Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

A Software Suite for Predictive Energy Control: Modular Architecture, Forecasting–Optimization Integration, and Reproducible Deployment Workflow

Alexander A. Karmanov
Financial University under the Government of the Russian Federation
Petr V. Nikitin
pvnikitin@fa.ru
Financial University under the Government of the Russian Federation
Rimma I. Gorokhova
Financial University under the Government of the Russian Federation

Research Article

Keywords: predictive control, software suite, energy management, telemetry processing, nonlinear model predictive control, monitoring, reproducible workflow, industrial AI

Posted Date: April 17th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-9219687/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License. Read Full License

Additional Declarations: No competing interests reported.

A Software Suite for Predictive Energy Control: Modular Architecture, Forecasting–Optimization Integration, and Reproducible Deployment Workflow

Alexander A. Karmanov [1], Petr V. Nikitin [2], Rimma I. Gorokhova [3]

[1] Department of Artificial Intelligence, Faculty of Information Technology and Big Data Analysis, Financial University under the Government of the Russian Federation, Moscow 125993, Russia;
aakarmanov@fa.ru (A.K.)
[2]Department of Artificial Intelligence, Faculty of Information Technology and Big Data Analysis, Financial University under the Government of the Russian Federation, Moscow 125993, Russia; pvnikitin@fa.ru (P.N.)
[3]Department of Information Technology, Faculty of Information Technology and Big Data Analysis, Financial University under the Government of the Russian Federation, Moscow 125993, Russia; rigorokhova@fa.ru (R.I.)
* Correspondence: pvnikitin@fa.ru (P.N.);

Abstract

Predictive control studies often report forecasting or optimization algorithms in isolation, whereas deployment requires a software suite that coordinates telemetry ingestion, data-quality control, forecasting, optimization, monitoring, and model lifecycle management. This paper presents a modular software suite for predictive control of energy consumption systems and analyzes it as a reproducible data-to-decision workflow. The suite is organized around six interacting layers: the physical object, telemetry and preprocessing, the forecasting service, the optimization and supervisory decision service, monitoring, and lifecycle support. An implementation is demonstrated on an indoor ice rink refrigeration system, where a control-oriented state model and a nonlinear predictive optimizer are embedded as service components. The paper formalizes functional, operational, and reproducibility requirements, describes the architecture and orchestration logic, and reports runtime and traceability results for the full execution cycle. In the case study, the mean full cycle time is 0.0311 s and the maximum time is 0.1064 s for a 5 min supervisory interval, leaving a reserve above 2800×. The suite also records model versions, configuration manifests, runtime logs, and monitoring indicators that support retraining and release management.

Keywords: predictive control; software suite; energy management; telemetry processing; nonlinear model predictive control; monitoring; reproducible workflow; industrial AI

1. Introduction

Research on predictive control for energy systems has produced a large body of forecasting models, optimization formulations, and digital-control experiments. The practical barrier to deployment, however, is often not the absence of another algorithmic improvement, but the absence of a software system that connects telemetry, data preparation, forecasting, optimization, supervisory output, and maintenance within one reproducible workflow. This shift from isolated model quality to executable data-to-decision pipelines and lifecycle-managed operation is increasingly visible in industrial artificial intelligence, building-control delivery, digital twins, and reproducible control benchmarking [1,2,3,4,5].

The same trend is evident in the cyber-physical systems literature. Studies on intelligent cyber-physical systems, modular testing frameworks, highly available service-based deployments, microservice practices, event-driven orchestration, and continuous delivery show that advanced analytics become operationally useful only when embedded in modular and monitorable software environments [6,7,8,9,10,11,12]. For predictive energy control, this observation is decisive: a forecasting model and a constrained optimizer may each be technically sound while their combination remains undeployable if data-quality checks, synchronization logic, artifact tracking, and fallback behavior are not formalized at the system level.

An additional challenge arises from lifecycle management. In industrial time-series applications, the predictive core does not remain valid indefinitely. Operating regimes change, disturbance distributions shift, and delayed feedback reveals forecast or control degradation after the fact. Accordingly, MLOps architectures, industrial lifecycle studies, drift-monitoring surveys, scalable telemetry pipelines, and continuous data-delivery models have increasingly treated monitoring, retraining, and artifact lineage as first-class architectural concerns rather than auxiliary engineering tasks [13,14,15,16,17,18,19,20,21]. In the context of predictive control, this means that a deployable system must preserve not only a control cycle, but also a controlled release cycle for the model and its configuration.

Energy-system applications provide a particularly relevant setting for this problem. Digital-twin studies in energy management and platform-oriented building control show that the value of a predictive model depends on how it is embedded into the surrounding software infrastructure, including telemetry alignment, optimization invocation, and service interoperability [5,22,23,24]. Reviews of model predictive control in buildings make the same point from a control perspective: prediction, optimization, constraints, and operational timing have to be treated jointly if the resulting system is to move beyond offline experimentation [25].

The present paper addresses this gap by describing a software suite for predictive control of energy consumption systems and by analyzing it as a modular, reproducible, and computationally feasible software system. The case study is an indoor ice rink refrigeration system. The forecasting and optimization cores are treated as integrated algorithmic services within the suite rather than as the main subject of the article. The central contribution lies in the software-system layer: the formalized requirements, the service architecture, the orchestration logic, the monitoring and lifecycle loop, and the runtime evidence for quasi-real-time use.

The contributions of the paper are fourfold. First, functional, operational, and reproducibility requirements are formulated for a predictive-control software suite that must work on streaming telemetry and produce supervisory actions under strict temporal causality. Second, a modular architecture is developed that separates telemetry ingestion, forecasting, optimization, monitoring, and lifecycle support while preserving a single end-to-end execution chain. Third, the implementation logic is presented as a reproducible data-to-decision workflow with explicit artifact storage, service contracts, and release management. Fourth, the suite is exercised on an operational ice rink refrigeration system, and its runtime characteristics, monitoring outputs, and traceability properties are reported. The resulting contribution is a modular software suite with explicit orchestration, artifact traceability, lifecycle management, and deployable supervisory execution.

The remainder of the paper is organized as follows. Section 2 formalizes the design requirements for the software suite. Section 3 presents the architecture of the suite. Section 4 describes the implementation logic. Section 5 introduces the case study and the integrated algorithmic services. Section 6 reports end-to-end execution, runtime, monitoring, and reproducibility results. Section 7 discusses the scientific position and the limitations of the software-system contribution. Section 8 concludes the paper.

2. Design Requirements for the Software Suite

2.1. Functional Requirements

The software suite is intended to support predictive supervisory control of an energy system operating on continuously updated telemetry. This objective implies a functional scope broader than that of a standalone model server or an offline optimizer. At minimum, the suite must receive raw measurements, align them on the operational time axis, validate them against basic plausibility rules, transform them into a feature-ready state description, build a multi-step state forecast, solve a constrained supervisory decision problem, emit a setpoint recommendation, and record the full cycle for later inspection.

This functional decomposition has two implications. First, prediction and control cannot be represented as isolated modules that exchange only final numeric arrays. They must be embedded in a broader execution context that controls the origin, quality, and timing of the data consumed by both services. Second, monitoring and lifecycle support are not optional additions. Once the output of the suite

affects a supervisory decision, runtime behavior, model validity, and configuration consistency become part of the result that the software system must produce.

In the implemented suite, these functional requirements are operationalized through an explicit module decomposition, typed runtime objects, and persistent storage of the intermediate artifacts exchanged between telemetry, forecasting, optimization, monitoring, and release-management services.

2.2. Operational Requirements

The suite must also satisfy operational requirements that do not follow automatically from algorithmic correctness. The execution cycle must be temporally causal, because the forecasting and optimization services are evaluated under the same information pattern that would be available at decision time. The cycle must remain computationally feasible within the supervisory update interval. The suite must handle missing or delayed measurements without silently contaminating downstream calculations. It must expose safeguard logic for infeasible or numerically unstable optimization runs. Finally, it must preserve enough modularity to update the predictive core, the optimization settings, or the telemetry mapping without rewriting the entire system.

The case study concerns a thermal-energy object with an upper-level supervisory update interval of five minutes. Under such conditions, even a numerically accurate controller is not deployable if its execution time, error-handling strategy, or configuration management remain opaque. The architecture therefore has to be designed for repeated execution under fixed timing and traceability constraints rather than for one-off numerical demonstrations.

These operational requirements are enforced by cycle scheduling, freshness and completeness checks on the telemetry window, safeguard logic for non-nominal optimization runs, and profiled execution stages whose runtime can be compared directly with the supervisory interval.

2.3. Reproducibility and Lifecycle Requirements

The third requirement block concerns reproducibility and lifecycle control. Each execution cycle must be attributable to a concrete telemetry window, a concrete released model bundle, a concrete control configuration, and a concrete runtime log. This condition is needed both for scientific repeatability and for operational diagnostics. Without it, it is impossible to reconstruct why a given setpoint was produced, whether it was computed under the expected model release, and whether runtime anomalies or degraded data quality influenced the result.

In addition, the suite must support a controlled transition from one released model bundle to another. Telemetry archives, historical decision logs, delayed quality indicators, and configuration manifests must therefore be structured as lifecycle inputs rather than as unrelated auxiliary files. This requirement distinguishes a software suite from a collection of scripts. The former preserves operational continuity while allowing retraining, re-identification, validation, and release update; the latter does not.

These requirements are enforced through released bundles, immutable manifests attached to each completed cycle, and persistent records that bind the active forecasting bundle, control configuration, runtime summary, and monitoring outputs to each emitted supervisory decision.

Table 1 summarizes the design requirements adopted in the present study. Together, these requirements define the software-system problem addressed by the article. The suite is expected to execute one control cycle reliably, but it is also expected to preserve the conditions under which future cycles, updated releases, and retrospective analyses remain reproducible.

Table 1. Design requirements for the predictive control software suite.

Requirement group	Requirement	Operational role
Functional scope	Telemetry ingestion, preprocessing, forecasting, optimization, monitoring, lifecycle management	Ensures that the suite covers the full path from incoming measurements to supervisory decision and subsequent maintenance actions.

Temporal causality	Strict use of historically available observations at each execution time	Prevents leakage of future information into both prediction and control decisions.
Service modularity	Separation of data, forecasting, optimization, monitoring, and release-management components	Allows the forecasting and optimization cores to be replaced or updated without redesigning the full system.
Constraint awareness	Explicit support for operational bounds, safeguards, and fallback logic	Keeps generated setpoints admissible in the supervisory layer and limits unsafe or nonconvergent decisions.
Runtime feasibility	Stable end-to-end execution within the 5 min supervisory update interval	Makes repeated quasi-real-time operation possible instead of limiting the suite to offline experiments.
Reproducibility	Persistent storage of data windows, model versions, configuration manifests, and runtime logs	Makes each reported execution cycle traceable and repeatable.
Lifecycle support	Monitoring of quality degradation and controlled release of updated model packages	Connects operational use with retraining, re-identification, and validation.

3. Architecture of the Software Suite

3.1. Overall Architecture

The software suite is organized as a modular architecture with six interacting layers: the physical object, the telemetry and preprocessing layer, the forecasting service, the optimization and supervisory decision service, the monitoring layer, and lifecycle support. The overall architecture is shown in Figure 1. The suite does not treat these layers as a linear chain of scripts. Instead, it organizes them as a closed loop in which operational execution, monitoring, and release update are explicitly connected.

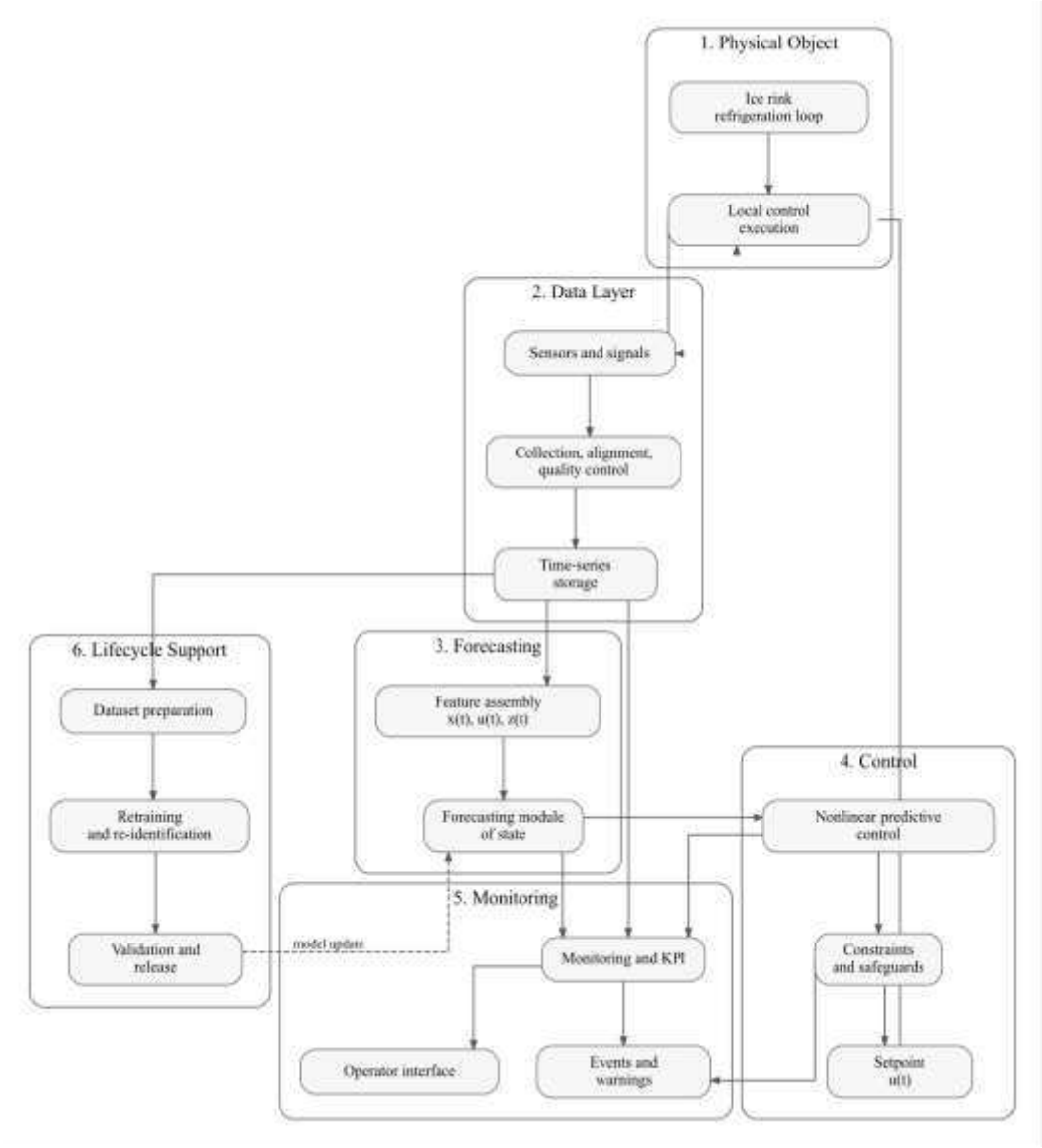


Figure 1. Overall architecture of the predictive control software suite.

The architecture can be expressed as the following composition of mappings:

$$\mathcal{D}_{\text{raw}} \overset{\Psi_{\text{data}}}{\rightarrow} \mathcal{D}_{\text{feat}} \overset{\mathcal{F}_\theta}{\rightarrow} \hat{X}_{t+1:t+H} \overset{\mathcal{O}_{\text{ctrl}}}{\rightarrow} u_t^* \overset{\Psi_{\log}}{\rightarrow} \mathcal{R}_t, \tag{1}$$

where $\mathcal{D}_{\text{raw}}$ denotes incoming raw telemetry, Ψ_{data} is the telemetry-ingestion and preprocessing layer, $\mathcal{F}_\theta$ is the released forecasting service, $\hat{X}_{t+1:t+H}$ is the predicted state trajectory on the supervisory horizon, $\mathcal{O}_{\text{ctrl}}$ is the constrained optimization service, u_t^* is the recommended supervisory output, and $\Psi_{\log}$ maps the completed cycle to a runtime and artifact record $\mathcal{R}_t$.

Equation (1) is intentionally simple. Its role is not to reproduce the full mathematics of the predictive model or the optimizer, but to formalize the boundary of the software system studied in this paper. The architecture is therefore defined by the composition of service-level mappings rather than by the internals of a single algorithmic core.

3.2. Telemetry Ingestion and Preprocessing Layer

The telemetry layer transforms raw measurements into a synchronized, quality-controlled, and feature-ready representation. This layer enforces temporal alignment, signal selection, plausibility checks, and formation of the historical window consumed by the forecasting service. It also stores intermediate artifacts required for traceability, including the synchronized minute-level table and the results of quality checks.

This layer plays a dual role. Operationally, it provides a consistent input to the forecasting and optimization services. Scientifically, it preserves the exact data context of each reported execution cycle. For predictive control, these two roles are inseparable. If the time policy or the signal-alignment logic changes silently between training, validation, and online inference, then neither runtime analysis nor control evaluation remains interpretable.

3.3. Forecasting Service

The forecasting service receives the prepared history window together with the admissible future supervisory candidate needed by the control problem. It returns a multi-step state trajectory on the control horizon. In the present suite, the forecasting core is a control-oriented state model released as a service bundle rather than as a bare weight file. The service bundle includes the model parameters, the preprocessing configuration, and the release metadata used to identify the exact predictive core active during a decision cycle.

This service-oriented representation is important for two reasons. First, it decouples the predictive core from the surrounding orchestration logic. Second, it allows the predictive model to be replaced or updated through a controlled release mechanism without redesigning the telemetry or optimization layers. The forecasting service is therefore a software interface with versioned inputs and outputs, not merely an algorithm invoked by an experimental notebook.

3.4. Optimization and Control Decision Service

The optimization service consumes the forecasting-service interface, evaluates the control objective under operational bounds, and produces a supervisory setpoint recommendation. In architectural terms, it is a constrained decision service rather than a mathematical appendix to the predictive model. It must therefore expose not only the recommended value, but also the solver status, the runtime statistics, and the safeguard state associated with the decision.

This design prevents a common failure mode of research prototypes: a nominally successful optimization routine that cannot be diagnosed when it slows down, fails, or produces an inadmissible output. In the present suite, the optimization stage is treated as a service that participates fully in logging, monitoring, and lifecycle management.

3.5. Monitoring, Diagnostics, and Lifecycle Management

The monitoring layer receives forecasts, supervisory outputs, runtime traces, and delayed realized measurements. It aggregates these signals into key performance indicators, event flags, and maintenance cues. The lifecycle layer uses those outputs together with the accumulated telemetry archive to decide whether retraining, re-identification, or release validation is required.

This structure links immediate operational observability with longer-horizon model maintenance. The suite therefore does not reduce monitoring to a dashboard. Monitoring is part of the control system itself because it generates the conditions under which the next valid model release can be prepared and audited. Figure 5 shows the lifecycle loop implemented in the suite.

3.6. Artifact Storage and Reproducibility Support

Artifact storage is treated as an independent architectural component. It persists data windows, model bundles, control configurations, runtime records, and release manifests. This persistence layer allows a completed cycle to be reconstructed post hoc and enables controlled comparison between released versions of the forecasting and optimization services.

Table 2 summarizes the main modules, their interfaces, and the artifacts stored by the suite. The table also makes the central architectural distinction of the paper explicit. The suite is not defined only by what it computes during one cycle, but also by what it stores and exposes for later verification, diagnosis, and controlled release updates.

Table 2. Modules of the software suite, their interfaces, and stored artifacts.

Module	Main inputs	Main outputs	Stored artifacts

Telemetry ingestion and preprocessing	Raw telemetry stream, signal mapping, time policy	Synchronized minute-level table, feature-ready history windows, quality-control flags	Data snapshots, preprocessing configuration, quality-control reports
Forecasting service	Historical window, admissible future control candidate, released model bundle	Multi-step state trajectory on the prediction horizon	Model weights, inference configuration, forecast records
Optimization and control decision service	Forecasting service handle, control bounds, objective configuration	Recommended supervisory setpoint, solver status, optimization trace	Controller configuration, runtime profile, solution trace
Monitoring and diagnostics	Forecasts, decisions, realized delayed measurements, runtime signals	Key performance indicators, warnings, maintenance events	Runtime logs, monitoring summaries, incident history
Lifecycle management	Accumulated telemetry archive, decision log, released bundle metadata	Updated model package, validation report, release manifest	Release notes, parameter manifests, retraining history

Table 3 makes the runtime contracts of the implemented architecture explicit. The software suite is therefore defined not only by module names, but also by the typed objects exchanged between services, the mandatory metadata attached to these objects, the admissible failure modes, and the artifacts persisted after each stage.

Table 3. Service contracts and runtime entities of the implemented software suite.

Component	Input object	Output object	Mandatory metadata	Failure mode	Stored artifact
Telemetry acquisition	Raw measurement slice	Time-stamped telemetry window	Window end time, source identifier, freshness flag	Missing or stale telemetry window	Telemetry snapshot
Preprocessing layer	Telemetry window, signal mapping, alignment policy	Aligned history representation, quality flags	Mapping version, alignment policy identifier, missingness summary	Schema mismatch or failed plausibility check	Minute-level table, quality report
Forecasting service	Aligned history, released forecasting bundle, candidate control trajectory	Multi-step state trajectory	Bundle version, inference timestamp, horizon identifier	Bundle unavailable or inference exception	Forecast record
Optimization service	Forecast trajectory, controller configuration, control bounds	Supervisory candidate, solver trace	Configuration hash, solver status, runtime summary	Infeasible search or non-successful nominal termination	Solution trace, runtime record

Safeguard layer	Supervisory candidate, previous admissible setpoint, warning state	Emitted supervisory record	Warning flags, fallback reason, emission timestamp	Candidate rejected by safeguard logic	Decision record
Monitoring service	Runtime record, delayed observations, execution anomalies	Monitoring summary, maintenance cues	Cycle identifier, release identifier, monitoring timestamp	Late or incomplete delayed feedback	Monitoring summary, incident history
Release registry	Candidate bundle, validation report, manifest	Released bundle descriptor	Bundle version, validation status, changelog	Failed validation or incomplete manifest	Release manifest

4. Implementation Logic

4.1. End-to-End Operational Workflow

The operational workflow begins with telemetry acquisition and ends with a supervisory setpoint together with a recorded execution trace. The sequence is shown in Figure 2. First, raw measurements are aligned and validated. Second, the forecasting service constructs a state trajectory over the prediction horizon. Third, the optimization service evaluates admissible supervisory candidates and selects the setpoint to be emitted to the upper control layer. Fourth, the suite stores the resulting forecast, decision record, runtime summary, and monitoring signals.

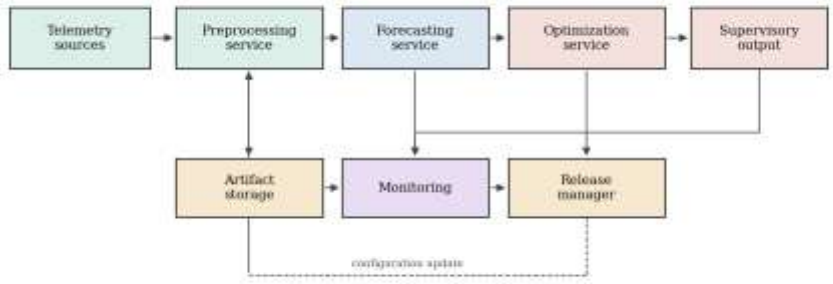


Figure 2. End-to-end workflow from telemetry to supervisory action.

This workflow contains both synchronous and asynchronous elements. Telemetry alignment, forecasting, optimization, and decision emission belong to the synchronous operational cycle. Monitoring aggregation, retrospective quality assessment, retraining preparation, and release validation are asynchronous or slower-cycle processes. Their separation is deliberate. It preserves fast operational execution without sacrificing lifecycle control.

4.2. Orchestration and Scheduling

The orchestration logic is driven by the supervisory update interval. At each update time, the suite builds one causally valid execution window, triggers the forecasting service, invokes the optimization service, applies safeguard checks, and records the outcome. If the data layer or the optimization layer reports a blocking anomaly, the cycle is annotated accordingly and the suite can retain the last admissible supervisory setting rather than emit an opaque failure.

This scheduling model turns the predictive-control problem into a repeated service cycle with fixed temporal boundaries. The benefit of such a formulation is that runtime feasibility, warning logic, and

operational continuity become measurable properties of the software system rather than implicit assumptions.

Algorithm 1. Supervisory execution cycle.

1. Acquire the telemetry window for the current supervisory timestamp.
2. Validate completeness, freshness, and schema consistency of the acquired measurements.
3. Build the aligned state and history representation required by the forecasting service.
4. Load the active released bundle and the associated control configuration.
5. Run the forecasting service on the current history representation.
6. Run the optimization service on the forecast and operational constraints.
7. Apply safeguard and fallback logic if the nominal optimization result is rejected.
8. Emit the supervisory setpoint record to the upper control layer.
9. Persist the forecast, decision trace, runtime record, and release identifiers.
10. Update monitoring outputs and maintenance indicators for the completed cycle.

4.3. Configuration Management and Release Logic

Configuration management is organized around released bundles. A released bundle contains the active forecasting model, its preprocessing parameters, the optimization settings, and the metadata needed to identify the release used in a decision cycle. The transition from one bundle to the next is represented as

$$\mathcal{V}_{k+1} = \Omega\left(\mathcal{V}_k, \mathcal{D}_k^{\text{new}}, \mathcal{R}_k^{\text{hist}}\right), \tag{2}$$

where $\mathcal{V}_k$ is the current released bundle, $\mathcal{D}_k^{\text{new}}$ is the newly accumulated telemetry archive, and $\mathcal{R}_k^{\text{hist}}$ is the historical execution and monitoring record.

Equation (2) formalizes lifecycle support as a controlled software operation. The role of Ω is to encapsulate retraining, re-identification, validation, and release issuance without breaking the traceability of the already completed operational cycles. This distinction is important because a predictive-control deployment cannot be evaluated solely by the quality of its current release; it must also be evaluated by the clarity of its release-transition mechanism.

In the implemented suite, a released bundle includes model weights, preprocessing parameters, signal mapping and normalization settings, controller bounds, objective weights, horizon settings, and a validation manifest. Once promoted to an active release, the bundle contents that affect forecasting and supervisory computation remain immutable for all cycles executed under that release. Between releases, the model weights, preprocessing parameters, controller settings, and validation report may change, whereas the release identifier, activation timestamp, and changelog record the transition itself. Validation is initiated after an offline review of accumulated telemetry, delayed forecast-quality indicators, and supervisory execution logs. Promotion of a candidate release is blocked by failed forecast validation, failed supervisory validation, schema incompatibility, incomplete manifest metadata, or unresolved anomalies in the telemetry-to-feature mapping.

4.4. Interfaces with the Supervisory Layer

The software suite operates above the local control and actuation level. Its output is a supervisory setpoint rather than a direct low-level actuation command. This interface choice keeps the suite compatible with existing local stabilization logic and allows deployment as an upper decision layer. It also constrains the software-system responsibility: the suite must provide a causally valid, runtime-feasible, and auditable supervisory recommendation, while the lower control layer remains responsible for direct execution on the physical plant.

The interface contract therefore includes more than the numeric setpoint itself. The emitted record must also include the timestamp, the active release identifier, the status of the optimization service, the runtime summary, and the warning state. Without this contract, the software system would not provide enough information for monitoring, rollback, or post hoc diagnosis.

Table 4. Supervisory output record schema emitted by the software suite.

Field	Example type	Role in the supervisory contract
timestamp	UTC or local time stamp	Identifies the decision instant for the emitted record.
cycle_id	Structured string	Links the output to the exact execution cycle and runtime log.
release_id	Bundle identifier	Fixes the active released forecasting bundle used in the cycle.
config_hash	Short hexadecimal hash	Identifies the active control configuration applied by the optimizer.
solver_status	success / warning	Distinguishes nominal solver termination from a warning state.
warning_state	none / flagged	Summarizes whether safeguard logic or runtime warnings were raised.
setpoint_c	Real-valued scalar	Contains the supervisory setpoint emitted to the upper control layer.
runtime_sec	Real-valued scalar	Records the profiled full-cycle runtime associated with the emitted decision.

Table 4 summarizes the supervisory output record emitted by the suite. The supervisory interface is therefore defined as a structured runtime contract rather than a single scalar command.

5. Case Study

5.1. Pilot Object and Available Signals

The case study concerns an indoor ice rink refrigeration system operated as a thermally inert supervisory-control object. The available telemetry includes temperatures in the glycol loop and the slab, compressor power, meteorological and indoor environmental measurements, and operating-context signals required for forecasting and optimization. The software suite processes these signals at minute resolution and issues supervisory updates every 5 min.

This object is appropriate for the present article because it combines all software-system difficulties that motivate the suite: heterogeneous telemetry, a control-relevant thermal horizon, hard operational bounds, and a need for repeated execution under a fixed time budget. The case study is therefore used to test the software architecture under realistic telemetry and supervisory conditions rather than to reopen the full algorithmic comparison problem.

5.2. Algorithmic Services Integrated into the Suite

The forecasting and optimization cores are treated here as integrated algorithmic services within the software suite. Their full mathematical development is reported in companion manuscripts and is not repeated here. The forecasting service uses a control-oriented state model released as a reusable service bundle. The optimization service uses a receding-horizon nonlinear predictive-control formulation with constrained supervisory search. Gradients required by the optimization service are obtained by automatic differentiation of the forecasting core, and the resulting constrained problem is solved by sequential least-squares quadratic programming.

The suite uses a 180 min historical window and a 30 min prediction horizon. The supervisory output is parameterized as six 5 min control blocks. These settings are not presented here as the primary scientific result. They define the operational context within which the software-system layer is evaluated.

5.3. Evaluation Protocol

The evaluation protocol follows the execution logic of the suite rather than an isolated algorithmic benchmark. The analysis focuses on four aspects: end-to-end execution of the operational workflow, runtime profile of the service chain, monitoring and maintenance behavior, and reproducibility of completed cycles. Runtime measurements are collected for preprocessing, optimization, and the full cycle.

Monitoring outputs are interpreted in relation to data quality, solver status, delayed forecast quality, and maintenance cues. Reproducibility is assessed through the stored artifacts attached to each run and each release.

5.4. Computational Environment

The case study was executed on a Dell PowerEdge R740 server equipped with two Intel Xeon Gold 6148 processors, corresponding to 40 physical cores and 80 execution threads, 768 GB of RAM, and two NVIDIA Tesla V100 accelerators with 32 GB of memory each. The supervisory pipeline did not require graphics-processor acceleration. All reported operational runs were executed in a Python-based stack in CPU mode, with telemetry handling and preprocessing implemented in standard numerical and table-processing libraries, the forecasting service implemented in PyTorch, and the supervisory optimizer executed through the SciPy implementation of sequential least-squares quadratic programming. The installed accelerators were available for auxiliary model-development tasks, but were not required for the runtime results reported in Section 6.

6. Results

6.1. End-to-End Execution of the Software Suite

The suite executes a complete data-to-decision cycle at every supervisory update time. One completed cycle includes a synchronized telemetry window, a feature-ready history representation, a multi-step predicted state trajectory, a recommended supervisory setpoint, a runtime record, and a monitoring record. In this sense, the suite produces not only a decision value, but also the computational context that makes the decision auditable.

The workflow in Figure 2 is therefore not merely a conceptual diagram. It corresponds to the actual order of execution in the case study. The telemetry layer prepares the historical input. The forecasting service returns the predicted trajectory required by the optimizer. The optimization service produces a supervisory setpoint under operational constraints. Monitoring and artifact storage record the resulting cycle. This end-to-end behavior is the primary empirical object of the paper.

The implemented maturity of the suite is heterogeneous and is reported explicitly. The telemetry-alignment logic, bundle loading, forecasting call, optimization call, supervisory output emission, runtime logging, and artifact persistence are fully implemented and profiled in the present case study. Delayed-quality monitoring, release review, and maintenance reporting are implemented in prototype form through the same runtime artifacts and offline validation utilities used to prepare updated bundles. Cross-object deployment, multi-site orchestration, and automated promotion of releases across several operational objects remain architectural extensions rather than experimentally evaluated deployment features.

6.2. Runtime Profile and Computational Feasibility

Computational feasibility is assessed against the 5 min supervisory interval. Table 5 reports the runtime profile of the principal execution stages. Figure 3 shows the distribution of mean, 95th percentile, and maximum runtime values by stage, whereas Figure 4 places the maximum optimization and full-cycle runtimes directly against the available supervisory time budget.

Table 5. Runtime profile of the end-to-end execution cycle.

Stage	Mean, s	95th percentile, s	Maximum, s	Share of 5 min interval, %
Preprocessing	0.000002	0.000002	0.000015	0.000005
Optimization	0.031067	0.052185	0.106362	0.035454
Full cycle	0.031069	0.052187	0.106364	0.035455

Table 6. Operational execution metrics of the profiled supervisory panel.

Metric	Value	Interpretation
Executed supervisory cycles	2304	Total number of profiled cycles in the evaluated panel.
Successful cycles	2304	Cycles that produced an admissible supervisory output record.

Cycles with solver warnings	113	Completed cycles with non-successful nominal solver termination and accepted safeguarded output.
Cycles with fallback activation	0	Cycles that required reuse of the previous admissible setpoint.
Cycles blocked by data-quality issue	0	Cycles prevented from downstream execution by telemetry-quality checks.
Mean preprocessing time, s	0.000002	Average runtime of telemetry alignment and state-history construction.
Mean optimization time, s	0.031067	Average runtime of constrained supervisory optimization.
Mean logging overhead, s	below profiler resolution	Persistence overhead remained below the resolution of the profiled runtime wrapper.

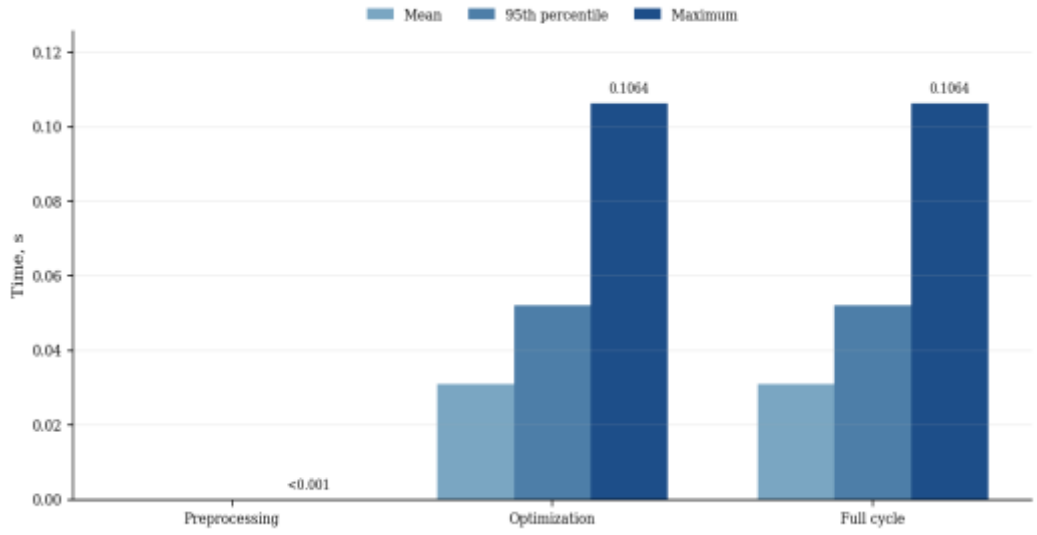


Figure 3. Runtime profile of the execution stages.

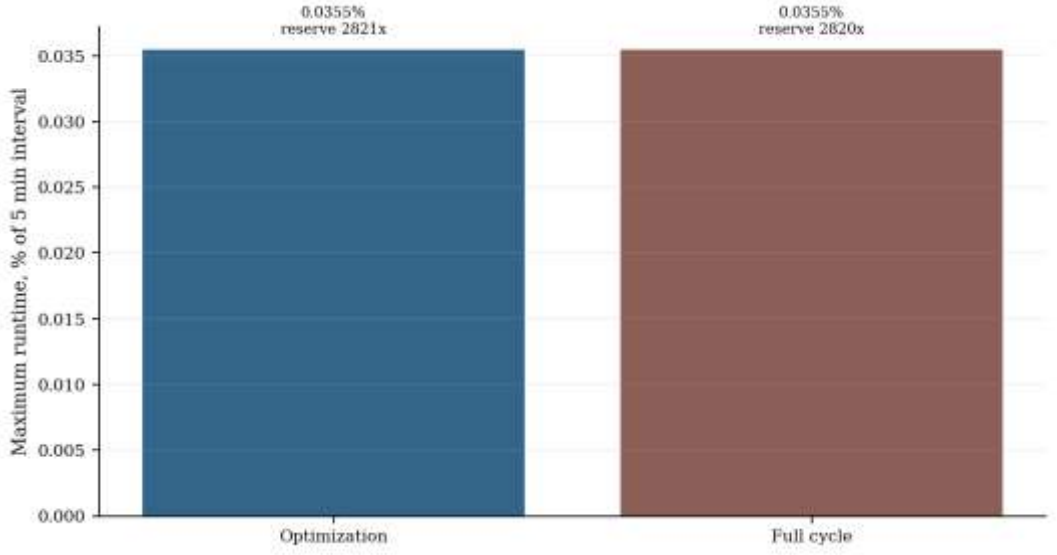


Figure 4. Use of the supervisory time budget.

The results show that preprocessing contributes negligibly to the total runtime, whereas the optimization stage dominates the cycle. The mean full-cycle runtime is 0.0311 s, the 95th percentile is 0.0522 s, and the maximum observed runtime is 0.1064 s. This corresponds to only 0.0355% of the 5 min supervisory interval. The practical consequence is a reserve above 2800× even at the worst observed cycle.

The preprocessing stage appears negligible due to vectorized in-memory operations and preloaded data structures; its measured runtime is at the lower resolution limit of the profiler.

Table 6 complements the stage-wise runtime profile with system-level execution counts. Across 2304 supervisory cycles, the suite emitted an admissible supervisory record in every cycle. A total of 113 cycles completed with a solver warning, defined here as non-successful nominal solver termination combined with a safeguarded accepted output. No cycle required fallback reuse of the previous admissible setpoint, and no cycle was blocked by telemetry-quality checks in the evaluated panel. The logging overhead remained below the resolution of the profiled runtime wrapper and therefore appears as zero in the summarized record.

This runtime condition can be stated formally as

$$\tau_{\text{cycle}}^{\max} < \Delta t_{\text{sup}}, \tag{3}$$

where $\tau_{\text{cycle}}^{\max}$ is the maximum observed full-cycle runtime and Δt_{sup} is the supervisory update interval. In the present case, Equation (3) is satisfied with a wide margin. The software-system result is therefore not limited to formal orchestration. It is supported by measured runtime feasibility under repeated execution.

6.3. Monitoring Outputs and Maintenance Logic

The monitoring layer produces quality indicators that connect operational execution with maintenance decisions. These indicators include data freshness, missingness, preprocessing integrity, solver status, runtime excursions, delayed forecast-quality degradation, and distribution shifts in key variables. The suite does not treat these signals as passive diagnostics. They are used to decide whether a cycle should be annotated, whether safeguard logic should be applied, and whether a release review should be scheduled.

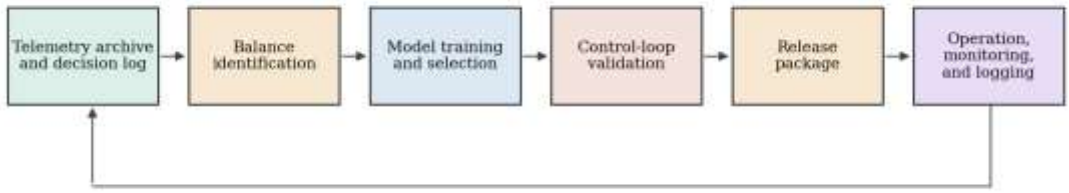


Figure 5. Lifecycle loop for continuous refinement of the software suite.

Table 7. Monitoring indicators and maintenance triggers implemented in the software suite.

Indicator	Observed layer	Trigger logic	Resulting action
Data freshness and missingness	Telemetry ingestion	Delayed or incomplete measurement window	Warning, cycle annotation, or fallback to the last admissible supervisory value
Preprocessing integrity	Data service	Failed alignment, schema mismatch, or implausible value range	Block downstream execution and request data review
Execution latency	Optimization service	Sustained growth of cycle time or solver time	Runtime warning and scheduling review
Solver status	Optimization service	Repeated nonconvergence or infeasible candidate sequences	Activate safeguard logic and inspect configuration or model version
Delayed forecast quality	Forecasting service	Systematic deterioration once delayed ground truth becomes available	Flag model degradation and schedule validation or retraining
Distribution shift in key variables	Monitoring and lifecycle layer	Persistent change in operating regime or disturbance distribution	Trigger release review and update preparation

Figure 5 and Table 7 show the connection between operational monitoring and release management. The suite maintains a distinction between immediate execution anomalies and slower-cycle model maintenance cues. This distinction prevents the operational layer from being overloaded with retraining logic while still ensuring that accumulated evidence can trigger controlled release updates.

The monitored execution panel shows that warning events were sparse relative to the total number of completed cycles and were concentrated in cycles with prolonged optimization runs. At the same time, the absence of telemetry-quality blocks and fallback activations confirms that the runtime safeguards were structured as preventive controls rather than as frequently triggered emergency routines. The monitoring layer therefore serves two functions simultaneously: it annotates the few non-nominal cycles that require later inspection and preserves a low-noise evidence base for release review.

6.4. Reproducibility and Traceability

The suite attaches a reproducibility record to every completed cycle. This record includes the input data window, the released forecasting bundle, the active control configuration, the runtime summary, the monitoring summary, and the release manifest associated with the active version. Table 8 summarizes the tracked entities.

Table 8. Reproducibility entities tracked by the software suite.

Tracked entity	Examples	Role in reproducibility
Input data window	Time-stamped telemetry slice, synchronized minute table, quality-control flags	Identifies the exact operational context of a decision cycle.
Forecasting bundle	Model weights, preprocessing parameters, released metadata	Fixes the predictive core used during inference and optimization.
Control configuration	Bounds, penalty weights, horizon settings, safeguard parameters	Reconstructs the optimization conditions that produced a given setpoint.
Runtime log	Stage durations, solver status, warning flags, cycle identifier	Makes computational feasibility and anomalies auditable.
Monitoring summary	KPI values, drift-like events, delayed quality checks	Connects operational performance with maintenance decisions.
Release manifest	Validation report, bundle version, changelog, activation time	Enables controlled rollback, comparison of releases, and lifecycle traceability.

Table 9. Example of one reconstructed supervisory cycle.

Field	Value
cycle_id	nmpc_phys_warm@2025-02-02T00:05:00
timestamp	2025-02-02 00:05:00
telemetry window	180 min history ending at 2025-02-02 00:05:00
bundle_version	conditioned_phys_0p001.pt
config_hash	f4070847a291
solver_status	success
output setpoint	-6.5820 °C
warning flags	none
runtime	0.032867 s

monitoring summary	freshness=ok; missingness=0; solver_warning=0; fallback=0

This artifact structure makes it possible to reconstruct a completed cycle from recorded telemetry context, released software state, and runtime metadata. Table 9 illustrates such a reconstruction for one completed supervisory cycle and shows that the telemetry context, active bundle, configuration hash, solver outcome, emitted setpoint, warning state, runtime, and monitoring summary can be read back as one coherent runtime record. The scientific implication is that the reported runtime and monitoring results are tied to concrete software and data artifacts rather than to undocumented execution conditions. The engineering implication is that rollback, comparison of releases, and audit of past decisions remain possible without re-running the entire historical workflow from scratch.

7. Discussion

7.1. Software-System Contribution versus Algorithmic Contribution

The central result of the present work is a software-system contribution rather than a new predictive model or a new optimization algorithm. The forecasting and optimization cores are necessary components of the suite, but they are not sufficient to produce a deployable predictive-control workflow. The software contribution lies in the way telemetry preparation, service decomposition, runtime control, monitoring, artifact storage, and release management are organized into one executable architecture.

In data-centric control research, algorithmic novelty and software-system novelty are often conflated. The present study argues that they should be separated analytically. A forecasting or control method can be strong in isolation while the surrounding system remains undeployable or unreproducible. Conversely, a modular and traceable software suite can transform existing predictive and optimization methods into an operationally meaningful workflow. The case study supports the second type of contribution.

7.2. Portable and Object-Specific Parts of the Architecture

Not all elements of the suite are equally specific to the ice rink domain. The signal mapping, some safeguard rules, the state composition used by the forecasting core, and the operational interpretation of the supervisory setpoint are object-specific. In contrast, the service decomposition, the data-to-decision mapping in Equation (1), the lifecycle operator in Equation (2), the runtime condition in Equation (3), the artifact structure, and the monitoring logic are portable architectural ideas.

This separation is important for reuse. The architecture is designed so that another thermal-energy system can replace the signal map, the forecasting bundle, and the optimization configuration while preserving the same software-system logic. The case study therefore supports transferability at the architectural level even though the present article evaluates only one object.

7.3. Limitations

The study has several limitations. First, the reported results concern a single operational object, namely an indoor ice rink refrigeration system. Second, the article evaluates the software suite through runtime, monitoring, and traceability evidence rather than through a multi-site deployment campaign. Third, the lifecycle triggers are operationally meaningful but are not benchmarked here against alternative monitoring frameworks or release strategies. Fourth, the forecasting and optimization cores are treated as integrated services; their mathematical comparison with alternative algorithms is outside the scope of the article.

These limitations do not diminish the software-system contribution, but they do define its current boundary. The present paper demonstrates a reproducible and computationally feasible predictive-control suite. It does not claim that the same concrete release package is already universally deployable across all industrial energy systems.

8. Conclusions

This paper presented a software suite for predictive control of energy consumption systems and analyzed it as a modular, reproducible, and computationally feasible software system. The suite integrates

telemetry ingestion, preprocessing, forecasting, optimization, supervisory output generation, monitoring, artifact storage, and lifecycle support in one operational workflow. Its architecture was formalized through service-level mappings and a release-update operator, and its execution was demonstrated on an indoor ice rink refrigeration system.

The reported results show that the suite is capable of end-to-end execution with a mean full-cycle runtime of 0.0311 s and a maximum runtime of 0.1064 s under a 5 min supervisory interval. The suite also records the data, model, configuration, runtime, and release artifacts required for traceability and controlled maintenance. These properties position the software system as a scientific contribution in its own right. The approach can be transferred to other energy systems by replacing the object-specific signal mapping and algorithmic bundles while preserving the same architectural and lifecycle logic.

Author Contributions: Conceptualization, P.N., R.I.; methodology, P.N.; software, A.K.; validation, A.K., R.I.; formal analysis, A.K.; investigation, A.K.; data curation, A.K.; writing—original draft preparation, A.K.; writing—review and editing, P.N., R.I.; visualization, A.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The raw operational data used in this study are not publicly available because they originate from an industrial facility and contain commercially sensitive information. Derived datasets, trained model artifacts, controller-evaluation outputs, figure-generation scripts, and manuscript export scripts are available from the corresponding author on reasonable request. The code archive can also be provided to editors and reviewers for confidential peer-review purposes upon request.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Peres, R.S.; Jia, X.; Lee, J.; Sun, K.; Colombo, A.W.; Barata, J.T. Industrial Artificial Intelligence in Industry 4.0—Systematic Review, Challenges and Outlook. *IEEE Access* **2020**, 8, 220121–220139. https://doi.org/10.1109/ACCESS.2020.3042874.
2. Wetter, M.; Ehrlich, P.; Gautier, A.; Grahovac, M.; Haves, P.; Hu, J.; Prakash, A.; Robin, D.; Zhang, K. OpenBuildingControl: Digitizing the Control Delivery from Building Energy Modeling to Specification, Implementation and Formal Verification. *Energy* **2022**, 238, 121501. https://doi.org/10.1016/j.energy.2021.121501.
3. Paul, L.; De Andrade Pereira, F.; Prakash, A.; Ham, S.W.; Feng, J.; Brown, R.; Pritoni, M. Open Building Operating System: A Grid-Responsive Semantics-Driven Control Platform for Buildings. *Science and Technology for the Built Environment* **2025**, 31(3). https://doi.org/10.1080/23744731.2024.2444819.
4. Blum, D.; Arroyo, J.; Huang, S.; Drgoňa, J.; Jorissen, F.; Walnum, H.T.; Chen, Y.; Benne, K.; Vrabie, D.; Wetter, M.; Helsen, L. Building Optimization Testing Framework (BOPTEST) for Simulation-Based Benchmarking of Control Strategies in Buildings. *Journal of Building Performance Simulation* **2021**, 14(5), 586–610. https://doi.org/10.1080/19401493.2021.1986574.
5. Ardebili, A.A.; Zappatore, M.; Ramadan, A.I.H.A.; Longo, A.; Ficarella, A. Digital Twins of Smart Energy Systems: A Systematic Literature Review on Enablers, Design, Management and Computational Challenges. *Energy Informatics* **2024**, 7, 94. https://doi.org/10.1186/s42162-024-00385-5.
6. Duque Gómez, H.; García-Rodríguez, J.; Azorín-López, J.; Tomás, D.; Fuster-Guilló, A.; Mora-Mora, H. IA-CPS: Intelligent Architecture for Cyber-Physical Systems Management. *Journal of Computational Science* **2021**, 53, 101409. https://doi.org/10.1016/j.jocs.2021.101409.

7. Aldalur, I.; Perez, J.; Muguerza, J.; Arana-Arexolaleiba, N.; Etxeberria-Agiriano, I. A Microservice-Based Framework for Multi-Level Testing of Cyber-Physical Systems. *Software Quality Journal* **2024**. https://doi.org/10.1007/s11219-023-09639-z.
8. Mena, M.; Criado, J.; Iribarne, L.; et al. Towards High-Availability Cyber-Physical Systems Using a Microservice Architecture. *Computing* **2023**, 105, 1745–1768. https://doi.org/10.1007/s00607-023-01165-x.
9. Waseem, M.; Liang, P.; Shahin, M. A Systematic Mapping Study on Microservices Architecture in DevOps. *Journal of Systems and Software* **2020**, 170, 110798. https://doi.org/10.1016/j.jss.2020.110798.
10. Al-Kasassbeh, S.M.; et al. Microservices Based Architectures for IoT Systems: State-of-the-Art Review. *Internet of Things* **2023**, 23, 100854. https://doi.org/10.1016/j.iot.2023.100854.
11. Rosemann, M.H.; et al. Integrating Process Management and Event Processing in Smart Factories: A Systems Architecture and Use Cases. *Journal of Manufacturing Systems* **2022**. https://doi.org/10.1016/j.jmsy.2022.05.012.
12. Zampetti, F.; et al. Continuous Integration and Delivery Practices for Cyber-Physical Systems: An Interview-Based Study. *ACM Transactions on Software Engineering and Methodology* **2022**. https://doi.org/10.1145/3571854.
13. Kreuzberger, D.; Kühl, N.; Hirschl, S. Machine Learning Operations (MLOps): Overview, Definition, and Architecture. *IEEE Access* **2023**, 11, 31866–31879. https://doi.org/10.1109/ACCESS.2023.3262138.
14. Faubel, L.; Schmid, K.; Eichelberger, H. MLOps Challenges in Industry 4.0. *SN Computer Science* **2023**, 4, 828. https://doi.org/10.1007/s42979-023-02282-2.
15. Rani, S.; et al. Industrial Edge MLOps: Overview and Challenges. In *Intelligent Analytics for Industry*; Elsevier: Amsterdam, The Netherlands, **2024**. https://doi.org/10.1016/B978-0-443-28824-1.50504-4.
16. Freitas, N.; Rocha, A.D.; Barata, J. Data Management in Industry: Concepts, Systematic Review and Future Directions. *Journal of Intelligent Manufacturing* **2026**, 37, 799–827. https://doi.org/10.1007/s10845-025-02570-z.
17. Hinder, F.; Vaquet, V.; Hammer, B. One or Two Things We Know about Concept Drift—A Survey on Monitoring in Evolving Environments. Part A: Detecting Concept Drift. *Frontiers in Artificial Intelligence* **2024**, 7, 1330257. https://doi.org/10.3389/frai.2024.1330257.
18. Lukats, D.; et al. A Benchmark and Survey of Fully Unsupervised Concept Drift Detectors on Real-World Data Streams. *International Journal of Data Science and Analytics* **2025**, 19, 1–31. https://doi.org/10.1007/s41060-024-00620-y.
19. Helu, M.M.; Sprock, T.A.; Hartenstine, D.; Venketesh, R.; Sobel, W. Scalable Data Pipeline Architecture to Support the Industrial Internet of Things. *CIRP Annals* **2020**, 69(1), 385–388. https://doi.org/10.1016/j.cirp.2020.04.006.
20. Pérez, M.S.; et al. From the Edge to the Cloud: A Continuous Delivery and Preparation Model for Processing Big IoT Data. *Simulation Modelling Practice and Theory* **2020**, 105, 102136. https://doi.org/10.1016/j.simpat.2020.102136.
21. Peixoto, T.; Oliveira, Ó.; Costa e Silva, E.; Oliveira, B.; Ribeiro, F. A Data Quality Pipeline for Industrial Environments: Architecture and Implementation. *Computers* **2025**, 14(7), 241. https://doi.org/10.3390/computers14070241.
22. Yu, W.; Patros, P.; Young, B.; Klinac, E.; Walmsley, T.G. Energy Digital Twin Technology for Industrial Energy Management: Classification, Challenges and Future. *Renewable and Sustainable Energy Reviews* **2022**, 161, 112407. https://doi.org/10.1016/j.rser.2022.112407.
23. Tao, F.; Sun, X.; Cheng, J.; et al. makeTwin: A Reference Architecture for Digital Twin Software Platform. *Chinese Journal of Aeronautics* **2024**, 37(1), 1–18. https://doi.org/10.1016/j.cja.2023.05.002.
24. De Paepe, D.; Vanden Hautte, S.; Steenwinckel, B.; Moens, P.; Vaneessen, J.; Vandekerckhove, S.; Volckaert, B.; Ongenae, F.; Van Hoecke, S. A Complete Software Stack for IoT

Time-Series Analysis that Combines Semantics and Machine Learning: Lessons Learned from the Dyversify Project. *Applied Sciences* **2021**, 11(24), 11932. https://doi.org/10.3390/app112411932.

25. Drgoňa, J.; Arroyo, J.; Cupeiro Figueroa, I.; Blum, D.; Arendt, K.; Kim, D.; Perarnau Ollé, E.; Oravec, J.; Wetter, M.; Vrabie, D.L.; Helsen, L. All You Need to Know about Model Predictive Control for Buildings. *Annual Reviews in Control* **2020**, 50, 190–232. https://doi.org/10.1016/j.arcontrol.2020.09.001.