

Article

Rule Graph-Based Low-Code Control for Renewable Energy and Storage Stations

Jiacheng Li ^{1,2,3}, Menghan Xiao ^{1,2,3,*}, Chang Ye ^{1,2,3}, Xun Xu ^{1,2,3} and Yuwei Gui ^{1,2,3}

¹ State Grid Hubei Electric Power Research Institute, Wuhan 430077, China; lijiaoch233@foxmail.com (J.L.); yech0131@163.com (C.Y.); xx_430@whu.edu.cn (X.X.); gyw20001228@163.com (Y.G.)

² Hubei Key Laboratory of Regional New Power Systems and Rural Energy System Configuration, Wuhan 430077, China

³ Hubei Engineering Research Center of the Construction and Operation Control Technology of New Power Systems, Wuhan 430077, China

* Correspondence: xiaomh12@163.com

Abstract

Renewable energy and energy storage stations require frequent updates of monitoring and control logic across heterogeneous devices and changing operating strategies. This paper proposes a rule graph-based reference architecture that combines low-code logic configuration, graph-model semantic binding, and microservice-oriented functional decomposition. A component status matrix separates the target architecture from the implemented subset. The runnable subset comprises a minimal FastAPI backend, REST/WebSocket telemetry interfaces, an in-process queue, and stateful rule evaluators; gateway, authentication, external message bus, time-series database, visual editor, and industrial protocol services remain design-level elements. Beyond the original single-rule example, a priority-ordered multi-device rule is implemented for cooperative BESS dispatch, communication/topology blocking, low-SOC protection, frequency-based load shedding, backup request, and five-sample recovery release. Existing local network benchmarks are complemented by a 600-step software-in-the-loop trace with scripted telemetry fluctuations and communication quality faults and by 500 in-process ASGI timing samples at each of the four point levels. The trace produced no safety dispatch or protected device violations. P99 application path latency ranged from 1.1962 to 5.5287 ms, but one 75.3065 ms outlier exceeded a 50 ms reference deadline, demonstrating that the Windows/FastAPI path is not deterministic. No industrial controller, hardware-in-the-loop facility, field data, or engineer usability study was used. Accordingly, the paper makes no claim of industrial real-time readiness or measured development effort reduction.

Keywords: renewable energy station; energy storage station; low-code control logic; rule graph; microservice reference architecture; graph-model mapping; software-in-the-loop; timing jitter; safety interlock



Academic Editor: Salvador Pérez Litrán

Received: 21 July 2026

Revised: 6 August 2026

Accepted: 18 August 2026

Published: 21 August 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction

Renewable and energy storage stations are evolving toward data-intensive and strategy-adaptive operation, where the control logic must be revised frequently and coordinated across heterogeneous devices [1–4]. Recent hierarchical control studies further illustrate the coordination burden created by hybrid AC/DC networks, renewable resources, and multiple storage devices [4]. In many projects, station logic is still delivered through manual coding and vendor-specific interface adaptation. This practice typically

leads to long development cycles, repeated integration effort, and the weak reusability of control artifacts [5,6].

Low-code and model-driven approaches are promising because they can reduce repetitive implementation work through configuration-oriented workflows and reusable abstractions [7–10]. The microservice architecture is also relevant in this domain because it supports the modular decomposition of access, orchestration, rule processing, and data handling functions [11–14]. However, the available evidence in many studies is either conceptual or scenario-specific, and the boundary between architecture design claims and field-validated performance claims is not always explicit.

This paper therefore adopts a conservative and auditable scope. We present a rule graph-based reference architecture and evaluate only its implemented local subset through network benchmarks, a complex cooperative control software-in-the-loop trace, and an application path timing consistency test. The study does not claim measured low-code productivity, industrial controller performance, hardware-in-the-loop validation, field deployment, or production readiness. Figure 1 summarizes this scope as a framework-level evidence chain. It starts from the engineering context and design strategy, links these to reference implementation artifacts, and separates local measured outputs from scripted software-in-the-loop scenario outputs. The field measurement branch shown in the framework is a planned extension for future validation; the reported results in this paper do not refer to field measurements. In Figure 1, the “synthetic evidence matrix” label refers only to generated disturbance inputs for the cooperative rule software-in-the-loop trace; synthetic development effort and component ablation results are not presented.

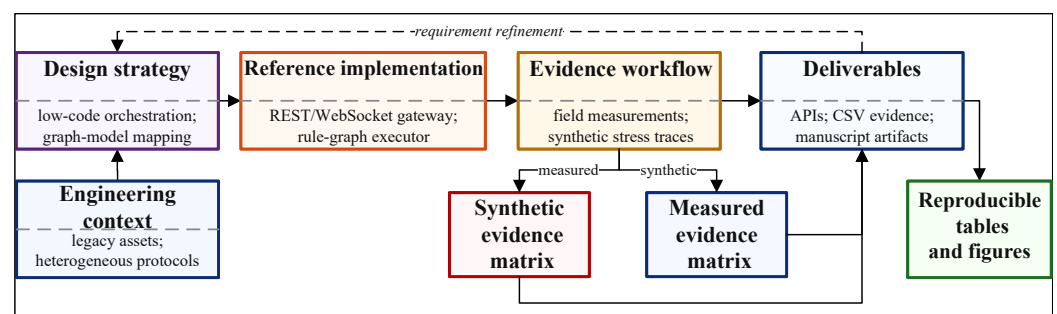


Figure 1. Overall research framework linking engineering context, design strategy, reference implementation artifacts, local measured and scripted scenario evidence outputs, and future field evidence extension. Solid arrows indicate the forward evidence workflow, whereas the dashed arrow indicates requirements-refinement feedback.

Contributions

The contributions, strictly aligned with available project evidence, are as follows:

1. A rule graph-based low-code microservice reference architecture with a component status matrix that distinguishes implemented/evaluated functions from design-level and future functions.
2. A typed rule graph and graph-model binding method with explicit unit, action, dependency, safety priority, cooperative dispatch, latch, and recovery semantics.
3. A minimal reference backend and API contract exposing REST/WebSocket telemetry ingestion and both single-rule and multi-device cooperative rule evaluation.
4. A reproducible local evaluation workflow containing network latency, resource use, and scalability benchmarks; a 600-step disturbed software-in-the-loop trace; and timing jitter measurements with raw samples.

5. A structured comparison with Node-RED and ThingsBoard that identifies modeling and domain adaptation differences without claiming unmeasured cross-platform execution superiority or human productivity gains.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 presents the materials and methods, including the requirements, architecture, rule graph modeling, API contracts, and benchmark protocols. Section 4 reports local network and software-in-the-loop results. Sections 5 and 6 discuss the implications, limitations, and conclusions.

2. Related Work

2.1. Low-Code and Visual Programming Platforms

Low-code and no-code technologies are widely used in enterprise workflow development and dashboard systems. Their main advantage is reducing coding effort through reusable components and configuration-first logic definition [7–10]. Existing studies and platforms generally report gains in development speed, but industrial control scenarios still require strict runtime behavior, deterministic communication, and model consistency, which are not always addressed in generic low-code stacks.

2.2. Microservice Architecture in Industrial and Energy Systems

The microservice architecture has been adopted in industrial software to improve modularity, independent deployment, and maintainability [13–15]. In energy applications, microservice decomposition is useful for separating telemetry ingestion, alarm processing, control dispatch, and visualization. The tradeoff is additional inter-service communication overhead and higher operational complexity, especially under high-frequency data streams [12,16].

2.3. Renewable Energy Monitoring and Control Systems

Monitoring and control systems for PV, storage, and hybrid station operation often involve protocol adapters, data historians, and scheduling logic [1,2,17]. Many implementations remain project-specific, and logic reuse across stations is limited [5]. This motivates a standardized logic description method that can be configured and deployed with reduced dependence on source code-level modifications.

2.4. Graph–Model Integration and Standardized Data Modeling

Graph representations are suitable for station topology, dependency, and flow relationships. Standardized device models, meanwhile, provide reusable abstractions for telemetry points, commands, and alarms, typically aligned with interoperable standards such as IEC 61850 and CIM-related profiles [18,19]. The research gap lies in practical mechanisms that jointly bind the graph topology and model semantics to runtime executable rules in a low-code environment.

2.5. Research Gap and Comparison with Mainstream Flow Tools

General-purpose flow tools and industrial platforms provide important precedents. Node-RED models applications as message-passing flows of configurable nodes and supports reusable configuration nodes and subflows [20]; ThingsBoard organizes device events into rule chains composed of messages, rule nodes, and directed relations, with built-in and scripted processing functions [21]. Ignition provides tag-level event scripts and gateway-scoped execution patterns [22]; OpenEMS focuses on energy management applications [23]; and IEC 61499/Eclipse 4diac provides a function block route for distributed automation [24,25].

The distinction claimed here is not generic flow-based programming. It is the explicit composition of (i) a station topology graph, (ii) typed energy device points and admissible actions, (iii) unit/action validation, and (iv) priority-ordered safety interlocks in one portable rule package. Table 1 compares these modeling boundaries. Node-RED and ThingsBoard can reproduce much of the behavior through custom nodes, scripts, entity relations, or deployment-specific extensions; therefore, this paper does not claim that the proposed abstraction is universally more expressive or faster. A fair execution efficiency comparison would require equivalent deployed flows, aligned persistence and messaging options, identical payloads, and controlled hardware. Such a cross-platform benchmark was not performed and is listed as future work.

Table 1. Scope-based comparison of Node-RED, ThingsBoard rule chains, and the proposed rule graph reference method. “Custom” means that the capability can be constructed through scripts, extensions, or project-specific conventions rather than being established here as a native station-domain contract.

Dimension	Node-RED	ThingsBoard Rule Engine	Proposed Rule-Graph Method
Primary graph object	Message flow of connected nodes	Message/rule node graph with typed relations	Typed station rule graph bound to topology and semantic points
Reusable packaging	Flows, configuration nodes, and subflows	Exportable rule chains and rule nodes	Versioned package with binding and validation metadata
Device semantics	Message properties and contributed/custom nodes	Devices, assets, telemetry, attributes, and relations	Classes, typed points, units, actions, and station entities
Station-topology binding	Custom project convention or extension	Entity relations can provide context; power topology semantics remain project-specific	Explicit topology-to-point/action binding function
Unit and action validation	Custom node or function logic	Custom filters/scripts or data processing rules	Required validation gate in the reference method; partially implemented in the current backend
Safety interlock order	Flow wiring and custom logic	Rule chain routing and custom logic	Explicit safety-before-dispatch priority relation
Execution evidence here	Not deployed or benchmarked	Not deployed or benchmarked	Minimal backend and cooperative evaluator benchmarked
Domain adaptation	Broad IoT/automation ecosystem	Broad IoT device/telemetry platform	Renewable/storage station control logic focus

3. Materials and Methods

3.1. Problem Definition and Design Requirements

3.1.1. Functional Requirements

The reference workflow should support visual rule authoring concepts, standardized device model registration, multi-protocol access abstractions, rule consistency checks, and release-oriented configuration packaging. It should also support role-based access design and operation trace requirements for engineering traceability [26,27].

3.1.2. Non-Functional Requirements

The framework design should preserve modularity for future scaling, clear service boundaries for maintainability, and practical usability for mixed teams with different programming backgrounds.

3.1.3. Performance Targets

In this study, indicators are limited to local network latency, process resource use, scalability, in-process application path timing consistency, and logical invariant checks in a scripted software-in-the-loop trace. They are not field validation or human productivity indicators.

3.1.4. Security and Maintainability Requirements

Authentication, authorization, operation audit, and centralized configuration governance are required at the reference design level [26,27]. Versioned logic definitions and service-level observability are included as design constraints for future implementation.

3.1.5. Design Objectives

The design objective is to provide a transparent reference architecture and reproducible evaluation workflow that can be audited and extended. It is not to claim production readiness or completed industrial deployment.

Table 2 provides a worked-example comparison of the engineering artifacts used in manual code-first and rule graph representations for cooperative BESS dispatch and safety interlocks.

Table 2. Worked example comparison for implementing cooperative BESS dispatch and safety interlocks. The comparison describes engineering artifacts and makes no development time or productivity inference.

Engineering Task	Manual Code-First Representation	Rule-Graph Reference Representation
Point access	Field names, tags, conversions, and device lookups are encoded in application code	Input nodes bind to typed semantic points through a binding map
Cooperative allocation	Allocation equations and saturation checks are implemented as functions	Calculation and allocation nodes encode the same equations in the rule graph
Safety ordering	Execution order is maintained by branches, guards, and calling structure	Priority metadata require quality and protection gates before dispatch nodes
Validation	Type, unit, target, cycle, and precedence checks require tests or custom assertions	The reference validation gate defines these checks as package constraints
Retargeting	Device identifiers and interfaces are revised in source/configuration code	Bindings can be replaced while retaining the rule topology when semantics match
Evidence in this study	No engineer study or project record was collected	The cooperative rule is executable in the minimal backend, but visual authoring productivity is not evaluated

Table 3 maps the functional requirements defined above to the corresponding modules of the reference architecture; their implementation status is distinguished separately in Table 4.

Table 3. Functional requirements and corresponding reference modules.

Functional Requirement	Reference Module(s)
Visual logic orchestration	Visual programming designer, rule engine service
Standardized device model management	Device model service, graph-model binding engine
Real-time telemetry subscription	WebSocket session service, message bus
Rule deployment and rollback	Workflow orchestration service, configuration center
Protocol adaptation	Protocol adapter services, API gateway
Monitoring and audit	Visualization service, logging and audit service
User and permission management	Authentication center, API gateway

3.2. System Architecture

We propose a reference architecture with layered responsibilities: presentation, gateway/security, application services, integration runtime, and data/observability. The purpose is to structure station logic development and evaluation workflows, not to report a completed software product. The layered decomposition is illustrated in Figure 2. The figure shows engineer and operator interfaces connected through northbound REST/WebSocket access, gateway and security functions, configuration/rule/dispatch services, protocol adapters, message bus-like coordination, telemetry aggregation, and storage/observability concepts. These elements define the target reference architecture; Table 4 identifies the smaller subset that is implemented and evaluated.

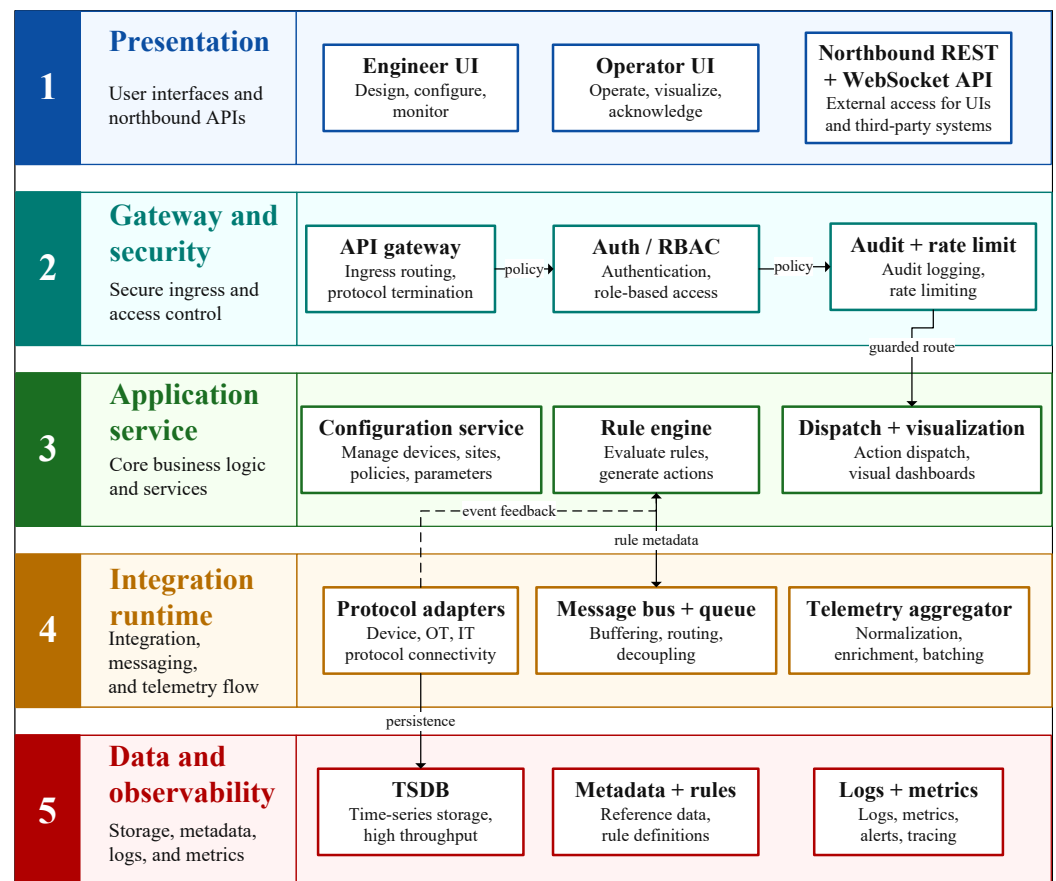


Figure 2. Proposed low-code reference architecture with presentation, gateway/security, application service, integration runtime, and data/observability layers. The figure is a target design; implementation status is given in Table 4.

Table 4. Implementation and evidence status of the components shown in the target architecture.

Component or Function	Current Status	Evidence and Boundary
REST telemetry and status API	Implemented and evaluated	Local Uvicorn/FastAPI network benchmarks and archived request samples
WebSocket telemetry API	Implemented and evaluated	Persistent-session local latency benchmark and archived samples
Single-rule evaluator	Implemented and evaluated	Deterministic PV-load-SOC-frequency threshold rule

Table 4. *Cont.*

Component or Function	Current Status	Evidence and Boundary
Cooperative multi-device evaluator	Implemented and evaluated in this work	Scripted software-in-the-loop trace and in-process ASGI timing samples
In-process queue worker and basic file logging	Implemented and evaluated	Single-process queue, counters, and runtime log; not an external message bus or observability stack
Visual editor and dashboard	Reference design only	No runnable frontend or operator usability study
API gateway and authentication/RBAC	Reference design only	No independent gateway, identity provider, or security evaluation
Configuration center and package registry	Reference design only	Version/package semantics are specified; no deployed configuration service
External message bus, cache, and time-series database	Reference design only	No RabbitMQ, Redis, InfluxDB, or equivalent middleware is used in the reported tests
Industrial protocol adapters	Future implementation and validation	IEC 61850/Modbus/OPC UA adapters and conformance tests are not included
RTU/PLC/industrial PC and HiL deployment	Future validation	No industrial controller, real-time operating system, hardware-in-the-loop, or station pilot evidence

3.2.1. Presentation Layer

This layer defines visual interaction concepts for rule configuration, model binding inspection, and monitoring-oriented dashboards in future implementations.

3.2.2. Gateway and Access Layer

API gateway and authentication concepts provide identity, routing, and access control boundaries. REST-style endpoints are used for configuration operations, while WebSocket-style channels are included for event-driven update patterns [28–30].

3.2.3. Application Service Layer

This layer defines reference services for configuration management, device model management, rule processing, orchestration, visualization, and auditability.

3.2.4. Application Logic and Data Layers

Graph-model mapping logic links topology entities to standardized semantic attributes. Protocol adapter abstractions normalize heterogeneous communication semantics. Message bus interaction is modeled as asynchronous coordination among rule, data, and event flows [16,31].

3.2.5. Reference Deployment Considerations

Service decomposition is designed to support independent evolution, version control, and incremental replacement in future implementation stages. This subsection states design intent rather than providing current deployment evidence.

Table 5 summarizes the intended responsibilities of the microservice components in the reference design.

Table 5. Microservice components and their responsibilities.

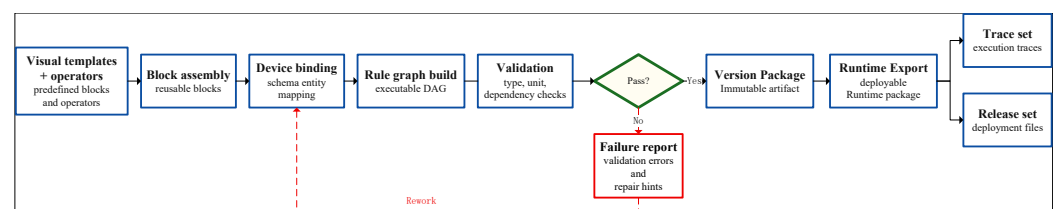
Component	Responsibility
Authentication center	User identity verification, token management, role-based access control
Service gateway	Unified external entry point, service routing, request throttling
Configuration center	Centralized configuration versioning and dynamic update
Message bus	Asynchronous event dissemination and decoupled service interaction
Rule engine	Execution of generated control logic and trigger processing
Logging and audit service	Operation trace, alarm/event audit, diagnostics support
Data access service	Unified access to time-series and relational data
Visualization service	Dashboard rendering, trend plots, alarm panels

3.3. Rule Graph Model and Low-Code Configuration Method

The reference workflow is designed to follow the sequence shown in Figure 3.

1. **Visual templates and operators:** Predefined blocks and operators provide reusable domain-level logic elements.
2. **Block assembly:** Engineers combine reusable blocks into control logic flows.
3. **Device binding:** Blocks are mapped to schema entities and standardized semantic points rather than vendor-specific register definitions.
4. **Rule graph build:** Visual flows are translated into executable directed rule graphs [7,10].
5. **Validation gate:** Type, unit, dependency, and boundary checks determine whether the rule graph can proceed.
6. **Failure report and rework:** Failed validation generates repair hints and returns the configuration to the rework path.
7. **Version package and runtime export:** Passed rules are packaged as immutable artifacts and exported as runtime packages.
8. **Trace and release outputs:** Execution traces and deployment files provide auditability for later review.

This workflow defines a reference method for logic configuration and artifact generation. It does not by itself establish production deployment readiness.

**Figure 3.** Visual programming workflow for station control logic configuration, validation, rework, version packaging, runtime export, and trace/release outputs.

The graph-model/device-semantic binding mechanism used in the device binding step is illustrated in Figure 4. The mechanism starts from two design inputs: a topology graph containing components, ports, and edges and a standardized semantic model containing classes, units, and constraints. These inputs are merged into runtime binding objects, converted into a point-tag-API mapping matrix, and then used to form deterministic rule mapping and execution contracts. A quality gate checks whether the binding is release-ready; failed bindings produce validation reports and require mapping revision.

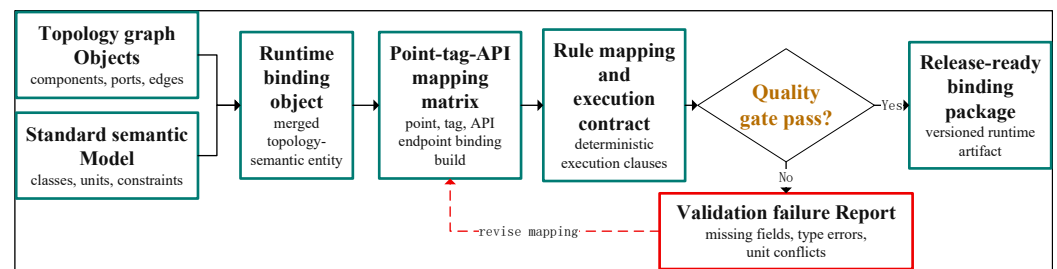


Figure 4. Graph-model integration and device binding from topology/semantic inputs to point-tag-API mapping, execution contracts, quality gate validation, and release-ready binding packages.

3.3.1. Rule Graph and Binding Definition

The low-code method is represented by a station graph, a semantic point model, and a rule graph. Let the station topology be $G_s = (V_s, E_s)$, where V_s contains physical or logical station entities such as PV inverters, BESS devices, buses, loads, and dispatch services, and E_s contains power, telemetry, or control relationships. Each entity $v \in V_s$ is associated with a semantic descriptor

$$m(v) = \{c_v, U_v, P_v, A_v\},$$

where c_v is the device class, U_v is the admissible unit set, P_v is the set of telemetry points, and A_v is the set of executable actions. A low-code rule is represented as a directed rule graph $R = (N_R, E_R)$, where nodes represent input points, predicates, logical operators, actions, delays, and safety interlocks. A binding function $b : N_R \rightarrow P_v \cup A_v$ maps rule nodes to semantic points or actions. A rule package is considered syntactically executable only if every bound input has a type, unit, and source entity; every action has a target device and admissible command type; and the rule graph is acyclic except for explicitly declared feedback monitoring loops.

The validation gate checks five conditions: (i) all referenced points exist in the semantic model; (ii) units are compatible with threshold values; (iii) action targets support the requested command; (iv) the rule graph contains no undeclared circular dependency; and (v) safety interlock nodes are evaluated before dispatch actions. These conditions are design-level validation rules. The current backend implements a minimal representative rule check rather than a complete industrial rule engine compiler.

3.3.2. Representative Rule Schema

The following rule illustrates how a visual block chain can be serialized into a rule graph-oriented configuration. It corresponds to a conservative islanded operation logic: if the BESS state of charge is low, the load exceeds the PV output by a configured deficit margin, and the frequency falls below a threshold, a flexible-load shedding action is requested.

```

trigger: telemetry_update
conditions:
  - point: BESS.SOC
    op: "<"; value: 25.0; unit: "%"
  - point: LOAD.P_minus_PV.P
    op: ">"; value: 80.0; unit: "kW"
  - point: ACBUS.Frequency
    op: "<"; value: 49.95; unit: "Hz"
logic: AND
action:
  type: shed_load
  target: FlexibleLoadB
  
```

priority: low

In the minimal backend, the same logic is implemented as a deterministic rule function over the request fields `pv_kw`, `load_kw`, `soc_pct`, and `freq_hz`. This example is included to make the method executable and auditable, not to indicate that a complete visual editor or industrial dispatch system has been implemented.

3.3.3. Priority-Ordered Cooperative Rule and Interlock Semantics

To move beyond the single load shedding example, each rule node $n \in N_R$ is assigned a type $\tau(n) \in \{\text{quality, protection, predicate, calculation, latch, action}\}$ and a priority $\rho(n)$. Lower priority numbers execute first. Every path to an action node must pass through all applicable quality and protection nodes, and the topological execution order π must satisfy

$$\rho(n_i) < \rho(n_j) \Rightarrow \pi(n_i) < \pi(n_j)$$

for safety-relevant predecessor pairs. In the implemented cooperative rule, communication validity, timestamp freshness, and breaker state have priority 0; BESS low-SOC protection has priority 1; frequency/energy predicates and latching have priority 2; and dispatch, load shed, and backup start actions have priority 3.

For M available BESS units, the active power deficit and unit headroom are

$$D_t = \max(P_{L,t} - P_{PV,t}, 0), \quad h_{i,t} = \max(\text{SOC}_{i,t} - \text{SOC}_{\min}, 0) P_{i,t}^{\max}.$$

When the quality/topology gate is valid, the requested dispatch is allocated as

$$u_{i,t} = \min\left(P_{i,t}^{\max}, D_t \frac{h_{i,t}}{\sum_{j=1}^M h_{j,t}}\right),$$

with $u_{i,t} = 0$ when $\text{SOC}_{i,t} \leq \text{SOC}_{\min}$. The residual deficit is $D_t^{\text{res}} = \max(D_t - \sum_i u_{i,t}, 0)$. A load shed latch is asserted when the frequency is below 49.80 Hz, either BESS SOC value is below 20%, or $D_t^{\text{res}} > 20$ kW. Emergency classification uses 49.50 Hz. The latch is released only after five consecutive valid samples with a frequency of at least 50.02 Hz, both SOC values of at least 25%, and a residual deficit no greater than 5 kW. A backup start request is issued when the residual exceeds 50 kW or no BESS discharge is available. Communication failure, timestamp staleness above 500 ms, or an open breaker activates a hard safe-hold gate that forces both BESS setpoints and the backup request to zero.

This evaluator is exposed by `POST /api/v1/rules/cooperative/evaluate`; a separate reset endpoint clears the latch before each reproducibility run. The implementation demonstrates one concrete composition of cooperative allocation, protection, supervisory logic, and recovery. It does not prove completeness for all station control languages, protection schemes, or vendor-specific interlocks.

3.4. Microservice Component Library

The reference framework defines a reusable component library for future station projects:

- Identity and access components for secure multi-user control boundaries;
- Configuration components for reusable station templates and environment-specific parameters;
- Messaging and rule-processing components for event-driven logic interaction patterns;
- Logging and audit components for traceability and governance requirements;
- Data access and visualization components for telemetry-oriented information flow.

Table 6. Minimal reference backend API contract used for local scripted benchmarks.

Interface	Input/Operation	Output/Role
GET /health	No payload	Health status, queue depth, processed message counter
GET /api/v1/status	No payload	Processed counter, control action counter, queue depth, last processing time
POST /api/v1/telemetry	Telemetry payload with PV, load, SOC, frequency, and point count fields	REST acceptance response and server-side processing time record
POST /api/v1/rules/evaluate	Trigger name, threshold, and current value	Minimal rule evaluation response for contract testing
POST /api/v1/rules/cooperative/reset	No payload	Clear load shed latch and recovery counter before a run
POST /api/v1/rules/cooperative/evaluate	Two-BESS telemetry, availability, breaker, communication, staleness, and backup-ready fields	Priority-ordered cooperative dispatch, interlock, latch, recovery, and backup response
WebSocket /ws/telemetry	Same telemetry payload transmitted through a persistent WebSocket session	WebSocket acceptance response and server-side processing time record

3.5.4. Reference Artifacts and Benchmark Configuration

Table 7 summarizes available implementation artifacts and benchmark configuration boundaries.

Table 7. Reference implementation artifacts and benchmark configuration boundaries.

Item	Available Evidence/Configuration
Scenario scope	Islanded PV–storage–load reference case with single-rule and cooperative two-BESS scripted configurations
Measurement point levels	100, 500, 1000, 2000 points
Reference backend service	Minimal local reference backend with REST/WebSocket telemetry interfaces
Benchmark workflow	Scripted request generation, runtime sampling, CSV export, plotting, disturbed software-in-the-loop trace, and application path timing sampling
Containerization boundary	Container orchestration template prepared for reference deployment but not used as field evidence
Runtime evidence	Local runtime logs and benchmark execution records
Generated benchmark outputs	Latency, resource usage, scalability, cooperative rule trace, logical invariant, and timing summaries from scripted local runs
Hardware boundary	General-purpose Windows workstation; no RTU, PLC, industrial PC, real-time OS, or HiL target
Excluded evidence	No developer study, historical project time records, synthetic productivity percentage, or synthetic component ablation proxy is used as a result
Evaluation boundary	Local workstation network measurements and scripted software-in-the-loop inputs; not field deployment evidence

3.6. Benchmark Design and Scripted Software-in-the-Loop Protocol

Local benchmark results were collected from the minimal FastAPI reference backend on a Windows 10 workstation with a 12th-Gen Intel Core i5-12490F CPU (6 physical cores, 12 logical processors) and approximately 16 GB RAM, using Python 3.10.11. This workstation is not representative of an RTU, PLC, industrial PC, or real-time controller. The backend queue size was 40,000 messages. The benchmark script used four measurement point levels: 100, 500, 1000, and 2000 points.

Latency measurements used 200 requests per interface and measurement point level. REST requests were sent through `POST /api/v1/telemetry`; WebSocket measurements used one persistent connection per measurement point level. Resource measurements used paced REST input rates of 2, 5, 10, and 20 messages/s for 100, 500, 1000, and 2000 points, respectively, with a 24 s sampling window and a 0.25 s CPU/memory sampling interval. Scalability measurements used a separate burst-style workload of 200 sequential REST requests per measurement point level without the paced waiting interval used in the resource scenario. Therefore, the CPU results from the paced resource and burst scalability scenarios are not directly comparable because they use different workload definitions.

The cooperative rule trace used fixed random seed 20260806 and 600 simulated steps at a nominal 100 ms interval. The input trajectory contained five phases: normal operation; PV reduction and load increase; emergency under-frequency; low-SOC operation of one BESS; communication validity, timestamp staleness, and breaker state faults; and recovery. Gaussian perturbations were applied to PV, load, frequency, SOC, and timestamp age. The injected quality events comprised 13 communication fault samples, 12 stale input samples above 500 ms, and 7 open breaker samples. Logical invariant checks counted any nonzero BESS or backup dispatch during a hard safety block and any nonzero dispatch from a BESS at or below the 15% protection threshold.

Timing consistency was evaluated through the FastAPI `TestClient` in-process ASGI request path to isolate application dispatch and rule execution from TCP transport. Twenty warm-up requests preceded each run. For each point level, 500 sequential requests were measured. Reported statistics are the median, mean, population standard deviation (used here as a jitter descriptor), P95, P99, maximum latency, and number of samples above an illustrative 50 ms reference deadline. This deadline is not asserted as a station protection requirement. The original REST/WebSocket benchmark remains the transport path measurement; the new timing test must not be compared directly with the network transport latency results.

The earlier assumed base times, complexity factors, 45.63% development time proxy, and component ablation proxies have been removed from the evidence chain because no developer study or historical engineering project records are available to justify them. Table 2 provides a worked-example artifact comparison only; human productivity remains an open validation question.

4. Results

The evaluation contains two bounded local evidence paths: (i) network path measurements from the runnable Uvicorn/FastAPI backend and (ii) a scripted software-in-the-loop cooperative rule trace plus in-process ASGI timing samples. Neither path uses field station data, industrial controllers, or human participants.

Table 8 summarizes the four evaluation scenarios and their corresponding evidence types and metrics.

Table 8. The evaluation scenarios and metrics from the implemented local reference backend.

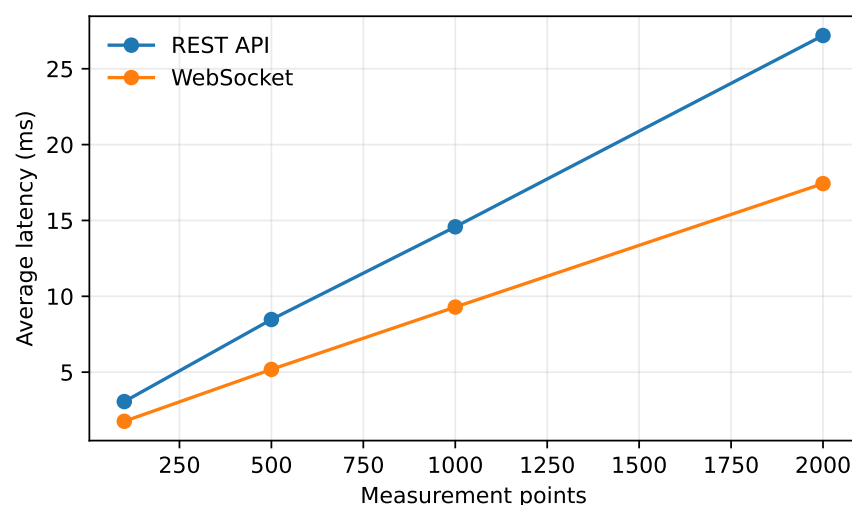
Scenario	Evidence Type	Metrics
Scenario 1: REST vs. WebSocket latency	Local network path benchmark	Average, P95, and P99 request latency (ms)
Scenario 2: resource usage under paced input	Local network path benchmark	Average/P95 CPU and memory, input rate, accepted ratio
Scenario 3: scalability from 100 to 2000 points	Local burst network path benchmark	Response time, throughput, CPU, memory, accepted ratio
Scenario 4: cooperative control under disturbances	Local scripted software-in-the-loop trace and in-process ASGI timing	Interlock/action counts, logical violations, median, mean, jitter, P95, P99, maximum latency, and deadline misses

4.1. Scenario 1: Scripted Benchmark Latency on Minimal Reference Backend

On this local reference backend, scripted benchmark outputs indicate that the WebSocket channel has lower average and tail latency than REST for the same scripted workload, as reported in Table 9 and summarized visually in Figure 6. This observation is limited to the current implementation and workstation configuration and should not be interpreted as field station performance.

Table 9. Scripted benchmark latency outputs from the local reference backend.

Measurement Points	Interface	Avg. Latency (ms)	P95 (ms)	P99 (ms)	Sample Size
100	REST API	3.06	4.75	5.22	200
100	WebSocket	1.76	2.82	3.22	200
500	REST API	8.47	11.11	13.19	200
500	WebSocket	5.18	7.07	8.12	200
1000	REST API	14.58	18.99	21.36	200
1000	WebSocket	9.29	12.84	14.67	200
2000	REST API	27.19	33.11	36.41	200
2000	WebSocket	17.43	22.64	24.76	200

**Figure 6.** Scripted benchmark latency comparison on the local reference backend.

4.2. Scenario 2: Scripted Benchmark Resource Usage

The paced resource outputs in Table 10 show a monotonic increase in CPU usage with the message volume, while memory usage remains stable at around 42–43 MB in the current minimal implementation, as shown in Figure 7. The near-constant memory values reflect the small single-process backend and bounded in-memory queue during this short local run; they should not be generalized to a production historian, cache, or multi-service deployment. These outputs reflect the provided reference code only and do not represent optimized production deployment behavior.

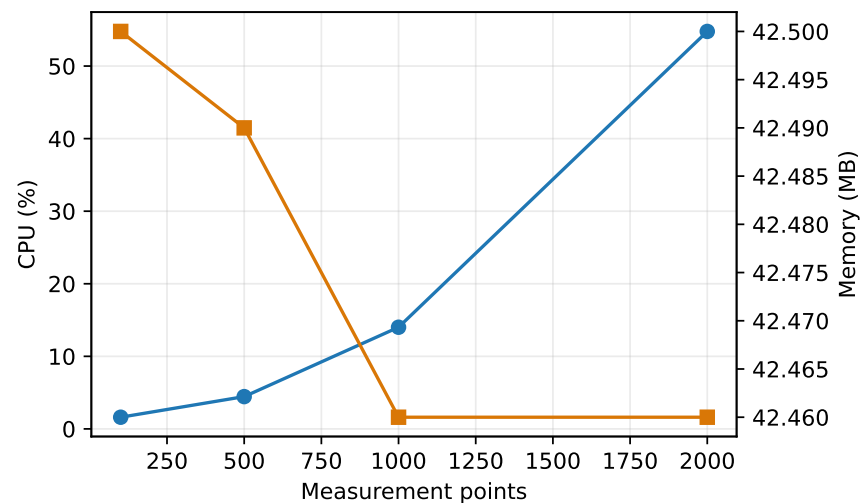


Figure 7. Scripted benchmark CPU and memory usage on the local reference backend. The blue line with circular markers denotes average CPU usage on the left axis, and the orange line with square markers denotes average memory usage on the right axis.

Table 10. Paced scripted resource outputs on the local reference backend.

Measurement Points	Architecture	Avg. CPU (Paced, %)	P95 CPU (Paced, %)	Avg. Memory (MB)	P95 Memory (MB)	Input Rate (msg/s)	Accepted Ratio (%)
100	Minimal Runnable Reference Backend	1.61	6.20	42.50	42.50	2	100.00
500	Minimal Runnable Reference Backend	4.45	11.80	42.49	42.50	5	100.00
1000	Minimal Runnable Reference Backend	14.01	20.44	42.46	42.46	10	100.00
2000	Minimal Runnable Reference Backend	54.77	64.90	42.46	42.46	20	100.00

4.3. Scenario 3: Scripted Benchmark Scalability

The scripted benchmark response time increases with the point count and scripted message volume, and the achieved throughput follows the implemented local benchmark script configuration, as reported in Table 11 and shown in Figure 8. The CPU values in Table 11 are higher than those in Table 10 because the scalability scenario uses an unpaced burst-style request loop, whereas the resource scenario uses paced input rates and a fixed sampling window. The 100% accepted request ratio means that the local backend queue was not saturated during these scripted runs; it should not be interpreted as proof of control correctness, fault tolerance, or field reliability.

Table 11. Burst-style scripted scalability outputs from the local reference backend.

Measurement Points	Response Time (ms)	Throughput (msg/s)	Avg. CPU (Burst, %)	Avg. Memory (MB)	Accepted Request Ratio (%)
100	3.25	303.97	68.40	42.53	100.00
500	8.38	118.84	84.67	42.53	100.00
1000	14.99	66.50	92.99	42.53	100.00
2000	27.45	36.37	93.75	42.53	100.00

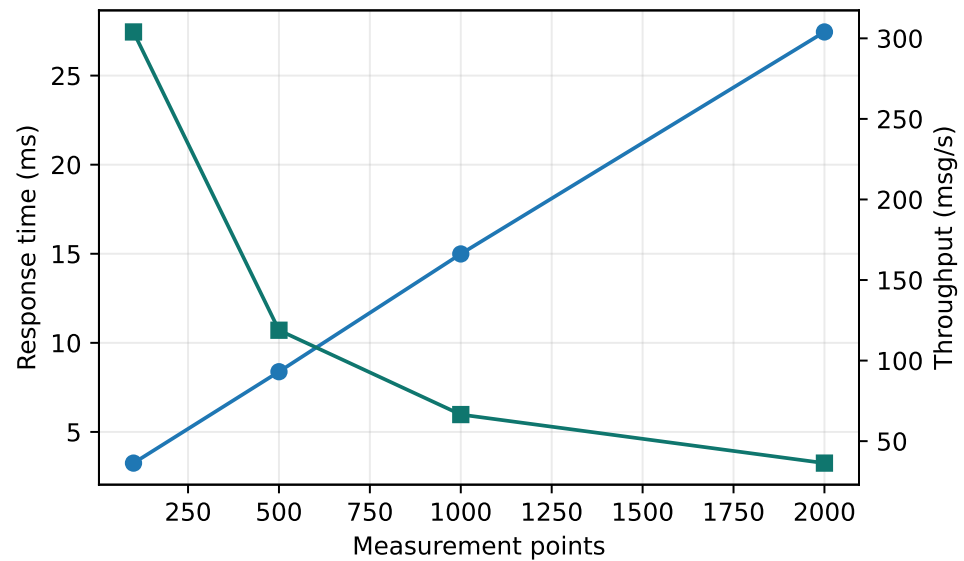


Figure 8. Scripted benchmark scalability trends on the local reference backend. The blue line with circular markers denotes response time on the left axis, and the teal line with square markers denotes throughput on the right axis.

4.4. Scenario 4: Cooperative Rule Software-in-the-Loop Trace and Timing Consistency

Table 12 summarizes the 600-step trace. Communication, staleness, and breaker faults overlap in some steps; therefore, their individual counts do not sum to the 25 hard safety block samples. During every safety block, both BESS setpoints and the backup request remained zero. The low-SOC protection also prevented dispatch from the protected BESS in every applicable sample. These zero-violation counts verify the implemented invariants for this scripted trajectory, but they are not reliability probabilities and do not cover all possible state combinations.

Table 12. Local scripted software-in-the-loop trace events and rule outcomes.

Trace Metric	Value
Trace samples/nominal simulated step	600/100 ms
Injected communication fault/stale input/ open breaker samples	13/12/7
Hard safety block samples	25
Load shed latch/backup start samples	240/160
Rule mode transitions	22
Dispatch actions during hard safety block	0
Dispatch actions from a BESS at or below 15% SOC	0
Mean/P99/maximum request latency	5.5832/6.3091/88.6616 ms

Figure 9 shows the relations among the scripted frequency trajectory, cooperative BESS setpoints, latched actions, safety blocks, and application path request latency. The 20–28 s interval triggers emergency shedding; the subsequent low-SOC interval suppresses BESS 1 and reallocates available discharge to BESS 2; communication quality and breaker events in the 36–43 s interval force a safe hold; and the latch is released after the specified five-sample recovery condition. This is a software-in-the-loop API trace driven by generated inputs, not a hardware-in-the-loop experiment or a dynamic power system plant model.

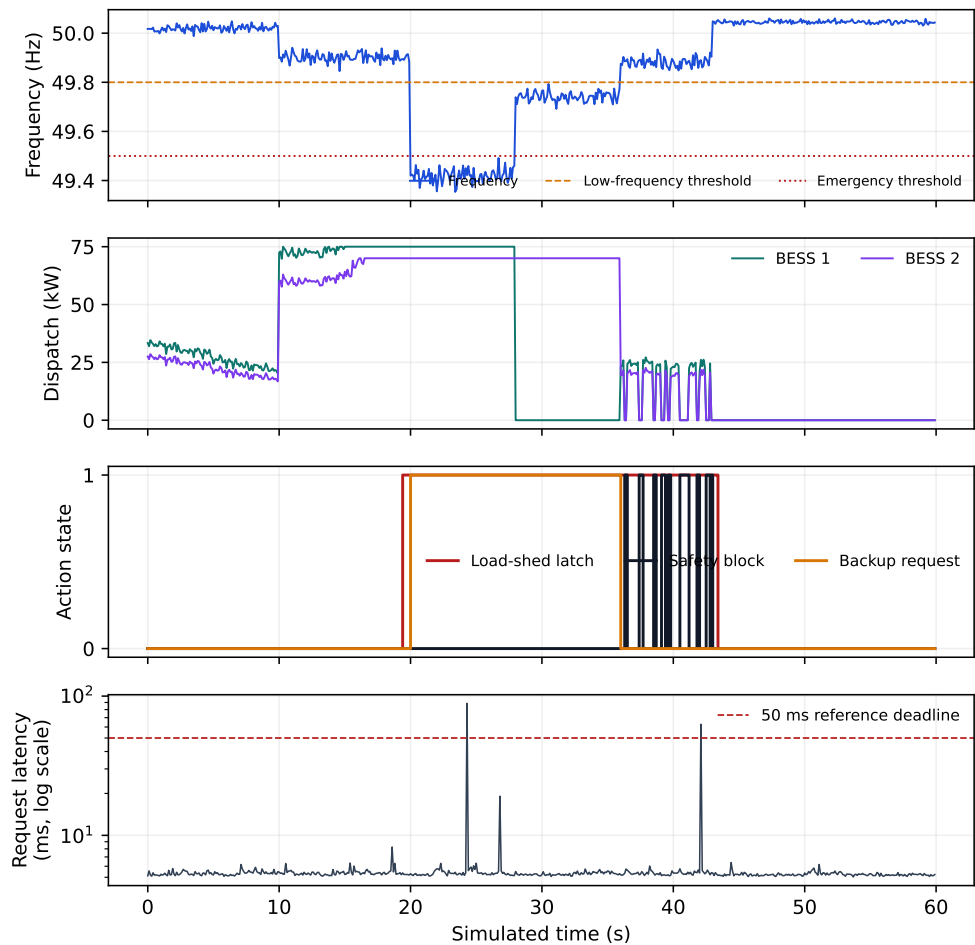


Figure 9. Local scripted software-in-the-loop trace for the cooperative two-BESS rule. The panels show frequency thresholds, dispatch allocation, latched/safety actions, and in-process ASGI request latency. Overlapping traces in the action-state panel indicate simultaneous active binary states and are distinguished by color in the legend. The red latency line is an illustrative 50 ms reference deadline, not a validated station protection requirement.

Table 13 reports the timing consistency after 20 warm-up requests per point level. The median and P99 latency increase with the deterministic workload coefficient. The 100-point group contains one 75.3065 ms scheduler outlier, which raises its mean and standard deviation and produces the only 50 ms reference deadline miss. The outlier, together with the lack of a real-time operating system, prevents a determinism claim even though the remaining P99 values are below 6 ms.

Table 13. In-process FastAPI ASGI timing consistency for the cooperative rule evaluator (500 measured requests per point level after 20 warm-up requests).

Points	Median (ms)	Mean (ms)	Std. Jitter (ms)	P95 (ms)	P99 (ms)	Maximum (ms)	Reference Deadline (ms)	Misses
100	0.9025	1.1057	3.4020	1.0694	1.1962	75.3065	50	1
500	1.8212	1.8876	0.8608	2.0408	2.1765	20.9985	50	0
1000	2.9301	3.0131	1.0403	3.2232	3.6298	26.0098	50	0
2000	5.1701	5.2545	1.1501	5.3453	5.5287	26.5559	50	0

5. Discussion

The present evidence chain contains no synthetic productivity or proxy ablation claim. It combines transport path measurements from the minimal local backend with an executable cooperative rule trace and application path timing samples. The contribution is therefore a reference method plus bounded implementation evidence, not a completed low-code platform or industrial control product.

Relative to Node-RED and ThingsBoard, the proposed method places station topology, typed points, units, admissible actions, and safety priority metadata in the rule package contract. Node-RED and ThingsBoard offer broader ecosystems and can implement similar behavior through nodes, scripts, relations, and extensions. The present study neither deploys these platforms nor demonstrates lower execution times than them. Its narrower novelty is the explicit energy station binding and validation contract. A controlled cross-platform benchmark remains necessary before any execution efficiency ranking can be derived.

The cooperative rule trace strengthens the expressiveness evidence beyond a single threshold rule by combining two-device allocation, communication and topology gates, per-device SOC protection, frequency thresholds, a persistent load shed latch, recovery confirmation, and backup request. The zero invariant violations across 600 scripted samples shows that the implemented evaluator followed the specified priority constraints for the trace. The results do not establish language completeness, formal verification, or reliability under untested states.

The timing results also change the interpretation of “real-time”. The P99 application path latency remained below 6 ms in the four sampled point level groups, but a 75.3065 ms outlier produced one miss of the illustrative 50 ms deadline. The outlier is consistent with scheduling variability on a non-real-time desktop operating system. FastAPI, Python, TCP/WebSocket communication, and microservice messaging do not by themselves provide bounded worst-case execution. Safety-critical interlocks would require placement on a validated PLC/RTU/real-time controller or an independently certified protection layer; the proposed service layer should be treated as supervisory until such partitioning and worst-case timing analysis are completed.

Hardware validity remains limited. A 12th-Gen desktop CPU can overstate the compute capacity relative to some RTUs and underrepresent the I/O, protocol, environmental, and deterministic scheduling characteristics of industrial controllers. Conversely, the single-process backend omits database, broker, security, and multi-service overhead. The reported resource and latency values therefore cannot be transferred quantitatively to an industrial PC, PLC, or station network.

Finally, “low-code” is currently a design characteristic and not an empirically measured productivity outcome. The worked example in Table 2 explains how bindings and validation metadata could separate reusable rule topology from device-specific mappings, but no engineer participants or historical project records were analyzed. A future study should define representative tasks, recruit station control engineers with balanced experience, counterbalance tool order, record the completion time and errors, and report uncertainty and learning effects.

Table 14 consolidates the present evidence limitations and the corresponding validation work required in future studies.

Table 14. Present limitations and evidence required for future validation.

ID	Current Limitation	Planned Improvement
L1	Tests use a Windows desktop, not an RTU, PLC, industrial PC, or real-time OS	Repeat latency, jitter, CPU, and memory tests on representative industrial hardware
L2	No engineer usability study or historical project time evidence	Conduct a preregistered, counterbalanced developer study with task, error, and time measures
L3	Node-RED and ThingsBoard were compared by scope, not by controlled execution benchmark	Implement equivalent flows and align persistence, messaging, payloads, hardware, and load generation
L4	The backend is a minimal single service; most Figure 2 components remain design-level	Deploy independent gateway, identity, configuration, rule, broker, database, and observability services
L5	The 600-step trajectory is generated software-in-the-loop input, not a dynamic plant, HiL rig, or pilot	Connect the rule service to a real-time simulator and later to supervised station hardware
L6	One 50 ms reference-deadline miss occurred and no worst-case timing bound is established	Use real-time targets, isolate safety functions, and perform worst-case and fault injection timing analyses
L7	One cooperative scenario cannot establish complete rule language expressiveness	Add grid-connected, black start, anti-islanding, thermal limit, and multi-level protection case suites
L8	No real station trace, industrial protocol adapter, or field deployment is available	Validate anonymized traces and IEC 61850/Modbus/OPC UA adapters under approved pilot conditions

6. Conclusions

This work proposed a rule graph-based reference architecture and implemented a bounded FastAPI subset for low-code control logic research in renewable and energy storage station scenarios. The article makes the implementation boundary explicit, presents no unsupported synthetic productivity and component ablation claims, offers a direct scope comparison with Node-RED and ThingsBoard, and formalizes a priority-ordered cooperative two-BESS rule.

The new 600-step scripted software-in-the-loop trace exercised cooperative dispatch, communication/topology blocking, low-SOC protection, load shed latching, recovery, and backup requests without invariant violations in the tested trajectory. The application path P99 latency remained below 6 ms, but an isolated 75.3065 ms outlier caused one miss of an illustrative 50 ms deadline. This finding prevents a determinism or industrial real-time claim.

The Supplementary Materials reproducibility package (File S1) provides the executable evaluator, API artifacts, scripts, raw samples, CSV summaries, logs, and plotting code. The results establish the reproducibility of the local reference workflow only. Industrial controller measurements, equivalent Node-RED/ThingsBoard deployments, engineer productivity studies, dynamic plant HiL tests, real protocol adapters, and field pilots remain necessary before engineering effectiveness or operational readiness can be assessed.

Supplementary Materials: The following supporting information can be downloaded at <https://www.mdpi.com/article/10.3390/electronics15163745/s1>: A reproducibility package containing the minimal FastAPI reference backend, REST/WebSocket and cooperative rule benchmark scripts, generated CSV summaries, raw network/timing/trace samples, plotting scripts, runtime logs, a

container template, and README instructions. The package contains no real station operation logs, field measurements, industrial controller outputs, or human participant data.

Author Contributions: Conceptualization, J.L. and M.X.; methodology, J.L., M.X., C.Y. and X.X.; software, J.L. and C.Y.; validation, J.L., C.Y. and Y.G.; formal analysis, J.L. and X.X.; investigation, C.Y. and Y.G.; resources, M.X.; data curation, J.L. and C.Y.; writing—original draft preparation, J.L.; writing—review and editing, M.X., C.Y., X.X. and Y.G.; visualization, J.L. and Y.G.; supervision, M.X.; project administration, M.X. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Science and Technology Project of the State Grid Hubei Electric Power Co., Ltd., grant number B31532255957.

Data Availability Statement: The data and code supporting this study are provided in the Supplementary Materials reproducibility package, including the minimal backend, cooperative rule evaluator, benchmark and plotting scripts, generated CSV summaries, and raw local network, timing, and scripted trace samples. No real station operation logs, field measurements, industrial controller outputs, human participant data, or confidential utility data are included. Additional materials can be requested from the corresponding author if required for verification, subject to institutional approval.

Acknowledgments: The authors would like to thank the State Grid Hubei Electric Power Research Institute, the Hubei Key Laboratory of Regional New Power Systems and Rural Energy System Configuration, and the Hubei Engineering Research Center of the Construction and Operation Control Technology of New Power Systems for their support.

Conflicts of Interest: All authors are affiliated with the State Grid Hubei Electric Power Research Institute and the Hubei provincial laboratory and engineering research center listed in the affiliations. This study received funding from the Science and Technology Project of the State Grid Hubei Electric Power Co., Ltd. and was conducted as part of an internal research project. The funding organization provided financial and institutional support but did not impose restrictions on the analysis, interpretation, or manuscript preparation. The authors declare no other conflicts of interest.

References

1. Fang, X.; Misra, S.; Xue, G.; Yang, D. Smart Grid—The New and Improved Power Grid: A Survey. *IEEE Commun. Surv. Tutor.* **2012**, *14*, 944–980. [CrossRef]
2. Olivares, D.E.; Mehrizi-Sani, A.; Etemadi, A.H.; Cañizares, C.A.; Iravani, R.; Kazerani, M.; Hajimiragha, A.H.; Gomis-Bellmunt, O.; Saeedifard, M.; Palma-Behnke, R.; et al. Trends in Microgrid Control. *IEEE Trans. Smart Grid* **2014**, *5*, 1905–1919. [CrossRef]
3. Guerrero, J.M.; Vasquez, J.C.; Matas, J.; de Vicuña, L.G.; Castilla, M. Hierarchical Control of Droop-Controlled AC and DC Microgrids—A General Approach Toward Standardization. *IEEE Trans. Ind. Electron.* **2011**, *58*, 158–172. [CrossRef]
4. Khan, M.Y.A.; Liu, H.; Wang, J.; Zhang, R.; Zhang, Y. Hierarchical Control Framework for Offshore Hybrid AC/DC Microgrid Integrating Renewable Energy Resources and Hybrid Energy Storage System. *J. Energy Storage* **2026**, *152*, 120749. [CrossRef]
5. Khodaei, A. Provisional Microgrid Planning. *IEEE Trans. Smart Grid* **2017**, *8*, 1096–1104. [CrossRef]
6. Knowles, W.; Prince, D.; Hutchison, D.; Disso, J.-F.P.; Jones, K. A Survey of Cyber Security Management in Industrial Control Systems. *Int. J. Crit. Infrastruct. Prot.* **2015**, *9*, 52–80. [CrossRef]
7. Di Ruscio, D.; Kolovos, D.; de Lara, J.; Pierantonio, A.; Tisi, M.; Wimmer, M. Low-Code Development and Model-Driven Engineering: Two Sides of the Same Coin? *Softw. Syst. Model.* **2022**, *21*, 437–446. [CrossRef]
8. Bucaioni, A.; Cicchetti, A.; Ciccozzi, F. Modelling in Low-Code Development: A Multi-Vocal Systematic Review. *Softw. Syst. Model.* **2022**, *21*, 1959–1981. [CrossRef]
9. Pinho, D.; Aguiar, A.; Amaral, V. What About the Usability in Low-Code Platforms? A Systematic Literature Review. *J. Comput. Lang.* **2023**, *74*, 101185. [CrossRef]
10. Sahay, A.; Indamutsa, A.; Di Ruscio, D.; Pierantonio, A. Supporting the Understanding and Comparison of Low-Code Development Platforms. In Proceedings of the 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Portorož, Slovenia, 26–28 August 2020; pp. 171–178. [CrossRef]
11. Lewis, J.; Fowler, M. Microservices: A Definition of This New Architectural Term. Available online: <https://martinfowler.com/articles/microservices.html> (accessed on 27 May 2026).
12. Newman, S. *Building Microservices*, 2nd ed.; O'Reilly Media: Sebastopol, CA, USA, 2021.

13. Jamshidi, P.; Pahl, C.; Mendonça, N.C.; Lewis, J.; Tilkov, S. Microservices: The Journey So Far and Challenges Ahead. *IEEE Softw.* **2018**, *35*, 24–35. [CrossRef]
14. Di Francesco, P.; Lago, P.; Malavolta, I. Architecting with Microservices: A Systematic Mapping Study. *J. Syst. Softw.* **2019**, *150*, 77–97. [CrossRef]
15. Dragoni, N.; Giallorenzo, S.; Lafuente, A.L.; Mazzara, M.; Montesi, F.; Mustafin, R.; Safina, L. Microservices: Yesterday, Today, and Tomorrow. In *Present and Ulterior Software Engineering*; Mazzara, M., Meyer, B., Eds.; Springer: Cham, Switzerland, 2017; pp. 195–216. [CrossRef]
16. Richardson, C. *Microservices Patterns*; Manning Publications: Shelter Island, NY, USA, 2018.
17. Sridhar, S.; Hahn, A.; Govindarasu, M. Cyber-Physical System Security for the Electric Power Grid. *Proc. IEEE* **2012**, *100*, 210–224. [CrossRef]
18. IEC 61850-7-420:2021; Communication Networks and Systems for Power Utility Automation—Part 7-420: Basic Communication Structure—Distributed Energy Resources and Distribution Automation Logical Nodes. International Electrotechnical Commission (IEC): Geneva, Switzerland, 2021.
19. IEC 61970-301:2020; Energy Management System Application Program Interface (EMS-API): Common Information Model (CIM) Base. International Electrotechnical Commission (IEC): Geneva, Switzerland, 2020.
20. OpenJS Foundation. Node-RED User Guide: Concepts. Available online: <https://nodered.org/docs/user-guide/concepts> (accessed on 6 August 2026).
21. ThingsBoard. Rule Engine 2.0 Overview. Available online: <https://thingsboard.io/docs/user-guide/rule-engine-2-0/overview/> (accessed on 6 August 2026).
22. Inductive Automation. Ignition User Manual: Tag Event Scripts. Available online: <https://www.docs.inductiveautomation.com/docs/8.1/platform/tags/tag-event-scripts> (accessed on 3 June 2026).
23. OpenEMS Association. OpenEMS: Open Source Energy Management System. Available online: <https://openems.io/> (accessed on 3 June 2026).
24. IEC 61499-1:2012; Function Blocks—Part 1: Architecture. International Electrotechnical Commission (IEC): Geneva, Switzerland, 2012.
25. Eclipse Foundation. Eclipse 4diac: Open Source PLC Environment for Industrial and Home Automation. Available online: <https://www.eclipse.org/4diac/> (accessed on 3 June 2026).
26. Stouffer, K.; Pease, M.; Tang, C.Y.; Zimmerman, T.; Pillitteri, V.; Lightman, S.; Hahn, A.; Saravia, S.; Sherule, A.; Thompson, M. *Guide to Operational Technology (OT) Security*; NIST Special Publication 800-82 Rev. 3; National Institute of Standards and Technology: Gaithersburg, MD, USA, 2023. [CrossRef]
27. IEC 62443-3-3:2013; Industrial Communication Networks—Network and System Security—Part 3-3: System Security Requirements and Security Levels. International Electrotechnical Commission (IEC): Geneva, Switzerland, 2013.
28. Fielding, R.T. Architectural Styles and the Design of Network-based Software Architectures. Doctoral Dissertation, University of California, Irvine, CA, USA, 2000.
29. Fielding, R.; Reschke, J. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. In *RFC 7231*; Internet Engineering Task Force (IETF): Fremont, CA, USA, 2014. [CrossRef]
30. Fette, I.; Melnikov, A. The WebSocket Protocol. In *RFC 6455*; Internet Engineering Task Force (IETF): Fremont, CA, USA, 2011. [CrossRef]
31. OASIS. *Advanced Message Queuing Protocol (AMQP) Version 1.0*; OASIS Standard: Burlington, MA, USA, 2012.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.