol-based telemetry, local language-model assistance, and a standardized AI tool interface while keeping device management and data persistence in distinct services?

Objectives

- Design a service-separated architecture for device registration, telemetry, monitoring, and command dispatch.

- Provide an MQTT/WebSocket gateway to connect hardware nodes with application services.
- Integrate a locally hosted language model through an AI backend for conversational interaction and selected analysis workflows.
- Expose typed device-management operations through an MCP server for external AI clients.
- Define a reproducible evaluation protocol for functional correctness, performance, reliability, and security.

1. Contributions

This work presents:

- an implementation architecture that composes web, API, gateway, persistence, AI, and MCP services; (ii) a virtual-pin abstraction intended to normalize interactions with devices;
- a local-first AI integration path in which model inference can be hosted within the deployment environment;
- an evaluation plan that identifies the measurements needed to substantiate performance and reliability claims.

II. RELATED WORK AND TECHNICAL BACKGROUND

MQTT is a lightweight publish/subscribe messaging protocol widely used in constrained and networked IoT environments. Its topic-based communication model decouples data producers from consumers and is suitable for telemetry streams and asynchronous device commands [1]. WebSocket provides a persistent, bidirectional communication channel between a browser or service and a server, making it useful for interactive monitoring interfaces [2].

Containerization provides a repeatable packaging model for services and dependencies. Docker Compose allows multi-container applications to be described and started from a single configuration, while service boundaries permit components to be maintained independently [3]. PostgreSQL provides relational persistence and transaction support for identity, configuration, and telemetry metadata [4].

Locally hosted large language models can support natural-language interfaces without requiring each prompt and response to be sent to an external model provider. Such systems still require careful handling of authorization, tool validation, model uncertainty, latency, and the possibility that a model proposes an unsafe or incorrect action. MCP defines a protocol through which applications can expose tools, resources, and prompts to compatible AI clients [5]. Nunnarri combines these existing technologies into one implementation; the novelty claim is therefore limited to the platform integration and its concrete design choices, not to inventing MQTT, LLMs, or MCP.

III. SYSTEM REQUIREMENTS AND DESIGN

1. Functional requirements

- **Device lifecycle:** register and retrieve device metadata and current state.
- **Telemetry:** receive or publish measurements and retain historical records for later queries.
- **Control:** transmit device commands and update virtual-pin values through a common management layer.
- **Visualization:** provide a dashboard for devices, sensor data, monitoring, and control.
- **AI assistance:** route chat, analysis, and automation requests to a dedicated AI backend linked to a local model service.
- **AI interoperability:** expose supported operations as MCP tools and make selected system data available as resources and guided prompts.

Deployment: coordinate services and health checks in a containerized environment.

2. Non-functional requirements

The intended design emphasizes modularity, local deployment, service isolation, responsive interaction, maintainability, and controlled access to device actions. These are design objectives and must be validated by tests. The public repository should be reviewed for hard-coded credentials, permissive cross-origin settings, insecure broker configuration, and missing authorization checks before the

application is exposed outside a trusted development network.

IV. Proposed Architecture

The platform is composed of the following logical layers. Their stated ports and frameworks reflect the repository documentation and configuration; the actual deployment should be confirmed using the final configuration used for experiments.

1. Data and command flow

A device publishes telemetry to an MQTT topic managed by the gateway. The gateway provides the network entry point for messages, while backend routes and service logic coordinate device records, telemetry persistence, and dashboard updates. The frontend communicates with backend APIs and live gateway channels to represent device state and make user-initiated changes. An AI request is routed through the AI backend, which communicates with the locally configured model service and may use platform data to generate an answer or select a supported operation. The MCP server exposes defined operations to an MCP-compatible client; an invoked tool runs its handler and returns a textual or structured result. This separation allows device protocols and AI-client integration to evolve without requiring every function to reside in one process.

2. Virtual-pin abstraction

The repository documents a virtual-pin scheme, V0 through V255, intended to provide a consistent application-level representation of device inputs and outputs. A virtual pin can represent a sensor value, a Boolean actuator state, or another application-defined value. The abstraction can simplify dashboard widget binding and AI tool arguments because higher-level functions can refer to a logical pin instead of directly encoding a board-specific GPIO mapping. A production implementation should maintain an explicit device-to-pin schema, validate pin type and range, and reject unauthorized or unsafe commands.

3. Local AI integration

The AI backend is implemented as a separate Node.js/Express service with chat, analysis, and

automation route groups. The repository describes integration with Ollama and a Llama-family local model. In this architecture, local inference may reduce dependence on external inference services and can keep telemetry within the operator-controlled environment. These benefits depend on deployment settings, model choice, hardware resources, and whether external services are invoked elsewhere; they should not be interpreted as a measured latency or privacy guarantee.

4. Model Context Protocol interface

The MCP server uses the official JavaScript SDK and registers handlers for listing and invoking tools, reading resources, and retrieving prompts. Repository documentation lists operations for device discovery, registration, commands, virtual-pin access, telemetry, anomaly analysis, health checks, and related management functions. It also describes resources for device state, system health/statistics, and alert information, plus prompts for device diagnosis, actuator assistance, and environmental audits. The set of operations available at runtime should be verified from the built server, and write-capable tools should be gated by user authorization and explicit safety checks.

5. Persistence and deployment

The Docker Compose configuration defines separate PostgreSQL containers for user and IoT records and coordinates the application services. Separation can reduce accidental coupling between account data and high-volume telemetry, although both databases remain part of the same deployment and require backups, access control, credential rotation, and monitoring. Container health checks can help detect unavailable dependencies but are not a substitute for end-to-end application tests.

V. IMPLEMENTATION

The implementation described in the repository uses React 18 and Vite for the frontend; Node.js and Express for the main API and AI backend; PostgreSQL 15 for the two database services; Aedes for MQTT broker functionality; WebSocket communication for real-time interaction; Ollama for local model serving; and the Model Context Protocol JavaScript SDK for

the AI interoperability server. The project includes Dockerfiles, a Docker Compose configuration, example backend environment settings, MCP client configuration examples, a firmware sketch, and connection guides for ESP8266-based development.

1. Core service interfaces

The main backend groups API routes for authentication, devices, sensors, AI, widgets, and administration. The AI service groups endpoints under chat, analysis, and automation. The gateway is the messaging boundary for MQTT and WebSocket traffic. The MCP service wraps selected backend and gateway actions in typed tool definitions and exposes system information through resources. This division makes it possible to test service-level behavior independently and then perform integration tests across the message-to-database and command-to-device paths.

2. Security considerations

The codebase documents JWT-based authentication, role-based access control, per-device tokens, and optional GitHub OAuth integration. Their effectiveness must be confirmed by inspecting route middleware and executing negative tests. In particular, the version submitted for review should not include actual passwords, tokens, or environment files in version control. The public repository contains configuration artifacts that appear to include credential-like values; these should be treated as compromised, rotated, removed from tracked files, and moved to environment variables or a secrets manager. Device commands issued through AI tools should require authorization independent of model-generated text, validate the target device and pin, apply action-specific limits, and record an audit event.

VI. Evaluation Methodology

A research paper on an implemented system should distinguish architecture descriptions from empirical findings. The repository description alone does not establish quantitative performance. The following evaluation is proposed to produce reproducible measurements before submission.

1. Test environment

Record the CPU model and core count, RAM, operating system, Docker Engine and Compose versions, Node.js version, PostgreSQL image version, MQTT client settings, network type, Ollama version, model name and quantization, and whether hardware devices or a simulator were used. Report whether each test ran on one host or across hosts. Use the same configuration for repeated runs and publish scripts and raw logs where possible.

2. Functional tests

Device registration: test valid input, duplicate identifiers, malformed fields, and unauthorized requests.

Telemetry ingestion: publish known sensor values, confirm receipt at the gateway, and compare database records with source messages.

Virtual-pin control: test supported types and ranges, invalid pin identifiers, offline devices, and rejected commands.

- **MCP integration** :enumerate tools/resources/prompts, call representative read-only operations, and verify expected errors for
- invalid inputs.
- **AI workflow**: test natural-language requests against a fixed suite; judge whether responses are grounded in actual device state and
- whether control requests pass through authorization checks.
- **Recovery**: restart each service and database in turn, then verify reconnection, state recovery, and error reporting.

3. Performance metrics

Measure (i) telemetry end-to-end delay from publish timestamp to stored record or dashboard receipt; (ii) command response time from API request to device acknowledgement; (iii) throughput in messages per second under increasing offered load; (iv) message loss or duplication rate; (v) CPU and memory use by service; (vi) AI request latency and task success rate; and (vii) service recovery time after a controlled failure. Run each workload at least 30 times where feasible, report median and 95th percentile, and specify the sample size and exclusions. Do not label

the system “sub-millisecond” or “sub-10 ms” without measurements under a defined test setup.

VII. RESULTS

No validated benchmark results were available in the reviewed repository materials at the time this manuscript was drafted. Therefore, this section is intentionally presented as a measurement template rather than as a claim that the proposed tests have already succeeded. Populate the table only after running the tests. If an item is not implemented or tested, state “not evaluated” rather than estimating a value.

Table 2. Required measurements for a defensible experimental results section.

Replace “Not yet measured” with recorded results and provide the exact machine, workload, repetition count, and test scripts used.

VIII. DISCUSSION

The main design advantage of Nunnarri is the combination of service separation and a common management interface. MQTT handles asynchronous message transport, the backend centralizes application rules, PostgreSQL supports persistence, the frontend presents device state, and MCP provides a defined boundary between AI clients and platform operations. A locally hosted model can provide a natural-language interface while avoiding a mandatory cloud inference dependency. The virtual-pin abstraction may also reduce the effort required to map device-specific behavior into consistent application workflows.

These advantages are architectural hypotheses rather than demonstrated comparative improvements. The platform introduces operational

complexity because it contains multiple services and databases, and local model inference consumes compute resources. The MCP interface adds an additional security boundary: exposing tool definitions does not automatically make tool

execution safe. Incorrect model interpretations, stale sensor data, service unavailability, and malicious prompts could cause inappropriate actions if the server does not independently validate requests. Controlled experiments and security tests are necessary before drawing conclusions about usability, scalability, reliability, or safety.

XI. LIMITATIONS AND FUTURE WORK

This draft is based on public repository files and documentation rather than a reproducible deployment audit. Feature descriptions in the README may represent intended capabilities and do not prove that every workflow is complete, secure, or operational in a tested environment. No benchmark dataset, statistically summarized test run, comparative baseline, or externally validated security assessment was supplied. Further work should implement a repeatable telemetry workload generator, publish end-to-end latency distributions, test under concurrent device and client load, evaluate AI outputs on a labeled task set, and compare local AI assistance against a non-AI baseline. Additional work should include TLS-enabled MQTT, explicit per-tool authorization, secret scanning, structured audit logs, safe defaults for actuator commands, backup/restore tests, and documentation of supported hardware mappings.

X. Conclusion

This paper described Nunnarri, a modular IoT management platform that integrates web-based monitoring, a Node.js API, MQTT/WebSocket messaging, separated PostgreSQL persistence, a locally connected AI backend, and an MCP interface for AI-client interoperability. The implementation offers a foundation for managing device state and telemetry through a unified dashboard and exposing defined operations to compatible AI tools. At present, the strongest supportable contribution is the architecture and implementation design. Claims about latency, throughput, anomaly detection, security, and reliability remain open until measured under repeatable conditions. The next step is to execute the evaluation plan, publish the

configuration and raw results, and revise the paper to distinguish demonstrated results from intended capabilities.

Acknowledgment

The author acknowledges the open-source maintainers of the technologies used in the project, including React, Node.js, Express, PostgreSQL, MQTT/Aedes, Ollama, Docker, and the Model Context Protocol SDK. The final manuscript should verify versions and add any institutional or project acknowledgments required by the author.

REFERENCES

1. OASIS, "MQTT Version 5.0," OASIS Standard, Mar. 2019. [Online].
2. Available
: <https://docs.oasisopen.org/mqtt/mqtt/v5.0/mqtt-v5.0.html>
3. I. Fette and A. Melnikov, "The WebSocket Protocol," RFC 6455, Internet Engineering Task Force, Dec. 2011. doi: 10.17487/RFC6455. [Online]. Available: <https://www.rfc-editor.org/rfc/rfc6455> [3] Docker, "Docker Compose documentation." [Online]. Available: <https://docs.docker.com/compose/>
4. PostgreSQL Global Development Group, "PostgreSQL Documentation." [Online]. Available: <https://www.postgresql.org/docs/>
5. Model Context Protocol, "Specification and documentation." [Online].
6. Available: <https://modelcontextprotocol.io/>
7. Ollama, "Ollama documentation." [Online]. Available: <https://docs.ollama.com/>
8. H. A. T., "AI Universal IoT Management / Nunnarri," source-code repository. [Online]. Available:
<https://github.com/harshar007/AI-universal-IOT-mangement>. Accessed: Oct. 12, 2026.