The Journal of Systems and Software 241 (2026) 113021

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

PANCS: Proactive self-adaptation for nonlinear cyber–physical systems ☆

Farid Edrisi [a,*], Diego Perez-Palacin [a,b], Mauro Caporuscio [a], Raffaela Mirandola [b]

[a] Linnaeus University, Växjö, Sweden
[b] Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany

ARTICLE INFO

Keywords:
Self-adaptive systems
Proactive adaptation
MAPE-K
Adaptive model predictive control
Runtime adaptation
Nonlinear systems

ABSTRACT

Cyber–Physical Systems (CPS) often exhibit nonlinear behavior and are subject to considerable uncertainty arising from dynamic environmental conditions, hardware degradation, and incomplete system knowledge. While reactive self-adaptation frequently fails to maintain functionality and quality under these conditions, proactive adaptation proves effective, leveraging predictive insights. However, most existing proactive approaches are tailored to information systems and lack systematic architectural support for nonlinear CPS with real-time constraints. This paper presents *PANCS*, a reusable reference architecture grounded in the MAPE–K loop and enhanced with Adaptive Model Predictive Control. *PANCS* enables proactive runtime adaptation in complex, nonlinear CPS by anticipating system behavior and environmental dynamics. It further integrates learning mechanisms to mitigate model-to-reality discrepancies and ensures real-time feasibility through dynamic horizon adjustment and constraint-aware optimization. *PANCS* is evaluated through a warehouse robotic case study using distributed co-simulation under realistic uncertainty scenarios. As a reference architecture, *PANCS* is not domain-specific and its reusability is illustrated in two dimensions; integration within larger adaptive ecosystems through established MAPE–K decentralized patterns, and conceptual instantiation of its architectural components across two additional CPS domains.

1. Introduction

Cyber–Physical Systems (CPS) are systems that combine computing, networking, and physical elements into an integrated system. The main characteristic of CPS is the close interaction of both hardware and software resources for computational, communication, and control purposes, all of which are co-designed with the physically engineered components (Zacchia Lun et al., 2019). Nowadays, CPS are increasingly prevalent across a wide range of domains, including healthcare, robotics, smart grids and renewable energy, intelligent transportation systems, and avionics (Weyns et al., 2022).

However, engineering CPS is challenging since they inherently operate under various sources of uncertainty (Broy and Schmidt, 2014). These uncertainties may arise internally, from factors such as configuration changes, software or hardware faults, and component failures, or externally, from unpredictable environmental conditions and human interactions (Azari et al., 2025). Furthermore, sensing limitations often restrict CPS to operate with only partial knowledge of their surroundings, adding another layer of uncertainty (Camilli et al., 2021). To cope with these changes and uncertainties autonomously, a viable strategy is to enhance the system with self-adaptive capabilities (Musil et al., 2017). Systems with self-adaptation abilities can modify their behavior and/or structure based on their perception of the environment, the system itself, and their goals (de Lemos et al., 2013). Adaptation is often carried out through a feedback loop in self-adaptive systems (Weyns, 2020).

The traditional approach for designing self-adaptation is the *reactive* approach, which is a responsive method triggered by specific events or changes that cause a violation of the adaptation goals and necessitate a response (Weyns, 2020). Since this response is applied after a change or violation has already occurred, to be effective, it must happen instantaneously (Moreno et al., 2015); otherwise, by the time the adaptation is performed, its effectiveness may already be reduced (Vilchez et al., 2024). However, generating high-quality runtime adaptations often involves complex computations or large search spaces (Pandey et al., 2016), which introduce non-negligible latency (Moreno et al., 2016)—conflicting with the requirement for immediacy. Consequently, reactive systems may tend to compromise adaptation quality, prioritizing short-term, locally optimal responses at the expense of long-term, globally optimal behavior.

Proactive adaptation emerges as an alternative temporal dimension of adaptation, aiming to prepare for adaptation before it becomes required, based on predictive insights (Krupitzer et al., 2015). Ideally, the

☆ Editor: Eman Abdullah AlOmar.

* Corresponding author.
E-mail address: farid.edrisi@lnu.se (F. Edrisi).

https://doi.org/10.1016/j.jss.2026.113021
Received 27 November 2025; Received in revised form 29 May 2026; Accepted 23 June 2026
Available online 29 June 2026

feedback loop oversees the behavior of the system and the environment in the future to detect the need for adaptation. Then, the system self-adapt before a possible deviation or disruption from adaptation goals occurs during the operation time (Hielscher et al., 2008; Weyns, 2020). This approach can provide high-quality adaptations to reduce the risks and uncertainties associated with reactive adaptations, e.g., lagging behind changes and favoring short-term improvements (Moreno et al., 2017), and enhance the system's quality and resilience. However, proactive adaptation introduces its own challenges and considerations, e.g., complexity, performance, miss-prediction (Ghahremani and Giese, 2021).

Problem Statement. Most proactive self-adaptation approaches have been developed for information systems such as e-commerce platforms and web applications (Moreno et al., 2017; Angelopoulos et al., 2018; Chen and Jiao, 2022; Moreno et al., 2015, 2016, 2018; Ghahremani and Giese, 2021), with only limited efforts targeting CPS (Klös et al., 2018; Ayala et al., 2021; Luo et al., 2022). However, directly reusing approaches from information systems in CPS is challenging – and often impractical – due to fundamental differences in assumptions, such as system behavior and the nature of run-time models. On the other hand, existing CPS-specific proactive approaches often struggle with modeling overhead/runtime feasibility and architectural quality-limited reusability and alignment within larger adaptive ecosystems.

Accordingly, we identify the following Capabilities (C) for proactive self-adaptation in nonlinear CPS:

$C1$. **Technical Capabilities:**

- $C1.1$. Nonlinear and hybrid dynamics support to handle the inherent complexity of CPS.
- $C1.2$. Runtime learning to support adaptive parameter tuning and model refinement during operation to compensate for incomplete design-time information and model-to-reality discrepancies.
- $C1.3$. Predictive environmental and system insights to enable genuinely proactive – rather than merely reactive – adaptation decisions.
- $C1.4$. Adaptive computational management to maintain the real-time performance by dynamically adjusting planning effort without extensive predefined models.

$C2$. **Architectural Capabilities:**

- $C2.1$. Cross-domain reusability to enable instantiation across different CPS domains through modular components.
- $C2.2$. Ecosystem alignment via established architectural patterns (e.g., MAPE-K) to enable decentralized control within larger adaptive ecosystems.

The gap in the state of the art can be summarized as the lack of reusable proactive self-adaptation architectural solutions tailored to real-time nonlinear CPS that address the aforementioned technical and architectural capabilities.

Contribution. To address the highlighted requirements ($C1$–$C2$) and the identified gap, this work extends our previous study (Edrisi et al., 2025) by enhancing *PANCS* (Proactive self-Adaptation for Nonlinear Cyber–Physical Systems) and applying it to scenarios that include real-world uncertainties (e.g., noise, model-to-reality discrepancy). *PANCS* is a reference architecture for engineering self-adaptation in complex, nonlinear CPS. It provides the systematic architectural framework that makes control-theoretic mechanisms reusable across nonlinear CPS domains, where previously no such architectural solution existed. In particular, *PANCS* realizes Adaptive Model Predictive Control (*AMPC*)[1] upon the MAPE-K loop, to enable proactive adaptation in dynamic environments and to maintain its reusability, as motivated in Edrisi et al. (2025).

This work makes the following key contributions:

- We extend *PANCS* with improved architectural decomposition and run time models to address technical capabilities ($C1$). Indeed, a hybrid approach, combining control-theoretic numerical optimization with run-time models, enables proactive control while satisfying runtime qualities.
- We instantiate *PANCS* and evaluate its functionality and qualities under running example and through distributed co-simulation across several experiments, covering varied scenarios (different environment types, sensor faults, actuator degradation, dynamic obstacles). Results demonstrate significant improvements over reactive baselines.
- We illustrate *PANCS*'s reusability through its architectural design, where components can be instantiated across different CPS domains via modular design with clear extension points ($C2.1$), and the architecture can be integrated within larger adaptive ecosystems via MAPE-K conformance ($C2.2$). These reusability demonstrations are architectural and conceptual, where they provide concrete instantiation specifications that a domain expert could follow, but do not constitute independent empirical implementations in those domains.

The remainder of this article is organized as follows. Section 2 introduces the running example. Section 3 reviews related work on proactive self-adaptation approaches. While Section 4 overviews the architecture of *PANCS*, Section 5 presents the concrete instantiation of *PANCS* through the running example. Section 6 demonstrates *PANCS* applicability through a simplified use case, whereas Section 8 provides a comprehensive evaluation of *PANCS*. Section 9 illustrates how PANCS supports operational alignment within larger adaptive ecosystems through established MAPE-K coordination patterns. Section 10 demonstrates cross-domain reusability by mapping *PANCS* architectural elements to other domains than the running example. Section 11 discusses broader insights, lessons learned, and some limitations of the approach. In Section 12, several threats to validity are discussed, and finally, Section 13 concludes the paper and outlines directions for future work.

2. Running example

The running example is a mobile robot operating within a workspace, including static and dynamic entities, to perform its allocated tasks. It captures key characteristics of real-world CPS, including highlighted capabilities ($C1$–$C2$). Fig. 1 shows a layout of the warehouse environment used in the running example, where the system under study is shown as a blue robot with *Robot* label. Static elements include storage racks, conveyors, and other fixed equipment. Dynamic elements comprise other mobile robots (*Rob1*, *Rob2*, *Rob3*, and *Rob4* labels; shown as red robots) and moving obstacles (circle-shaped black obstacles). The robot under study operates independently, without communication or coordination with other agents.

The mission of the running example can be described as a set of goals. At the root, the goal is *Accomplish Mission Objectives without any collision*, which captures the robot's overall purpose in the warehouse. This root goal is AND-refined into two main subgoals: *Reach Target Locations without collision* (G1) and *Perform Assigned Tasks* (G2).

G1: Reach Target Locations without collision. The robot must navigate to designated target positions (e.g., T1, T2, T3 in Fig. 1) without collision with other obstacles in the environment. Multiple target locations may exist, and their execution order is guided by runtime priorities (High, Medium, Low).

G2: Perform Assigned Tasks. Once a target location is reached, the robot executes the corresponding warehouse operation, such as delivery or pick-up.

[1] AMPC offers a computationally efficient method for dealing with nonlinear systems by successively linearizing the system nonlinear model around the operating point (Adetola et al., 2009).

Table 1
Comparison of proactive self-adaptation approaches. Symbols: ✓ = fully supported, △ = not fully supported/limited, × = not supported.

Related works	Moreno et al. (2018)	Angelopoulos et al. (2018)	Chen et al. (2024)	Ayala et al. (2021)	Klös et al. (2018)	Luo et al. (2022)	Pereira et al. (2023)	Gesser et al. (2018)	*PANCS*	
Capabilities									Edrisi et al. (2025)	This work
*C*1.1	×	×	△	×	△	△	✓	△	✓	✓
*C*1.2	×	×	✓	✓	×	✓	✓	×	△	✓
*C*1.3	✓	△	✓	△	△	✓	✓	✓	×	✓
*C*1.4	△	△	△	×	△	△	△	×	✓	✓
*C*2.1	×	△	△	△	△	△	×	×	✓	✓
*C*2.2	×	×	△	×	×	×	×	×	✓	✓

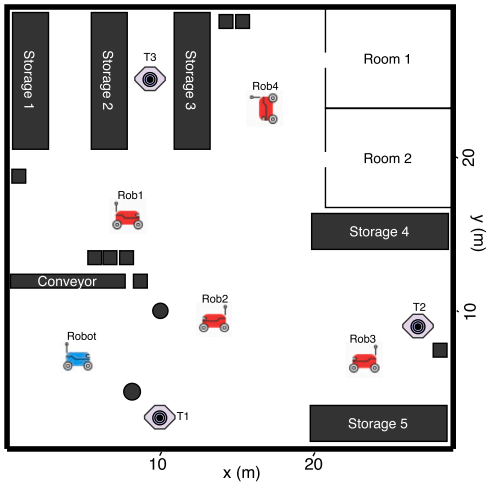


Fig. 1. Running example.

In addition to these functional goals, four non-functional soft-goals influence the mission execution:

SG1: Safety. The robot should remain operational and safe under external uncertainties, i.e., avoid near-collisions, when encountered with static and dynamic obstacles.

SG2: Fault Tolerance. The robot should remain operational in the presence of internal uncertainties, e.g., sensor or actuator faults.

SG3: Efficiency. The robot should minimize resource consumption, quantified as travel distance and time for accomplishing a mission.

SG4: Timeliness. All planning and task execution activities should respect real-time constraints, ensuring that decisions are made within the system's sampling interval.

Together, these soft-goals constrain how G1 and G2 are achieved, shaping the choice of feasible tasks and their execution at runtime.

3. Related work

Numerous approaches have been proposed for designing self-adaptive software (*SAS*), with an extensive body of literature now available (see Weyns (2020) for an overview). In this work, we focus on methods that involve or enable proactive adaptation in *CPS*, which are most closely related to our proposal. Recent studies in this domain emphasize the interplay between control theory, requirements/goal-driven engineering, and architecture-centric methodologies. To position *PANCS* relative to existing work, we evaluate approaches against capabilities derived from our problem statement and running example (*C*1–*C*2). Table 1 summarizes how existing proactive self-adaptation approaches address these capabilities.

Proactive Latency-aware Adaptation was proposed by Moreno et al. (2016), enhancing self-adaptive systems by integrating current and future adaptation requirements while considering adaptation latency. This approach initiates adaptations proactively, using Markov decision processes (MDPs) to optimize decisions over a finite look-ahead horizon. To address the complexity of constructing MDPs at runtime, the authors developed a method (Moreno et al., 2018) that minimizes the runtime overhead of constructing the MDP offline. During runtime, adaptation decisions are made by solving the MDP and integrating the stochastic environment model into the solution process. In another study, Angelopoulos et al. (2018) applied control theory to SAS to account for dynamic environmental changes in adaptation. They modeled the dynamic relationship between requirements and potential adaptations to develop a controller that optimizes requirement satisfaction relative to a cost function. The authors employed a linear time-invariant model to represent the system in their work.

While these solutions provide proactive approaches for performing self-adaptation, they are rarely valid for hybrid and nonlinear CPS. In particular, information systems are computational systems and can be represented using discrete-transition models, where system behavior evolves through discrete states and events. In contrast, engineering CPS requires integrating physical models, computational models, and network models (Barišić et al., 2022). The physical components are typically represented as continuous-time systems, governed by differential equations, while the computational and communication layers are modeled discretely (Barišić et al., 2022). Accurate representation of physical systems is challenging since most physical systems are inherently nonlinear (meaning that their outputs are not directly proportional to their inputs) and their behavior can evolve over time due to factors such as wear, aging, or environmental variation. This distinction is crucial, since proactive adaptation approaches rely on predictive models, and any simplification like treating nonlinear dynamics as linear introduces modeling inaccuracies. Such inaccuracies can lead to suboptimal or even unsafe adaptation decisions.

Ayala et al. (2021) introduce *ProDSPL*, which applies dynamic software product lines (DSPL) to drive proactive adaptation in CPS. ProDSPL systematically derives adaptation alternatives from variability models, emphasizing software variability and feature reconfiguration. While powerful for software-intensive systems, this approach remains largely agnostic to continuous dynamics, nonlinearities, and physical constraints. Moreover, the execution time averages are in the hundreds of milliseconds, with worst-case scenarios extending to several minutes. Due to the absence of any mechanism to improve runtime performance, this performance is inadequate for CPS that operate in rapidly changing dynamics.

Another line of research relevant to our work concerns runtime goal management. Klös et al. (2018) introduce a goal-aware CPS framework that continuously evaluates goal achievement and performs adaptations to re-establish satisfaction at runtime. Their approach emphasizes the quantitative modeling of goals and the evaluation of satisfaction levels against predicted environmental evolutions. Similarly, the *Captain* framework (Luo et al., 2022) employs feed-forward and feedback loops for runtime adaptation in autonomous systems. In the feed-forward loop, environmental data are monitored and analyzed to predict future evolutions, which are then used in a three-stage requirement satisfaction process (checking, analyzing, optimizing) to generate adaptation plans based on Model Predictive Control (*MPC*). The feedback loop monitors system characteristics (e.g., sensor degradation, component failures) and triggers the same pipeline when disturbances are detected. Although this framework supports partial satisfaction by minimizing

requirement violations, its reasoning remains centered on the explicit modeling and evaluation of goal satisfaction. While these approaches provide fine-grained reasoning about trade-offs among competing objectives, their practical adoption is often constrained by the overhead of defining and maintaining quantitative goal models at runtime. In highly dynamic or cluttered environments, this complexity can increase decision latency and reduce responsiveness. For instance, to preserve real-time feasibility, *Captain* employs coarse settings (e.g., 2-second sampling times and short prediction horizons), which limit its ability to work in fast-changing contexts.

Building on their earlier environment-aware MPC approach (Chen and Jiao, 2022), which augments the nominal system model with predicted environmental changes via a dynamic Bayesian model built from historical data, Chen et al. later proposed a two-layer control framework (Chen et al., 2024). This framework introduces a contextual goal model (CGM) to explicitly capture dynamic contexts, their influence on goals, and the quantification of softgoal satisfaction through quality functions. Thus, the new framework combines an MPC Control Layer, which computes optimal control parameters for proactive adaptation under varying contexts, with a Meta Control Layer that updates the CGM at runtime and fine-tunes MPC objectives and constraints. While this architecture separates goal reasoning from control optimization, its reliance on the historical data-based Bayesian model and a CGM brings run-time overhead and still requires explicit modeling of contextual factors, control parameters, and goal satisfaction functions, which is costly to design and maintain in highly dynamic CPS.

In the control-theory community, a closely related contribution is the work of Pereira et al. (2023), which applies reference-aware AMPC to design a path-tracking controller for autonomous vehicles. Regarding uncertainty handling, alternative approaches such as Robust MPC (RMPC), including min–max optimization and Tube MPC, explicitly handle uncertainties by minimizing worst-case disturbances (Gesser et al., 2018). *PANCS* differs from these controller-focused works in that it addresses AMPC as an architectural component within a reference architecture for self-adaptation, rather than as a domain-specific controller design problem. In addition, the RMPC variants may fall short in bounding the uncertainties offline, especially in nonlinear or time-varying systems.

4. *PANCS* architectural overview

This section presents the proposed reference architecture, *PANCS*, detailing its core components, their responsibilities, and the interactions that enable proactive self-adaptation in nonlinear CPS. We provide a high-level overview of the architecture, explaining how functionality is distributed across its major components and how these components collaborate within the MAPE-K loop.

Our approach to architecting CPS builds upon the conceptual model of self-adaptive systems (Weyns, 2020). Based on this model, we define four core elements: the *environment*, the *managed system*, the *managing system*, and the *CPS goals*. The *environment* encompasses all external factors beyond the CPS direct control, such as other mobile robots, static and dynamic obstacles, in the context of the running example. The *managed system* refers to the CPS under study, which is the robot itself in the running example. The *managed system* is considered in without any runtime adaptation logic; it executes actions but does not possess the capability to adjust its behavior autonomously in response to change. The *managing system* is responsible for overseeing and adapting the CPS behavior over time. It proactively handles uncertainties and dynamic conditions to guide the *managed system* toward fulfilling its goals. Finally, the *CPS goals* define the mission objectives that the CPS must achieve. These goals may evolve during execution and are used by the *managing system* to continuously adjust the CPS operational strategy.

The capability for proactive self-adaptation in CPS is enabled by a dedicated *managing system.* To systematically engineer this component, we employ *PANCS*, a reference architecture grounded in the MAPE-K loop and augmented with AMPC and runtime models to support proactive adaptation in dynamic environments. Leveraging AMPC enables *PANCS* to operate in a receding-horizon fashion, meaning it solves a constrained optimization problem over a finite future horizon at each time step, applies only the first control value, and then shifts the horizon forward, repeating the process with updated system states. The stability and convergence properties of AMPC under appropriate operating conditions have been studied extensively in the control theory literature (Mayne et al., 2000; Rahideh et al., 2008; Zheng et al., 2024). Accordingly, *PANCS* is designed to inherit these guarantees where the underlying modeling and operating assumptions are reasonably satisfied, while recognizing that performance may degrade in the presence of strong nonlinearities or significant model mismatch. The design of *PANCS* upon the MAPE–K architecture is intended to enable future reuse and ensure compatibility with larger adaptive ecosystems. This architectural combination allows the managing system to operate at both the system and controller levels. At the system level, the managing system serves as a high-level decision-maker, guiding the CPS toward preserving its desired qualities by constraining the set of feasible system states. At the controller level, it operates as a low-level manager, ensuring that the selected actions not only keep the system within these feasible states but also govern how these actions are executed in terms of the controller's resources and imposed high-level constraints (Edrisi et al., 2023).

Fig. 2 illustrates the proposed *PANCS* reference architecture, with abstract components of each element according to MAPE-K.

Monitor component serves as the entry point for environmental and internal states awareness by continuously collecting and updating the runtime data. It interfaces with multiple sensors to capture the information required for informed adaptation and control. It should be noted that, depending on sensor capabilities and the criticality of the information they provide, the *Monitor* component operates with heterogeneous sampling rates. As shown in Fig. 2, the *Env. Monitoring* subcomponent acquires and preprocesses environmental data, and the *MgS. Monitoring* subcomponent processes raw data extracted from CPS.

Analyze component is responsible for interpreting both the internal state of the CPS and its operational context, providing the configuration basis for optimization in the *Plan*. It is organized into two major subcomponents: *Uncertainty Recognition* and *Constraint Assessment*, each of which is further decomposed to address specific classes of challenges through relevant mechanisms.

The main responsibility of *Uncertainty Recognition* is capturing deviations and uncertainties that affect adaptation. It includes: *(i) Internal uncertainty*, which focuses on the managed system's hardware and software components. Fault detection identifies the existence of anomalous behaviors, then fault isolation pinpoints the affected component, enabling targeted corrective action. (*ii) External uncertainty* addresses changes in the environment. *Env. Analysis* inspects the CPS current operating environment in real time, and *Env. Prediction* anticipates future behavior of the environment. *(iii) Goal uncertainty* captures runtime uncertainties that affect the feasibility or relevance of functional goals (G). Such uncertainty may arise from incomplete or ambiguous information, unexpected environmental changes, or unpredictable interactions that render previously well-defined goals difficult or impossible to achieve.

Assessing these recognized uncertainties and translating relevant ones to constraints that *Plan* should adhere to is done through *Constraints Assessment*. Therefore, this component is designed as a collection of specialized subcomponents to handle high-level (system-level) qualities of CPS as constraints. The responsibilities of *Constraint Assessment* are organized around the canonical categories of constraints in an AMPC formulation: (i) *input constraints* which bound the range and rate of change of inputs, (ii) *output constraints* which define the admissible range of system outputs, and (iii) *mixed input–output (inequality) constraints* which construct linearized constraints boundaries based on new information, that will be used later in Section 5.1.3. Each type of

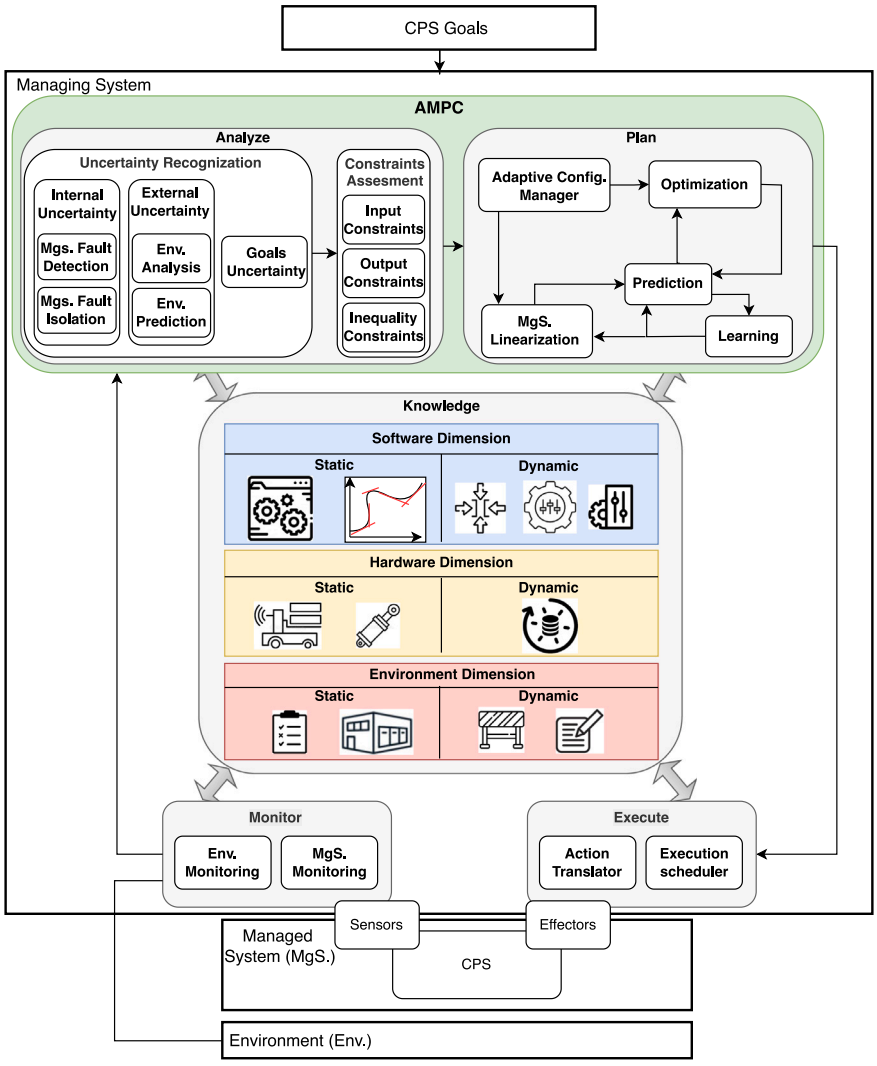


Fig. 2. Proposed *PANCS* architecture.

runtime uncertainty triggers updates in one or more of these categories, ensuring that the optimization problem remains consistent with the system's actual operational context.

Plan component serves as the low-level, controller-oriented manager of the CPS. While high-level decision making specifies uncertainties and constraints, the role of *Plan* is to determine *how* actions are operationalized within the current situations. To systematically realize the responsibilities of *Plan*, the component is architecturally decomposed into five coordinated subcomponents: *Adaptive Configuration Manager*, *MgS. Linearization*, *Optimization*, *Prediction* and *Learning*.

- The *Adaptive Config. Manager* establishes the planning setup by integrating inputs from *Analyze* and rule-based knowledge—a structured set of predefined rules and heuristics that encode expert insights and operational policies. It dynamically tunes control parameters (e.g., prediction or control horizons, constraint weights), assists in handling model-to-reality discrepancies, and may relax selected soft goals when operational challenges demand it.
- The *MgS. Linearization* component computes a time-varying linear model of the CPS nonlinear dynamics around the current operating point and discretizes it to be utilized in optimization problem. The purpose is to enable real-time optimization by converting nonlinear dynamics into quadratic program (QP) solvable in runtime, and handling time-varying systems through re-linearization at each timestep to adapt to changing operating points (unlike approaches with fixed models).
- The *Optimization* component formulates and solves a constrained optimization problem that aims to minimize deviation from the desired future states while minimizing control effort and adhering to constraints.
- To enable predictive accuracy in the presence of uncertainty, the *Prediction* component continuously forecasts future system outputs using a stochastic model. On the other hand, the *Learning* component improves predictions based on real-time incoming sensor data.

Execute component is responsible for enacting the control decisions produced by the *Plan* component. It applies the first control action from the optimized sequence to the managed system, ensuring that control actions are translated into actuator commands in a timely and reliable manner. This component is organized into two main subcomponents: *Action Translator* and *Execution Scheduler*. The *Action Translator* converts the abstract control outputs of the optimization stage into machine-understandable commands suitable for the actuators. The *Execution Scheduler* manages the timing and delivery of control actions. Since the optimization process may generate commands at a higher rate than the system's sampling interval, the scheduler buffers premature commands until their designated execution time. Conversely, if computational latency causes a delay, the scheduler reuses the most recent valid command.

Knowledge component acts as a structured shared information repository across the MAPE loop. To maximize efficiency and clarity, *Knowledge* is structured along three dimensions to mirror how engineers think about CPS: the world it operates in, the physical system itself, and the software logic driving it. Each dimension is further divided into static and dynamic categories to avoid unnecessary processing of unchanged data while ensuring that high-frequency updates stay accessible for real-time adaptation.

5. *PANCS* Instantiation

In this section, we realize the concrete instantiation of introduced architectural elements of *PANCS* through the running example described in Section 2, demonstrating how the abstract concepts are realized in practice.

In the context of the running example, the monitored data includes *LiDAR*, used for real-time detection of static and dynamic obstacles in the robot's surroundings, and *Inertial Measurement Unit (IMU)*,[2] aiding in the localization of the robot within the warehouse environment. The *Env. Monitoring* subcomponent acquires and preprocesses LiDAR data, including tasks such as handling missing values (NaN) and transforming measurements from the robot's body frame to the global frame. In contrast, the *MgS. Monitoring* subcomponent processes IMU data, extracting robot state information, i.e,. the pose from raw measurements.

Inside the *Analyze* component, Internal uncertainty focuses on covering *SG2* (Fault tolerance) for the running example. To do so, three main mechanisms for handling sensor, gradual actuator, and abrupt actuator faults are designed. *Sensor malfunctions* are detected and isolated by continuously comparing their outputs to known anomaly models, such as stuck values, out-of-range readings, or abrupt discontinuities. For actuator faults, because dedicated sensors for directly measuring them are not available in the running example, fault isolation relies on analyzing the residuals (difference between expected and actual values), computed in the *Learning* component (Section 5.1.4). The type and pattern of the residual signal provide evidence of which actuator is affected. For instance, we exploit the residual of the robot's heading angle (see Appendix C) as an indicator. A persistent residual exceeding a positive threshold suggests an abrupt fault in the right wheel, whereas a negative residual indicates an abrupt fault in the left wheel. For gradual fault detection and isolation, pure residual value might be insufficient. The slope of the moving average of the robot's hedging angle, in this case, is used, where a positive slope is a sign of right wheel gradual degradation, and a negative slope over time shows left wheel gradual fault.

External uncertainty is handled with collision avoidance mechanisms to cover *SG1* (Safety). Therefore, the used mechanisms in *Env. Analysis* includes obstacle identification, obstacle clustering, distance calculation to the robot, obstacles localization in the warehouse, and shape estimation. On the other hand, the dynamicity of identified obstacles and their future positions over the planning horizon (using linear regression) is performed in *Env. Prediction* component which allows proactive collision avoidance.

Goal uncertainty is flagged when, for instance, no feasible path exists to a designated target, obstacles appear in close proximity to the target that prevent safe task execution, or the robot's motion becomes overly cautious due to a densely populated environment. In each case, the underlying objective remains defined, but its realization is undermined by conditions that introduce doubt about its achievability within safety (SG1) and efficiency (SG3) constraints.

Internal uncertainties, such as actuator faults, are assessed in *Constraints Assessment* component through a fault assessment mechanism to estimate the degree of degradation from the magnitude of the residual in case of an abrupt fault and the slope of the residual's moving average in the occurrence of a gradual fault. The resulting scale factor is then used to adjust *Input Constraints* through the control input bounds to reflect reduced capability. *Output Constraints* define the minimum and maximum range for positions and heading angle. For example, if external uncertainties, human intervention, or commands from other agents limit the operational environment of the robot, it applies to output constraints. External uncertainties, such as infeasible paths, identifying new obstacles in the way of the robot, which need a combination of robot x-y positions are mapped through *Mixed Input–Output (Inequality) Constraints* into the optimization problem by updating relevant matrices and vectors.

Given the relative complexity of the *Plan* realization, its detailed description is provided in Section 5.1.

The *Action Translator* in *Execute* is realized through a dedicated velocity-to-torque conversion module, which performs the translation of velocity commands into torque values, specific to the type of our robot. The *Execution Scheduler* uses an internal clock to publish the most recent command at sample time T_s. The timing deviation (difference between received control action and T_s) is reported to the *Knowledge* component, which can be used later to inform the *Adaptive Config. Manager* to retune planning horizons or relaxation strategies, ensuring that execution remains both feasible and robust under varying runtime conditions.

The *Knowledge* component is structured as follows (see Fig. 2):

- *Environment Dimension.* Information describing the external context in which the CPS operates.
  - Static, e.g.,. warehouse map layout, conveyor locations, and safety clearance rules.
  - Dynamic, e.g.,. real-time obstacle positions, predicted obstacle position, etc.
- *Hardware Dimension.* Describes the physical platform under control.
  - Static, e.g.,. managed system physical characteristics, actuator limits, sensor placement, and sensor limits.
  - Dynamic, e.g.,. managed system's current states, data stream.
- *Software Dimension.* Captures the logic, models, and parameters guiding adaptation behavior.
  - Static, e.g.,. control architecture specifications, optimization algorithm, and model structure.
  - Dynamic, e.g.,. run-time constraints, learned model parameters, updated weights, and rule-based knowledge.

5.1. Plan instantiation

This section elaborates on the internal structure of the *Plan* component in *PANCS*, at the following subsections: the Adaptive Configuration Manager is discussed in Section 5.1.1, the Linearization process in Section 5.1.2, the Optimization procedure in Section 5.1.3, and the Prediction and Learning components in Section 5.1.4.

5.1.1. Adaptive confiq. Manager

The *Adaptive Config. Manager* is an adaptive reasoning component, responsible for dynamically configuring the parameters of the AMPC-based optimization infrastructure at runtime to ensure feasibility, safety, and performance under uncertainty. This component performs meta-level adaptation by computing how to configure the planning based on analysis outcomes (observed system conditions, predicted environmental changes, and goal-related information) and runtime feedback. To improve scalability, maintainability, and reuse across CPS domains, the Adaptive Config. Manager is structured around a clear separation between general (domain-independent) adaptation and domain-specific adaptation. This separation is a deliberate architectural design decision, where general rules are maintained centrally and

[2] An IMU consists of a 3-axis accelerometer and a 3-axis gyroscope.

apply uniformly across any *PANCS*-based CPS, while domain-specific rules are localized and independently evolvable by domain experts without modifying the core architecture. This structure addresses the challenge of rule evolution in long-running deployments, where new domain behaviors can be added to the domain-specific layer without impacting the general layer, and vice versa.

General adaptation. The general adaptation component manages configuration parameters that are common to AMPC-based CPS regardless of the application domain. These include the prediction horizon, control horizon, optimization weights, penalty variables, and updated model coefficients. Adaptations at this level are driven by domain-independent runtime indicators such as prediction uncertainty, constraint feasibility, and real-time budget utilization.

For example, if runtime monitoring reveals processing delays (the *Adaptive Config. Manager* closes the loop with the *Execute* and *Knowledge* components to assess timeliness), the *Adaptive Config. Manager* reduces the prediction or control horizons to maintain real-time feasibility. The relationship between horizon length and computation time is characterized through an offline calibration process during design time, yielding a simple empirical model that guides these runtime adjustments.

If the structure or reliability of the system model changes due to internal uncertainties, the general adaptation layer of the *Adaptive Config. Manager* addresses this change. For instance, when a sensor fault is detected by the *Analyze* component, the *Adaptive Config. Manager* increases the measurement noise covariance Q_m in the Kalman filter (see Section 5.1.4) to reduce reliance on the faulty sensor. Once the sensor returns to normal operation, Q_m is restored to its nominal value. Moreover, if an actuator fault is detected, the *Adaptive Config. Manager* updates the *MgS Linearization* by applying the corresponding scale factor computed by the *Analyze* component.

These adaptations directly affect the structure and solvability of the AMPC optimization problem and are applied uniformly across CPS domains.

Domain-specific adaptation. The domain-specific adaptation component refines configuration decisions by incorporating application semantics and expert knowledge that cannot be captured by domain-independent indicators alone. In the running example, this includes managing active goals, goal prioritization, and context-dependent interpretations of safety and performance trade-offs.

For instance, domain-specific rules determine how goals are activated or temporarily relaxed, how close the system may operate to constraint boundaries in specific contexts, and how conflicting objectives are resolved in exceptional situations. Such adaptations are necessarily informed by domain expertise (e.g., what constitutes acceptable proximity to obstacles in mobile robotics or acceptable deviations in other CPS domains) and are therefore localized to the domain-specific layer.

In the running example, when the *Goal Uncertainty module* detects that no feasible path exists to the designated target (for example, when a dynamic obstacle completely blocks all known routes and the environment topology prevents immediate alternative path discovery), the Adaptive Config. Manager implements a temporary goal substitution strategy. Specifically, it introduces an intermediate "waypoint" target positioned in a safe, forward direction (determined by identifying the largest obstacle-free sector within the robot's sensor range). Simultaneously, it reconfigures the optimization weights to reduce the target-reaching penalty while increasing the safety constraint penalty. This configuration biases the optimizer toward safe exploration rather than aggressive goal pursuit. The safety justification for this strategy is threefold. First, all safety constraints remain active, and collision-avoidance bounds are never relaxed during waypoint mode. The robot will not accept trajectories that violate minimum clearance distances, regardless of goal location. Second, forward motion enables the robot to acquire new sensor information, potentially revealing alternative paths as the environment evolves (e.g., the blocking obstacle moves, or previously occluded passages become visible). This is preferable to remaining stationary in a potentially unsafe location (e.g., obstructing other agents in a shared workspace) or declaring immediate mission failure. Third, the original target is reinstated as soon as the Goal Uncertainty module confirms feasibility (checked at each control step), ensuring the robot returns to nominal goal-directed behavior without manual intervention.

As another example, a distinct challenge arises when the designated target location is inherently in close proximity to static obstacles (common in warehouse environments where storage racks, conveyors, and structural elements create constrained workspaces). In such scenarios, strict enforcement of nominal obstacle avoidance constraints (e.g., maintaining 1 m clearance from all obstacles) can render the target physically unreachable, even though task completion requires the robot to approach closely (e.g., to 0.1 m for picking up an item adjacent to a shelf). To address this, when the robot is within a threshold distance of its target ($d(robot, target) < 1.0m$ in our implementation) and when obstacle proximity constraints would otherwise prevent target achievement, the Adaptive Config. Manager selectively reduces the penalty weight for obstacle avoidance constraints specifically for obstacles within the target vicinity. Concretely, for obstacles less than 1.0 m of the target, the inequality constraint penalty ρ_ϵ for that obstacle is reduced. Critically, this is a penalty relaxation, not a constraint removal, since the optimizer is allowed to accept small clearance reductions by paying a lower cost, but hard collision bounds ($d > 0.1m$) remain enforced. This strategy is only active in the final approach phase and only for obstacles that are spatially associated with the target location. For dynamic obstacles or obstacles away from the goal region, nominal safety penalties remain in effect. The intent is to enable task completion in constrained spaces while maintaining a defense-in-depth safety posture: hard bounds prevent collisions, reduced penalties enable goal achievement with controlled risk acceptance, and the spatial restriction ensures the relaxation applies only where operationally necessary.

In rare cases, adaptation decisions proposed by the general and domain-specific layers may conflict. In such situations, *PANCS* resolves conflicts by giving priority to the domain-specific adaptation, as it encodes context-aware, expert-defined semantics that reflect the operational intent of the system. This is analogous to traffic control, where traffic lights provide general, domain-independent regulation, but a police officer on-site has authority to override them when exceptional conditions arise. Similarly, in *PANCS*, the general adaptation layer establishes safe and feasible bounds, while the domain-specific layer may override general preferences within those bounds to ensure appropriate behavior in context-critical situations.

5.1.2. Linearization

The state of a dynamic system is defined by a minimal collection of variables, known as state variables, which are sufficient to fully characterize the system and its behavior in response to any inputs (Rowell, 2002). For systems exhibiting nonlinear behavior, such as mobile autonomous systems, the relationships among states and between states and inputs are nonlinear.

Let the system state and control variables be expressed as a tuple:

$$s : \{s_1, s_2, \dots, s_n\}, \quad u : \{u_1, u_2, \dots, u_m\}$$

where each s_i represents a state variable and u_i is a control parameter. The evolution of the state over time could be defined by a set of nonlinear differential equations of the form (Iqbal, 2021):

$$\begin{aligned} \dot{s}(t) &= f(s(t), u(t)) + n_s(t) \quad n_s(t) \sim \mathcal{N}(0, Q_p) \\ o(t) &= h(s(t), u(t)) + n_o(t) \quad n_o(t) \sim \mathcal{N}(0, Q_m) \end{aligned} \tag{1}$$

where $s(t)$ is the state vector, $u(t)$ is the control input vector, $o(t)$ is the observed properties vector, $n(t)$ is a noise term capturing uncertainties or modeling imperfections with zero mean and covariance matrices Q_p, Q_m, $f(\cdot)$ is a nonlinear function representing the system dynamics, and $h(\cdot)$ represents the output dynamic.[3]

[3] For more details on how this equation can be built for the robot in the running example, we refer readers to Eq. (C.1) in Appendix C.

At the core of *AMPC* is the idea of making the complex problem of nonlinear dynamics tractable in real time. Instead of solving the nonlinear equations directly, which is often computationally prohibitive, the system approximates its dynamics locally at each control step. In practice, this means that around the current operating state of the CPS, the nonlinear model is replaced by a linearized version that is valid for a short horizon. More frequent successive linearization improves approximation quality by limiting error accumulation. Appendix A provides details of the impact of linearization frequency on estimation quality and computation overhead. The resulting linear model enables accurate short-horizon prediction of system behavior while reformulating the control problem as a quadratic program (QP), which can be solved efficiently in real time for embedded or fast-sampling applications (Norby et al., 2024; Wang et al., 2024). Also, the final linearized model should be discrete because the optimization problem in *AMPC* is solved in a sampled-data setting, where predictions and control actions are computed at discrete intervals. Briefly, the aim is to reach from Eq. (1) to a discrete, and time-varying linear approximation of Eq. (1) as follows:

$$\begin{aligned} s_d(k+1) &= A_d^k s_d(k) + B_d^k u_d(k) + n_s(k) \quad n_s(k) \sim \mathcal{N}(0, Q_{pd}) \\ o_d(k) &= C_d^k s_d(k) + D_d^k u_d(k) + n_o(k) \quad n_o(k) \sim \mathcal{N}(0, Q_{md}) \end{aligned} \tag{2}$$

where A_d^k and B_d^k are the discrete-time state and input matrices, C_d^k and D_d^k are the observation matrices (as needed), and $n_s(k)$, $n_o(k)$ are discrete-time noise vectors with mean zero and covariance matrices Q_{pd} and Q_{md}.

The *MgS. Linearization* subcomponent performs this approximation task in our proposed architecture. Analytic Jacobian-based linearization (using partial derivatives) is adopted for linearization (see Eq. (3) (Iqbal, 2021)), as it is reusable and provides a computationally lightweight yet accurate local approximation that can be updated online at each timestep (Iqbal, 2021). Accordingly, the linearized continuous matrices around (s^*, u^*) are:

$$\begin{aligned} A^* &= \left.\frac{\partial f(s,u)}{\partial s}\right|_{s^*,u^*}, B^* = \left.\frac{\partial f(s,u)}{\partial u}\right|_{s^*,u^*}, \\ C^* &= \left.\frac{\partial h(s,u)}{\partial s}\right|_{s^*,u^*}, D^* = \left.\frac{\partial h(s,u)}{\partial u}\right|_{s^*,u^*} \end{aligned} \tag{3}$$

Since in *AMPC* the system matrices are recomputed at each sampling instant, linearization is performed around the current operating point, i.e., $s^* = s^k$ and $u^* = u^k$, where k denotes the current discrete time step. From the linearized continuous-time matrices in Eq. (3), we can discretize it with T_s sample time into discrete-time steps k using the Zero-Order-Hold approach Eq. (4).[4] It should be noted that if the system in Eq. (1) is directly given in discrete-time nonlinear form, Jacobian linearization can be applied around the operating point (s^k, u^k) to obtain the corresponding discrete-time matrices. In this case, the discretization step is unnecessary, and the resulting matrices in Eq. (4) reduce to $A_d^k = A^k$ and $B_d^k = B^k$.

$$\begin{aligned} A_d^k &= e^{A^k T_s} \approx I + A^k T_s + \frac{(A^k)^2 T_s^2}{2!} + \frac{(A^k)^3 T_s^3}{3!} + \dots \\ B_d^k &= \left(\int_0^{T_s} e^{A^k \tau} d\tau\right) B^k, \quad C_d^k = C^k, \quad D_d^k = D^k \end{aligned} \tag{4}$$

where I is the identity matrix of appropriate dimensions.

5.1.3. Optimization

As previously discussed, the *Plan* component functions as a low-level, control-oriented manager for the managed system. Its primary objective is to ensure that the CPS makes control decisions that guide it toward fulfilling its functional goals with as much satisfaction as possible of soft goals under different uncertainties. To achieve this, the *Optimization* subcomponent formulates a constrained optimization problem at each control step to determine an optimal sequence of control actions u_d (output of Plan component). This sequence aims to drive the set of the system's future predicted properties $\hat{o}_d$ (from Prediction component), forecasted using a linearized model (from MgS. Linearization component), toward their desired reference values o_g (from CPS Goals component), with minimum control effort while adhering to constraints (from Constraint Assessment in Analyze component) over a specified prediction horizon p (from Adaptive Config. Manager) (Angelopoulos et al., 2018). To do so, at each control step, an optimal sequence of control actions of length $c \le p$, called the control horizon (from Adaptive Config. Manager), is computed by minimizing a cost function while satisfying given constraints (see Eq. (5)). Following the receding horizon principle, only the first control action of the optimized sequence is applied to the CPS at the current step. At the next step, the horizon is shifted forward, and the optimization is repeated with updated state information. This iterative process enables proactive, context-aware adaptation that remains responsive to dynamic operating conditions.

The objective function is defined in Eq. (5) and has three main terms for penalizing (i) the deviation of predicted properties from their desired references, (ii) large control actions, and (iii) violations of constraints through the slack variable, scaled by a tuning weight ρ_ϵ which indicates the magnitude of the applied penalty.

$$\begin{aligned} &\min_{u_d,\epsilon} \sum_{n=1}^{p} \|\hat{o_d}(k+n) - o_g(k+n)\|_2^Q + \|u_d(k+n-1)\|_2^R + \rho_\epsilon \epsilon^2 \\ &subject\ to: \\ &u_l \le u_d(k+i) \le u_u, \quad \Delta u_l \le \Delta u_d(k+i) \le \Delta u_u \\ &o_l \le \hat{o_d}(k+j) \le o_u \\ &E u_d(k+i) + F \hat{o_d}(k+j) \le G + \epsilon Z \\ &\forall i \in 0, 1, \dots, p-1, \quad \forall j \in 1, 2, \dots, p \end{aligned} \tag{5}$$

where Q and R are matrices with weights to express the preferences on minimizing the deviation of the different components in the goals or control effort vectors, respectively; $\hat{o}_d(k+n)$ is an estimation of the values that will be observed n time steps in the future; the subscripts $(.)_l$ and $(.)_u$ denote the lower and upper bounds allowed values of the variable $(.)$, respectively; and $\Delta(.)$ represents the change in the variable $(.)$. In addition, through the last constraint, it is possible to set limits that involve linear combinations of the control and output variables. E, F, G, and Z are constant matrices or vectors. All variables in constraints are computed in *Constraints Assessment* of *Analyze* component. When constraints are declared *soft*, the optimization problem introduces a nonnegative slack variable $\epsilon \ge 0$. The scalar ϵ represents the optimizer's chosen amount of relaxation, applied proportionally across constraints via the elements of Z. At each step, the optimizer computes the minimal-cost solution ϵ. If $\epsilon > 0$, one or more constraints are deliberately relaxed by amounts $Z\epsilon$ to preserve feasibility or reduce tracking/control effort at the expense of paying the penalty $\rho_\epsilon \epsilon^2$. Increasing ρ_ϵ makes the solver less willing to allow violations, thereby tightening constraint enforcement; conversely, reducing ρ_ϵ yields a more permissive controller that prioritizes feasibility over strict constraint satisfaction.

Optimization component executes once for each execution of the *Planner*. To solve the optimization problem, it requests from the *Prediction* component several values of the future properties that would be observed $\hat{o}_d(i)$ when a control effort $u_d(i)$ is applied.

5.1.4. Prediction and learning

The linearized model Eq. (2) discussed in Section 5.1.2, while effective for prediction, may become inaccurate during operation due to unmodeled dynamics, inaccuracies, and uncertainties. To mitigate these limitations, a learning mechanism is required to continuously refine

[4] For more details on deriving discrete, and time-varying linear approximation of running example model, utilizing Eqs. (3) and (4), we refer the reader to Eq. (C.2) in Appendix C.

the model using incoming measurements. In our running example, a *Kalman Filter* continuously updates state estimates and model predictions based on sensor measurements. This is classical statistical learning for minimizing prediction error through recursive Bayesian estimation. Given that there are stochastic noises in the dynamic behavior of the system and in the sensing of the current state ($n_s(k)$ and $n_o(k)$ in Eq. (2)), the state predictor must work with an estimation of the state, represented as $\hat{s_d}$.

The *Prediction* executes the Kalman filter operations that forecast the future outputs using the best available estimation of the current state and the linearized model Eq. (2). This function can be requested several times by the *Optimization* component. For instance, it uses its best estimation of the system state at time $k-1$ (represented as $\hat{s_d}(k-1|k-1)$) and the control effort that is applied at time k to calculate the estimation of the next system state (represented as $\hat{s_d}(k|k-1)$). This function also produces the estimation of the future observations $\hat{o_d}$ using the system state estimation at that time.

The *Learning* function executes once for each execution of the *Planner*. It executes the Kalman filter operations that update the estimation of the current state when the monitoring brings the state observations at time k. Thus, it updates its current state estimation that used only past information (i.e., $\hat{s_d}(k|k-1)$) to an improved estimation that also uses the current observed values (represented as $\hat{s_d}(k|k)$).

Eq. (6) mathematically formulates how the *Kalman filter* equations are used to calculate the state estimations, addressing the stochastic uncertainty and separating its equations into prediction and learning functionalities.

$$
\begin{aligned}
&\hat{s}_d(k|k-i) = A_d^k \hat{s_d}(k-1|k-i) + B_d^k u_d(k) \quad (State\ Prediction)\\
&P(k|k-1) = A_d^k P(k-1|k-1)(A_d^k)^T + Q_p \quad (Covariance\ Prediction)\\
&\hat{o}(k|k-i) = C_d^k \hat{s_d}(k|k-1) \quad (Observation\ Prediction)\\
&K(k) = P(k|k-1)(C_d^k)^T \left(C_d^k P(k|k-1)(C_d^k)^T + Q_m\right)^{-1} \quad (Kalman\ Gain)\\
&r(k) = \left(o_d(k) - \hat{o}_d(k|k-1)\right) \quad (residual)\\
&\hat{s}_d(k|k) = \hat{s_d}(k|k-1) + K(k)\, r(k) \quad (State\ Learning)\\
&P(k|k) = (I - K(k)C_d^k)P(k|k-1) \quad (Covariance\ Learning)
\end{aligned}
\tag{6}
$$

5.1.5. Components interaction

Fig. 3 illustrates how the internal components of Plan connect and exchange information. In this figure, solid arrows (→) represent data flow between components. The arrowhead indicates the direction of information transfer. For example, the arrow from *Adaptive Config. Manager* to *Optimization* indicates that configuration parameters (horizon lengths, weight matrices, constraint bounds) flow from the manager to the optimizer. Reading Fig. 3 from left to right shows the temporal progression within a single planning cycle. The cycle begins with the *Adaptive Config. Manager*, receiving analysis results from the Analyze component (not shown in Fig. 3; see Fig. 2 for full context) and consulting rule-based knowledge to determine appropriate planning configurations. Configuration parameters flow to both the *MgS. Linearization* component (indicating parameter updates like fault-adjusted scale factors) and the *Optimization* component (indicating horizon lengths, weights, and constraint bounds). *MgS. Linearization* computes the time-varying linear approximation (matrices A_d^k, B_d^k, C_d^k from Eq. (4)) and provides these to both *Prediction* and *Learning* components, which use the linearized model for state forecasting and Kalman filter updates, respectively. The *Optimization* component formulates the constrained optimization problem (Eq. (5)) using the tuned parameters, linearized model, and constraint specifications. During solving, it repeatedly queries the Prediction component for future state estimates $\hat{s}_d(k+n|k)$ at candidate control sequences. The *Learning* component processes the most recent sensor measurements (obtained from Monitor, not shown in Fig. 3) using Kalman filter equations (Eq. (6)) to update state estimates and model parameters. These updated estimates improve the accuracy of subsequent predictions. Once optimization converges, the resulting control sequence $u_d(k\ :\ k+p-1)$ flows

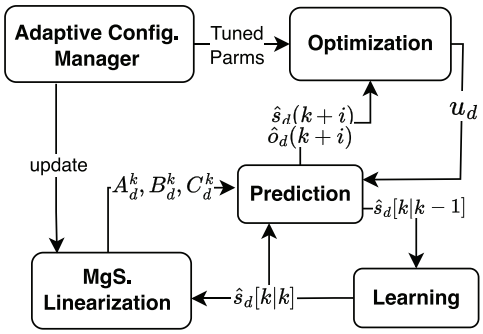


Fig. 3. Internal structure of the *Plan* phase in *PANCS*.

to the *Execute* component (not shown in Fig. 3; see Fig. 2), where only the first action $u_d(k)$ is applied to the managed system. Timing information and computational performance metrics are reported back to the Knowledge component, enabling the Adaptive Config. Manager to adjust parameters in the next cycle (closing the adaptation loop).

6. PANCS applicability

Before conducting the full evaluation of *PANCS*, its applicability is first examined using a *simplified use case*. The objective of this step is to isolate and validate the core technical capabilities of *PANCS* while avoiding additional integration and deployment complexities. This section demonstrates the applicability of *PANCS* under scenarios with varying levels of uncertainty and compares its performance with alternative approaches. In this preliminary use case, all system components, including the environment, the managed system, and the managing system, are simulated within a single platform, namely MATLAB/Simulink.[5] The insights gained from this simplified validation motivate the more realistic distributed co-simulation evaluation presented in the next section.

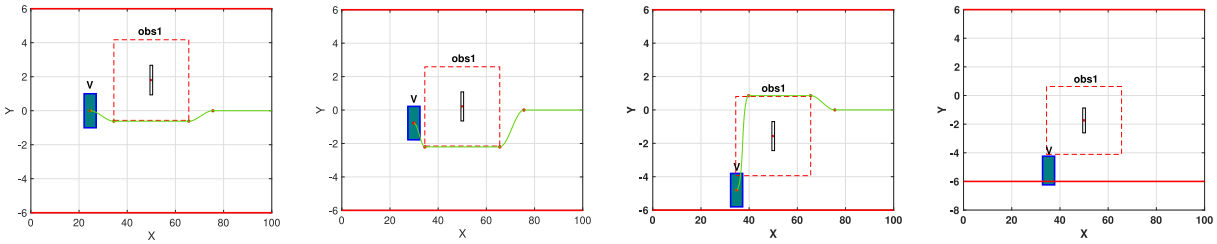
In this use case, we consider a ground vehicle, called V, modeled with nonlinear dynamics derived from a car-like vehicle model (see Appendix B for modeling details). For simplicity, the task here is to reach a designated destination (instead of a sequence of destinations in the running example) while navigating through an environment populated by both static and dynamic elements, including other autonomous robots and obstacles.

During operation, V may encounter two types of uncertainty: *environmental*, including obstacle placements, and movements; and *internal*, related to the state of V. Indeed, V has no prior knowledge of the existing obstacles and their trajectories or velocities. In complex environments, obstacles may appear or disappear unexpectedly. In addition, the V actuators and sensors may fail, while V has no prior information about them.

We examine V under two self-adaptation strategies: a proactive approach powered by *PANCS* and a baseline reactive approach. We also implemented a standard proactive MPC method to compare with *PANCS*. However, consistent with expectations, MPC failed to operate reliably under the tested scenarios. In detail, MPC relies on an accurate, predefined mathematical model of the system to predict its future behavior. However, in time-varying or nonlinear systems, the real system dynamics change over time, leading to model mismatch and making it difficult for the fixed model to remain accurate. This causes prediction errors and consequently poor quality or even instability of the system. Indeed, this limitation highlights the need for adaptive strategies that can cope with dynamic and uncertain conditions in real time.

Scenario Descriptions. Two progressively challenging scenarios that introduce both environmental and internal uncertainties are introduced:

[5] https://mathworks.com



(a) The first detection moment (b) The second detection moment (c) The third detection moment (d) Hitting the lower boundary.

Fig. 4. Failing the reactive method in completing the robot *V* task in scenario 1-B.

- *Scenario 1:* In this scenario, the environment contains five dynamic elements, three of which are in motion. It tests the vehicle's ability to safely navigate in the presence of moving obstacles and robots.
- *Scenario 2:* This scenario extends the previous one by introducing internal uncertainty—specifically, a fault in the steering. In addition to navigating a dynamic environment, *V* must adapt to reduced actuation reliability during execution.

Results. For each of the previously described scenarios, we experimented 24 instances with randomized values of initial positions and obstacle trajectories. We obtained that in *Scenario 1*, *V* reached $Dest_V$ in *20* out the *24* experiments (*83.33*%) when using *PANCS* and in *14* (*58.33*%) when using the reactive method. While *PANCS* is not able to succeed in all instances of the *simplified use case*, the proactive adaptation implemented by *PANCS* succeeds in more cases than the reactive one when facing a very dynamic environment. Moreover, we did not observe any instance in those *24* experiments where *PANCS* failed and the reactive method succeeded, indicating that *PANCS* provided an absolute improvement in the task accomplishment.

To provide a visual hint, we show in the following the details of two experiment instances, one when both methods succeed (*Scenario 1-A*) and one when the reactive fails and *PANCS* succeeds (*Scenario 1-B*). Therefore, we provide videos for the *V* navigation in *Scenario 1-A* when using the reactive [6] and *PANCS* [7] methods. *PANCS* reaches smoother trajectories since it anticipates future status, while the reactive method immediately adjusts to newly detected obstacles without considering the future consequences.

In the videos for *Scenario 1-B*, we observe that while the *PANCS* approach[8] successfully navigates among obstacles, the reactive method [9] fails at the first obstacle. Some important instances in the story of the failure of the reactive method are shown in Fig. 4. *V* notices the existence of dynamic *obs1* when its center is positioned at coordinates $(24, 0)$ (moment shown in Fig. 4(a)). The robot attempts to pass on the right, following the solid green line, because the *Analyze* notices that the path that avoids $obs1$ on the right side is shorter. The next time *V* senses *obs1*, *V* is located at $(29.94, -0.77)$ and *obs1* has moved in the same direction as *V* is trying to avoid it (Fig. 4(b)). The path to avoid $obs1$ on the right is still shorter. After *300* milliseconds, *V* is at coordinates $(34.28, -4.8)$ (Fig. 4(c)) and *obs1* movement continued in the same direction as *V*. At that moment, the *Analyze* notices that only avoiding *obs1* from the left remains viable, as the green line in the figure illustrates. However, *V* cannot adjust in time due to its maximum steering capabilities and hits the lower boundary (Fig. 4(d)).

For *scenario 2*, we introduced internal uncertainty in *V* operation by implementing a faulty steering that only applies *40*% of the steering command (e.g., if steering command is set to $\frac{\pi}{18}$ rad/second, only $\frac{\pi}{45}$ rad/second is actually applied). In this scenario, we obtained that *V* reached V_{dest} in *16* out of the *24* experiments (*66.66*%) when using *PANCS* and only in *7* (*29.2*%) when using the reactive method. With these data, it appears that internal uncertainty affects the accomplishment of *V* task when implementing *PANCS* since it succeeds fewer times than in *scenario 1*, while two positive findings arise when compared with the reactive method. First, *PANCS* succeeds in more cases than the reactive implementation, and second, the success rate of the reactive method is halved when adding internal uncertainty, while the success rate of *PANCS* only decreased by *20*%.

For this scenario, we illustrate the behavior of one of the experiment instances by providing videos when *V* implements the reactive [10] and *PANCS*.[11] In the case of the reactive method, *V* collides with dynamic *rob5*. Figs. 5(a)–(c) illustrate important states of the story of *V* failure in the recorded instance. Due to the unhandled actuation fault, the reactive method shows sharp turns or tight maneuvers to achieve the destination by using the maximum allowed steering. In turn, *V* succeeds when it implements *PANCS*. Its *Learning* component enables handling actuation fault to some degree since it automatically updates its internal model in real-time to account for discrepancies between *V* measured behavior and its expected behavior. Consequently, *PANCS* is able to compensate for the steering fault to some degree. Figs. 5(d)–(f) illustrate how *V* implementing *PANCS* avoids collision with *rob5* in the recorded instance.

It is important to note again that a simplified version of *PANCS* was implemented for this use case. Specifically, the *Analyze* component excludes the implementation of *Environment Prediction* and *Goals Uncertainty*. Furthermore, the observed improvement over the reactive method in the second scenario can be primarily attributed to *PANCS*'s learning capability embedded within the Kalman filter (the *Adaptive Config. Manager* does not directly modify the system model).

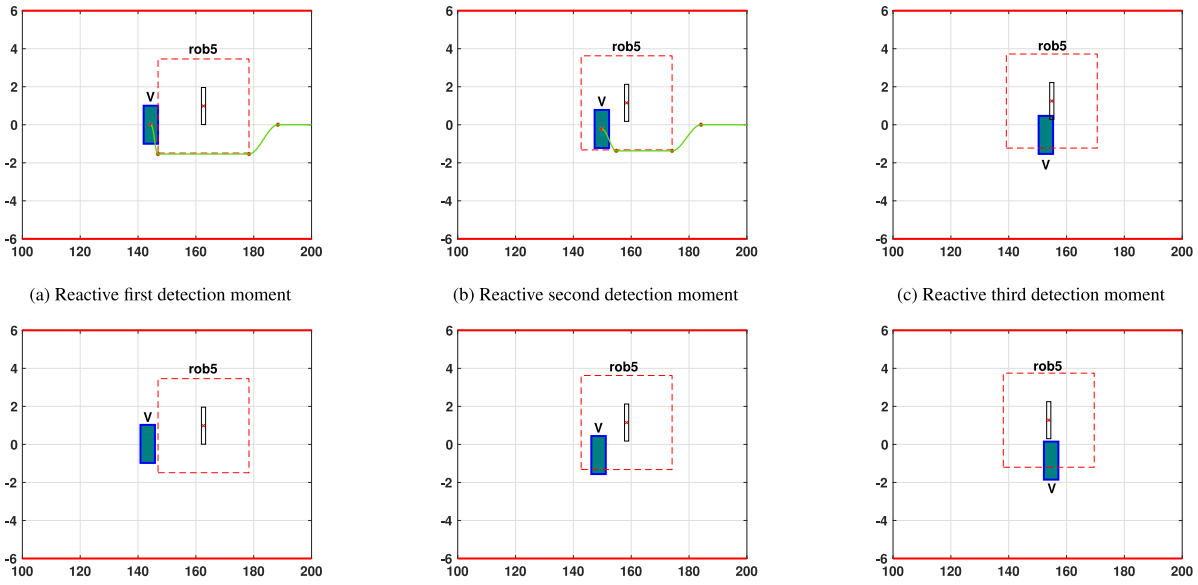
7. Evaluation design

The architecture of *PANCS*, presented in Section 4, was designed to address specific technical capabilities (*C1.1–C1.4*) and architectural capabilities (*C2.1–C2.2*) identified in Section 1. These capabilities were concretely instantiated, in Section 5, through the running example. Furthermore, the applicability of *PANCS* was examined through a simplified use case in Section 6. To assess whether the proposed architecture and its instantiation effectively realize the claimed capabilities, we formulate explicit evaluation Research Questions (RQs) that connect each capability to measurable outcomes or architectural evidence as follows:

RQ1: *Does proactive adaptation, enabled by PANCS, improve mission completion rates, safety, and efficiency while maintaining real-time feasibility, in the running example, compared to baseline self-adaptation approaches?*

This question directly addresses technical capabilities (C1.1–C1.4). We hypothesize that *PANCS*'s ability to anticipate future system states

[6] Clickable video link to Scenario 1-A - Reactive
[7] Clickable video link to Scenario 1-A - *PANCS*
[8] Clickable video link to Scenario 1-B - *PANCS*
[9] Clickable video link to Scenario 1-B - Reactive
[10] Clickable video link to scenario 2 - Reactive
[11] Clickable video link to scenario 2 - *PANCS*



(a) Reactive first detection moment (b) Reactive second detection moment (c) Reactive third detection moment

(d) *PANCS* first detection moment (e) *PANCS* second detection moment (f) *PANCS* third detection moment

Fig. 5. Encountering with *rob5* in scenario 2 using reactive (upper) and *PANCS* (lower).

and environmental changes through runtime learning and predictive optimization over an adjustable planning horizon, for balancing the deliberate use of computational budget for proactive planning without compromising responsiveness or incurring deadline violations, will result in improved qualities. The evaluation of this hypothesis is presented in Section 8.

RQ2: *What is the impact of PANCS components on execution overhead and quality improvements?*

This question evaluates the contribution of individual *PANCS* components and the execution overhead they introduce in addressing technical capabilities (C1.1–C1.4). Specifically, we analyze the execution-time overhead of each component on a per-sample basis and conduct an ablation study to quantify their respective effects on mission completion rate, safety, and efficiency, while ensuring real-time feasibility. This question is answered in Section 8.

RQ3: *How can PANCS-enabled CPS coordinate with other adaptive agents within larger multi-agent adaptive ecosystems?*

This question evaluates architectural capability C2.2 (Ecosystem alignment via established patterns). The concern is whether *PANCS*'s adherence to the MAPE-K loop facilitates operational alignment with other adaptive agents through standardized coordination mechanisms. This question is answered through qualitative architectural analysis in Section 9.

RQ4: *To what extent can the PANCS architectural components be systematically instantiated across heterogeneous CPS domains?*

This question evaluates architectural capability C2.1 (Cross-domain reusability through modular design), where the claim is that *PANCS* provides a reusable reference architecture, not merely a domain-specific solution. This requires demonstrating that the same architectural structure applies across diverse domains without requiring significant redesign. This question is answered through qualitative architectural analysis in Section 10.

8. PANCS evaluation

For evaluating the approach out of the development environment and mimicking the real-world application, in this section, we follow a distributed co-simulation paradigm designed for real-time control and adaptation of nonlinear CPS to implement the running example, described in Section 2. Accordingly, the architecture adopts a client–server pattern that integrates the managed system with the managing system. These components can be deployed on separate computational nodes, potentially running on heterogeneous platforms, and exchange data through a time-synchronized network interface.

8.1. Experimental setup

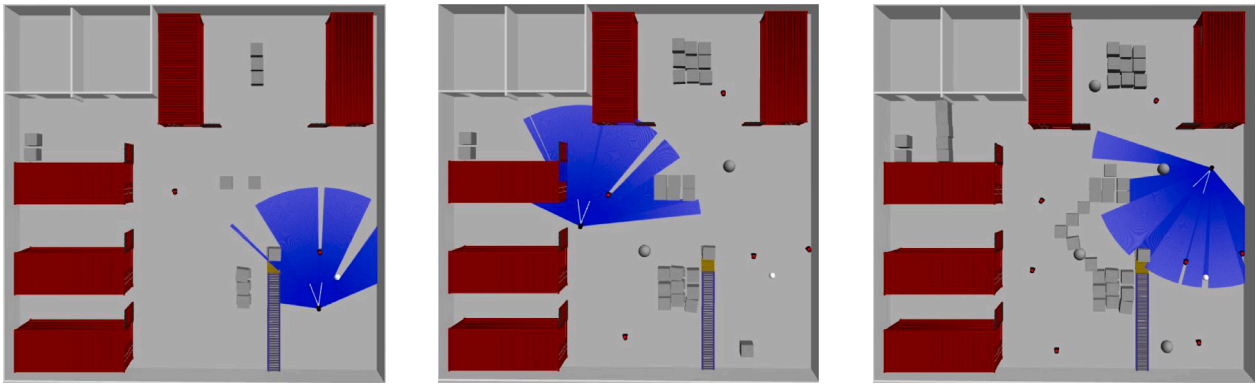
In our case, we integrate *Gazebo*[12] 3D robotics simulator running on a Linux virtual machine, and *MATLAB/Simulink*, performing runtime adaptation, which runs on a Windows 10 host. To bridge two systems, the Gazebo/MATLAB co-simulation plugin serves as middleware. It facilitates synchronized data exchange at a fixed sample rate to ensure that both systems remain aligned. The simulation progresses in lockstep with the control engine, enforced by a synchronization mechanism (e.g., pacer or time-stepping protocol).

To model the robot in the running example, we use a differential drive robot as the managed system (the model can be found in Appendix C).

Fig. 6 illustrates three representative warehouse test environments – Easy (E), Medium (M), and Hard (H) – used in our experiments. These configurations reflect varying complexity levels of external uncertainty. To make it clear, we adopt two intuitive and reproducible indicators to provide a definition of *complexity* for *external uncertainty*. The first is static obstacle density, defined as the ratio of occupied area to total area of the warehouse: $\rho = \frac{\text{occupied area}}{\text{total area}}$. The second is dynamic congestion, measured as the average number of dynamic agents (other robots and circle shaped obstacles in Fig. 6) per Ω unit of free area: $\lambda = \frac{\#\text{ dynamic agents}}{\Omega\text{ of free area}}$ (in our example $\Omega = 100(\text{m}^2)$). Classification of environment is done by thresholding these two factors: Easy (Fig. 6(a)) corresponds to low density and congestion ($\rho < 0.20$ and $\lambda < 1$), Medium (Fig. 6(b)) covers moderate values, and Hard (Fig. 6(c)) denotes high clutter or high congestion ($\rho \geq 0.30$ and $\lambda \geq 2$).

The environments were constructed using Gazebo world files, which encode the static knowledge of both the environment and the managed

[12] https://gazebosim.org/home



(a) Easy Environment ($\rho = 0.18$ and $\lambda = 0.40$). (b) Medium Environment ($\rho = 0.23$ and $\lambda = 1.12$). (c) Hard Environment ($\rho = 0.31$ and $\lambda = 2.1$).

Fig. 6. Three different sample instances of experiments in Gazebo.

system. This includes the layout of the warehouse, walls, shelves, the initial positions and shapes of all obstacles and robots, along with their embedded sensors. In addition to external uncertainty, we also introduce *internal uncertainty*, represented by the presence of faults within the managed system. These faults fall into one of three categories: Sensor Fault (SF), Actuator Gradual Fault (AGF), and Actuator Sudden Fault (ASF), each capturing different modes of system degradation.

We compare *PANCS* with a reactive baseline across all experimental configurations. Consistent with our earlier findings in Section 6, MPC was unable to complete the defined tasks in the running example. However, these approaches are not directly suitable for our introduced capabilities in Section 1 due to their assumptions and computational requirements. In particular, both methods primarily rely on offline characterization of uncertainty bounds and invariant sets, which conflicts with our requirements for runtime learning and adaptive model refinement (**C1.2**), proactive adaptation based on evolving environmental insights (**C1.3**), and adaptive computational management under changing runtime conditions (**C1.4**). Min–max RMPC assumes that the uncertainty set $\mathcal{W}$ can be characterized offline at design time in order to optimize against worst-case disturbances (Gesser et al., 2018). In our setting, however, uncertainties emerge dynamically at runtime and cannot be fully specified a priori. These include: (i) heterogeneous sensor faults with unpredictable timing and severity (e.g., noise spikes, stuck-at faults, and dropouts), (ii) actuator degradations occurring during operation, and (iii) dynamic obstacles with unknown trajectories that require continuous online observation and prediction. Capturing all possible combinations of faults, severities, sensors, and actuators would require an impractically large uncertainty model and extensive offline controller synthesis. Moreover, worst-case optimization introduces substantial conservatism (Jetto and Orsini, 2020). In dense and dynamic environments such as our Hard configuration, assuming simultaneously adversarial disturbances for all obstacles would excessively restrict feasible trajectories, leading to very low mission completion rates. Similarly, Tube MPC relies on the offline computation of invariant tubes around a nominal trajectory, typically under assumptions of bounded disturbances and relatively stable system dynamics. For nonlinear CPSs, exact reachable tube computation is generally intractable, while practical approximations based on polytopes or ellipsoids introduce conservative constraint tightening that degrades control performance, particularly near navigation targets (Rakovic et al., 2012). More importantly, our setting involves runtime-varying dynamics caused by evolving faults and environmental changes. Under such conditions, the robust tubes must be recomputed online, significantly increasing computational overhead and undermining the main advantage of offline tube construction (Gonzalez et al., 2011).

8.2. Evaluation metrics

To assess the quality and effectiveness of the proposed adaptation approach and systematically answer the *RQ1* and *RQ2*, we employ a set of reliability, safety, efficiency, and feasibility measures, based on the context of the running example, as follows:

Mission Success Ratio (MSR) Let $G = \{g_1, g_2, \dots, g_{|G|}\}$ denote the set of active tasks (goals) in a given mission, and let $\delta(g_i) = 1$ if goal g_i is completed within its operational constraints, and $\delta(g_i) = 0$ otherwise. The *MSR* measures the proportion of active tasks (goals) successfully accomplished during an experimental run ($G_{\text{succ}} = \sum_{i=1}^{|G|} \delta(g_i)$). It should be noted that, in the case of hitting an obstacle during a given mission, or failure to accomplish the goal within the feasible time (defined as the maximum allowed time for completing a task), all remaining $\delta(g_i)$ values are set to zero. The *MSR* for a single run is defined as $\text{MSR} = \frac{G_{succ}}{|G|}$.

Safety Risk (SR). It is quantified through the detection of *near-collision* events. A near-collision is defined as a spatiotemporal condition in which the Euclidean distance between the robot's boundary and any obstacle boundary at sampling step k ($d(k)$) falls below a predefined safety threshold d_{safe}, but a collision does not occur. Formally, the number of near-collisions over T set of sampled time steps can be expressed as $N_{\text{nc}} = \sum_{k=1}^{|T|} \mathbf{1}(d(k) < d_{\text{safe}})$ where $\mathbf{1}(.)$ is the indicator function. The total exposure time is accordingly, $t_{\text{nc}} = T_s \cdot N_{\text{nc}}$ where T_s is the sample time. The *SR* is then the fraction of risky operation time relative to the total mission time as $SR = \frac{t_{\text{nc}}}{T}$.

Performance Efficiency (PE). Let $t_{\text{start}}(g_i)$ be the time when goal g_i becomes active and $t_{\text{reach}}(g_i)$ be the time when it is reached. The goal fulfillment time is defined as $T_{\text{goal}}(g_i) = t_{\text{reach}}(g_i) - t_{\text{start}}(g_i)$. The performance efficiency metric is defined as $P = \frac{\sum_{g_i \in G_{\text{succ}}} T_{\text{goal}}(g_i)}{|G_{\text{succ}}|}$, where lower values of P indicate more efficient task execution under dynamic conditions.

Mission Efficiency (ME). Mission efficiency evaluates the spatial optimality of the robot's path. Let $D(g_i)$ be the total traveled distance from $t_{\text{start}}(g_i)$ to $t_{\text{reach}}(g_i)$. The *ME* for an experimental run is the average traveled distance per successfully completed goal: $\text{ME} = \frac{\sum_{g_i \in G_{\text{succ}}} D(g_i)}{|G_{\text{succ}}|}$. Lower path lengths generally reflect better planning, obstacle avoidance, and adaptation strategies.

Real-Time Feasibility (RTF). The proposed adaptation approach must operate within strict real-time constraints to ensure responsiveness. Let $T_{\text{proc}}(k)$ denote the actual processing time required to compute the adaptation decision at sampling step k, and T_{sample} be the fixed sampling interval. The processing time utilization or Real-Time Feasibility (RTF) in a run is given by $RTF = \frac{mean(T_{\text{proc}})}{T_{\text{sample}}}$. Under this definition, lower RTF values indicate better real-time feasibility, as they reflect less time spent on computation relative to the available budget. A value below

Table 2
Paired comparison of Reactive vs. *PANCS* across all runs.

Metric	Reactive mean (SD)	PANCS mean (SD)	Effect size (Cohen's d)	p-value
MSR	0.608 (0.391)	0.911 (0.183)	0.992	<0.001
SR [%]	30.767 (16.671)	15.810 (12.021)	1.029	<0.001
PE [s]	25.094 (8.419)	25.162 (5.888)	0.010	0.588
ME [m]	14.545 (4.417)	13.165 (3.104)	0.383	0.0006
RTF [%]	58.893 (8.690)	82.493 (8.399)	2.489	<0.001

1.0 is required to meet strict real-time constraints, and smaller values imply greater computational headroom.

8.3. Results

This section will discuss the results of *PANCS* and the baseline approach. For each run, we recorded *MSR* (higher better), *SR* (lower better), *PE* (lower better), *ME* (lower better), and *RTF* (lower better) (The detailed results can be found in Table D.5 in Appendix D). Several thresholds are defined as part of the evaluation metrics. For *MSR*, we set a maximum mission time of 300 s (considering the warehouse dimensions) per goal to distinguish between achievable-but-slow completions and genuine failures. This generous time limit ensures that failures reflect fundamental inability rather than arbitrary time constraints. For *SR*, the nominal safety threshold d_{safe} is set to 1.0 m (approximately 4× the robot's largest dimension), defining the boundary below which proximity events are considered near-collisions. However, *PANCS* may temporarily relax this threshold down to 0.1 m (hard collision bound which is never violated) in constrained workspaces (e.g., when approaching goals located near obstacles) through domain-specific adaptation, as described in Section 5.1.1. For *RTF*, rather than minimizing computational cost at the expense of adaptation quality, we configured *PANCS* to utilize 80% of the available runtime budget (T_s = 60 ms). All these parameters are exposed as configurable design parameters in *PANCS* and can be tuned for different operational contexts without architectural modification.

Table 2 summarizes experiments' statistics (Reactive vs *PANCS*) where we report *mean* and *standard deviation (SD)* to summarize central tendency and variability across experimental runs. To evaluate the performance differences between two methods, we conducted paired statistical tests. First, we applied the Shapiro–Wilk test to check whether the differences between paired results followed a normal distribution. If the normality assumption held, we used a paired t-test, which is suitable for comparing means when data is normally distributed. If the data did not meet the normality criteria, we instead used the Wilcoxon signed-rank test—a non-parametric alternative that does not assume normality. For each comparison, we report the *p-value*, which indicates whether the observed difference is statistically significant. Additionally, we include the paired Cohen's d effect size to quantify the magnitude of the difference between the two methods, helping to interpret not just whether the difference exists, but how meaningful it is.

Across paired observations, *PANCS* demonstrated statistically significant improvements over the reactive baseline on multiple quality attributes. Specifically, the mean *MSR* increased from $\overline{\text{MSR}}_{\text{re}}$[13] = 0.608 (SD = 0.391, N = 48) to $\overline{\text{MSR}}_{PANCS}$ = 0.911 (SD = 0.183), indicating a substantial improvement in task completion. A paired Wilcoxon signed-rank test confirmed the significance of this increase ($p < 0.001$). In terms of safety (*SR*), the average value decreased from $\overline{\text{SR}}_{\text{re}}$ = 30.767 (SD = 16.671) to $\overline{\text{SR}}_{PANCS}$ = 15.810 (SD = 12.021), reflecting a mean reduction of 14.957 units. This reduction was statistically significant according to a paired t-test ($p < 0.001$), with a large effect size (Cohen's d = 1.029), which highlights not only statistical but also practical significance.

[13] $\overline{(.)}_{re}$ means average of (.) for reactive approach.

It should be highlighted that *PANCS* does not sacrifice efficiency to gain improved safety or success rate. Table 2 indicates that *PE* remained stable under *PANCS* compared to baseline, with mean values changing only slightly from 25.162 to 25.094 (Wilcoxon p = 0.588). This consistency suggests that *PANCS* maintains a balanced execution profile, effectively utilizing the available time for proactive adaptation without introducing unnecessary delays in operation or premature decisions. On the other hand, *ME* showed a statistically significant improvement under *PANCS*. The mean path length decreased from $\overline{\text{ME}}_{\text{re}}$ = 14.545 (SD = 4.417, N = 20) to $\overline{\text{ME}}_{PANCS}$ = 13.165 (SD = 3.104), with a Wilcoxon p = 0.0006 and a moderate effect size (Cohen's d = 0.383). This indicates that *PANCS* not only maintains efficient use of operation time but also supports more streamlined and effective motion planning. Results of *PE* and *ME* mean that temporal and spatial efficiency is preserved while achieving improvements in other dimensions.

Under *PANCS*, the mean *RTF* increased from $\overline{\text{RTF}}_{\text{re}}$ = 58.893% (SD = 8.690, N = 45) to $\overline{\text{RTF}}_{PANCS}$ = 82.493% (SD = 8.399), a statistically significant difference of 23.6 percentage points (paired t-test, $p < 0.001$, Cohen's d = 2.489). However, part of this increase actually reflects *PANCS* design for utilizing 80% of the available runtime budget for proactive adaptation. The observed average confirms that *PANCS* consistently operated near this target, ensuring that decisions were made with sufficient deliberation while still respecting real-time constraints. The results validate this design choice: zero deadline violations and the 23.6 percentage point increase in *RTF* enabled 49.8% and 48.6% relative improvement of *MSR* and *SR*, respectively. The computational investment yielded measurable quality returns.

To complement the univariate analysis, we performed a multi-objective evaluation. Fig. 7 provides a multi-dimensional view of how *PANCS* and the reactive baseline handle trade-offs among safety (*SR*), efficiency (*PE* and *ME*), and computational feasibility (*RTF*) under diverse and randomly generated experimental settings. Despite variations in environments, goal placements, and task configurations, clear patterns emerge.

Safety vs. Efficiency. In the SR–PE and SR–ME plots (top row), *PANCS* consistently clusters toward lower risk values while maintaining comparable or better efficiency. Specifically, for SR-PE (top-left), *PANCS* achieves lower safety risk without increasing mission time, forming a non-dominated set compared to the reactive baseline, which often sacrifices safety for speed. In SR-ME (top-right), *PANCS* delivers shorter and more efficient paths while reducing risk, indicating that proactive adaptation improves trajectory quality without compromising safety. These two patterns confirm that *PANCS*'s predictive, constraint-aware planning enables safer and more efficient behaviors than the reactive strategy, which frequently exhibits dominated trade-offs.

Safety, efficiency vs. Real-Time Feasibility. The SR–RTF (middle-left), PE-RTF (down-left), and ME-RTF (down-right) plots show that *PANCS* operates with higher RTF values, reflecting its deliberate use of the available runtime budget for proactive adaptation. While the reactive baseline leaves substantial computational headroom (lower RTF), this often coincides with higher safety risk and longer paths. *PANCS*'s configuration to utilize up to 80% is managed through horizon planning, which constrains lookahead depth to operate near its runtime target and sustain real-time operation while mitigating risk and efficiency through proactive self-adaptation.

Efficiency Trade-Offs. In the PE–ME (Middle-right) trade-off space, *PANCS* demonstrates a clear advantage by clustering toward lower

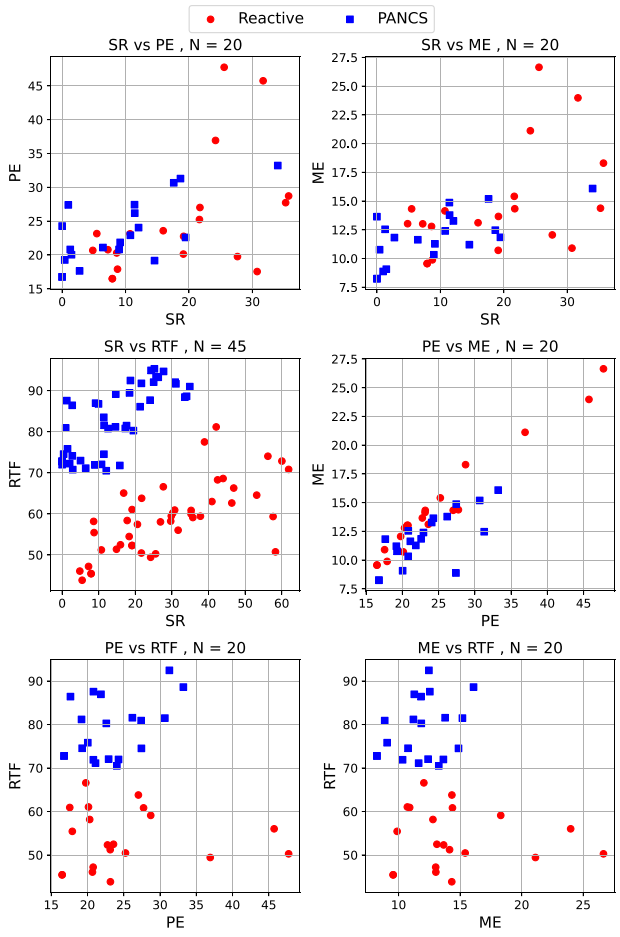


Fig. 7. Multi-dimensional view of *PANCS* and reactive baseline results.

path lengths (better motion efficiency) while maintaining execution times comparable to or better than the reactive baseline. This pattern indicates that proactive adaptation does not introduce timing penalties when optimizing trajectories. Instead, *PANCS* leverages its predictive planning to achieve streamlined paths without sacrificing responsiveness. In contrast, the reactive approach exhibits greater variability, often trading shorter execution times for longer, less efficient paths. These results suggest that *PANCS* provides a balanced improvement in both temporal and spatial efficiency, which is critical for applications where minimizing travel distance directly impacts energy consumption and task completion reliability.

We also computed aggregate scores (normalized metrics averaged per run) using equal weights (MSR, SR, PE, ME, RTF equally weighted). The mean aggregate score increased from $\mu_{\text{re}} = 0.5905$ to $\mu_{PANCS} = 0.6544$ over $N = 48$ paired runs (paired test $p = 0.0328$), indicating a statistically significant overall improvement in combined safety, efficiency, and real-time metrics.

These results provide strong affirmative evidence for answering *RQ*1. *PANCS* technical capabilities, derived from its architectural design, lead to enhanced functionality and operational quality in the running example. In particular, *PANCS*-enabled robot outperforms reactive approaches across key dimensions, including mission success ratio, safety, and efficiency, while preserving real-time feasibility.

To answer *RQ*2, we provide insight into the execution overhead of *PANCS* component and conduct an ablation study to realize the impact of each component on *Evaluation* metrics. Table 3 reports the execution time breakdown across MAPE-K components, averaged over 48 experimental runs with sampling Time, $T_s = 60$ ms. The measurements were obtained using the Simulink Profiler, which records task-level execution times based on high-resolution timing sources during simulation on a dedicated Windows 10 host running MATLAB/Simulink.

As expected, the Plan component dominates computational cost, consuming 44.4% of total processing time. Within Plan, the Optimization subcomponent (quadratic programming solver) accounts for 31.8% of total time, reflecting the inherent complexity of solving constrained optimization problems at each control step. The prediction and control horizon length directly influences this cost (in our experiments, $5 \leq p \leq 30$ and $1 \leq c \leq p$ steps depending on adaptive horizon tuning, with each additional step adding approximately $1 - 1.8$ ms to optimization time). Notably, the Kalman filter consumes only 2.2% of processing time despite executing at every control step. This efficiency validates the choice of Kalman filtering over more sophisticated but computationally expensive learning approaches (e.g., neural network inference, particle filters) for real-time CPS contexts. The Adaptive Config. Manager's low overhead (8.3%) demonstrates that rule-based adaptation logic can be executed efficiently through table lookups and conditional checks.

Monitor spends the second-largest portion of the computational budget. This reflects the integration with the Gazebo simulation environment: MgS. Monitoring (9.183 ms) includes ROS-MATLAB communication overhead for sensor data acquisition, coordinate frame transformations, and pose extraction from IMU measurements. While this overhead would be lower on physical hardware with direct sensor access, the distributed co-simulation architecture necessitates network communication. Env. Monitoring (4.035 ms) performs LiDAR point cloud processing and coordinate transformations to the global frame. The Analyze component's moderate cost (13.6%) reflects the deliberate design choice to avoid heavyweight reasoning. Uncertainty Recognition performs residual-based fault detection using lightweight statistical tests (threshold comparisons, moving average computation), lightweight environment prediction, and goal uncertainty detection, while Constraint Assessment primarily involves matrix lookups and simple arithmetic to update constraint bounds—avoiding expensive symbolic or search-based reasoning that would threaten real-time feasibility. The Execute component time spending shows that command publishing consumes notable time, primarily due to ROS message serialization and Gazebo service calls in the co-simulation testbed (Action translation consumes 8.4 ms). On physical hardware with direct actuator interfaces, this overhead would reduce significantly and is expected to drop to the low-millisecond range (sub-2 ms under typical real-time ROS 2 configurations (Park et al., 2020)).

The execution time distribution also reveals headroom for additional functionality within almost 20% unused budget, as planned. For instance, integrating perception algorithms (e.g., object classification, semantic mapping) or higher-level planning (e.g., multi-target task scheduling) could be considered. This supports *PANCS*'s suitability for integration into more complex CPS architectures where the managing system is one component among many. It is worth noting that the RTF target directly governs the budget available to the Plan component, and specifically to its Optimization component, since the remaining components (Monitor, Analyze, Execute) consume relatively fixed portions of the sampling interval regardless of horizon length (Table 3). A lower RTF target would reduce the optimization budget, forcing shorter prediction and control horizons and potentially degrading adaptation quality in complex environments. Conversely, a higher target would allow longer horizons and finer-grained optimization, but at the cost of a narrower safety margin against scheduling jitter or unexpected computational spikes. The 80% target represents a deliberate balance between adaptation quality and real-time robustness, validated empirically across 48 runs with zero deadline violations. Practitioners deploying PANCS on different hardware or under different co-resident process loads should treat this target as a tunable parameter and recalibrate it through offline profiling.

To assess individual component contributions and address *RQ*2, we conducted an ablation study where *PANCS* components were systematically disabled. Due to the computational expense of comprehensive

Table 3
Execution time breakdown by component (T_s = 60 ms).

Component	Mean time (ms)	% of T_s	% of Total
Monitor	13.234	22.1%	26.8%
Env. Monitoring	4.035	6.7%	8.2%
MgS. Monitoring	9.183	15.3%	18.6%
Analyze	6.741	11.2%	13.6%
Uncertainty Recognition	5.252	8.8%	10.6%
Constraint Assessment	0.136	0.2%	0.3%
Plan	21.941	36.6%	44.4%
Adaptive Config. Manager	4.106	6.8%	8.3%
MgS. Linearization	0.047	0.1%	0.1%
Optimization	15.726	26.2%	31.8%
Prediction	0.900	1.5%	1.8%
Learning	0.190	0.3%	0.4%
Execute	7.580	12.6%	15.2%
Action Translator	7.438	12.4%	15.0%
Execution Scheduler	0.110	0.2%	0.2%
Total Processing	**49.496**	**82.5%**	**100.0%**

Table 4
Ablation study: Component impact (Medium Environment with sensory fault).

Variant	MSR	SR (%)	PE (s)	ME (m)	RTF (%)
PANCS-Full	1.00	18.012	24.967	10.002	83.054
PANCS- without Env. prediction	0.667	–	–	–	–
PANCS- without Mgs.Linearization	0.333	–	–	–	–
PANCS- without Adaptive Config. Manager (General)	1.00	27.64	33.401	13.836	96.072
PANCS- without Adaptive Config. Manager (Domain-specific)	0.667	–	–	–	–
PANCS- without Kalman Filter (Learning)	0.00	–	–	–	–
Reactive	0.333	–	–	–	–

ablation across all 48 experimental configurations, we present results for a representative scenario: Medium-uncertainty environment with sensory fault (absence of robot y-axis position measurement for 12 s, from t = 7 s to t = 19 s) with three sequential goals. This scenario was selected because it balances challenge with tractability and represents typical operational conditions where both environmental and internal uncertainties co-occur. Table 4 presents the results. Note that metrics marked with $'-'$ indicate mission failure (collision or timeout), preventing meaningful measurement of safety, efficiency, or timing metrics for incomplete missions.

The removal of the Learning component resulted in complete mission failure (MSR = 0.00). This component performs state estimation correction using sensor measurements, and continuous model refinement to compensate for model-to-reality mismatch. Without it, the system operates in open-loop prediction mode, where the linearized model generates state predictions without measurement feedback. In nonlinear systems, even small initial errors or modeling inaccuracies accumulate rapidly. During the sensor fault period (t = 7-19 s), when y-axis measurements are unavailable, the system has no mechanism to detect or correct the growing state estimation error. The robot's internal belief about its position diverges significantly from reality, causing it to execute maneuvers based on incorrect state information, ultimately leading to collision. This result confirms that feedback-based learning is not merely an optimization but a fundamental requirement for reliable operation under uncertainty.

Removing successive linearization (MSR = 0.333) demonstrates the importance of adaptive model updates. To ablate this component, we performed linearization only once at initialization and reused the static linear model throughout operation. Since the system is nonlinear and the operating point changes continuously during navigation, the initial linearization becomes increasingly invalid over time. The Kalman filter compensates for this mismatch to some extent using measurement feedback, which explains why one goal was achieved. However, during the sensor fault period (t = 7-19 sec), the linear model no longer accurately represents the system dynamics at the current operating point, and the missing y-axis measurement cannot provide corrective feedback. This combination causes significant state estimation drift, leading to navigation failure and collision with obstacles.

Both the removal of Environment Prediction and Domain-Specific Adaptive Config. Manager resulted in MSR = 0.667 (two goals achieved, one failed), but through distinctly different failure mechanisms. Without Environment Prediction, the system cannot anticipate the future positions of dynamic obstacles (other robots). Instead of proactively planning collision-free trajectories that account for where obstacles will be, the optimization treats them as if they were stationary at their currently observed positions. In the test scenario, this led to a collision with an oncoming robot that the system failed to avoid because it planned a path that would have been safe if the obstacle had remained stationary, but was unsafe given its actual trajectory. In contrast, removing the Domain-Specific Adaptive Config. Manager preserved collision avoidance but caused a different failure: goal unreachability due to overly conservative safety constraints. Near the third goal location, static obstacles created a constrained workspace. Without domain-specific adaptation logic to selectively relax obstacle clearance penalties in the goal vicinity (while maintaining hard safety bounds), the optimizer determined no feasible solution existed that could simultaneously satisfy both strict clearance constraints and goal-reaching requirements. The mission timed out as the robot oscillated near the goal without being able to complete the final approach, despite having a physically feasible path.

Removing the General Adaptive Config. Manager component still allowed mission completion (MSR = 1.00), but with substantially degraded quality metrics: SR increased from 18.012% to 27.64% (+53%), PE increased from 24.967 sec to 33.401 sec (+34%), ME increased from 10.002 m to 13.836 m (+38%), and RTF increased from 83.054% to 96.072% (+16%). This component provides two critical adaptation mechanisms; (i) adjusting measurement noise covariance (Q_m) when sensor faults are detected to reduce reliance on degraded sensors, and (ii) dynamically managing prediction and control horizons to maintain

real-time feasibility. Without adaptive Q_m adjustment, the Kalman filter continued to trust the faulty y-axis sensor during t = 7-19s, leading to corrupted state estimates. The robot executed maneuvers based on partially incorrect state information, causing it to initially navigate away from the goal direction before correcting course once the sensor recovered. This detour explains the increased path length (ME), mission time (PE), and safety risk (SR) as the robot encountered more obstacles along its extended trajectory. Additionally, without adaptive horizon management, the system used fixed prediction horizon (p = 20) and control horizon (c = 10) throughout operation. This explains the RTF increase to 96.072% where the computational budget was nearly fully consumed by solving large optimization problems even in scenarios where shorter horizons would have sufficed. The fixed long horizons did enable the robot to eventually complete all goals (hence $MSR = 1.00$), but at the cost of computational efficiency and reduced responsiveness.

The ablation results reveal several important architectural properties of *PANCS*. There is a clear hierarchy of criticality: state estimation and model adaptation (Learning and Linearization) are foundational where without them, the system cannot function properly. Predictive capabilities (Environment Prediction) and adaptive parameter tuning (Adaptive Config. Managers) are essential for high-quality operation but the system can limp along in their absence, albeit with significantly degraded performance. However, this hierarchy is not absolute and component importance may vary with operational context. In the tested Medium-uncertainty scenario with moderate obstacle density, for example, Environment Prediction had comparable impact to Domain-Specific adaptation (both MSR = 0.667). In sparser environments, we expect Environment Prediction to become less critical since collision risks are lower and reactive avoidance may suffice. Conversely, in denser environments, where dynamic obstacles are numerous and their trajectories frequently intersect the robot's path, Environment Prediction would likely become the dominant factor—without anticipating future obstacle positions, collision becomes nearly inevitable regardless of other adaptations. Similarly, while sensor faults stressed the General Adaptive Config. Manager in our scenario, actuator faults would elevate the importance of Learning (for detecting control effectiveness degradation) and Domain-Specific adaptation (for goal relaxation when physical capabilities are compromised). This context-sensitivity suggests that while the PANCS architecture provides a complete toolkit, practitioners may prioritize computational resources toward the most scenario-relevant components when operating under tight real-time budgets. Overall, the results validate the integrated architecture where no single component is superfluous, and each contributes distinctly to overall system quality.

9. *PANCS* operational alignment

Building *PANCS* upon the MAPE–K architecture has important implications for future reuse and alignment within larger adaptive ecosystems. Such ecosystems are often characterized by restricted connectivity and distributed decision-making, which necessitate *decentralized self-adaptation*—where each subsystem autonomously manages its own goals, constraints, and adaptations while maintaining coherence with the broader system objectives (Halima et al., 2023). This decentralized nature increases the complexity of modeling and coordinating adaptation processes across multiple agents. To address this, each *PANCS*-enabled CPS instance exposes modular interfaces for its *Monitor*, *Analyze*, *Plan*, and *Execute* components. These standardized interfaces allow other adaptive agents to interact with specific parts of the system, facilitating information exchange and coordination without compromising autonomy. To answer *RQ3*, we demonstrate architectural alignment by instantiating *PANCS* within two established decentralized MAPE-K patterns, among others, from the literature (Weyns et al., 2013): the *Hierarchical Control Pattern* (Fig. 8-Top) and the *Information Sharing Pattern* (Fig. 9-Top). For each pattern, we provide a concrete multi-agent scenario description and detailed mappings to show how *PANCS* components expose and consume coordination interfaces to function as both a subordinate (receiving guidance from higher-level coordinators) and a peer (sharing information with other agents).

Consider a large-scale aircraft assembly facility in which multiple specialized *PANCS*-equipped mobile robots should collaborate with other MAPE-K-equipped agents to construct aircraft fuselages and wings. Different robot types transport and install various components such as fuselage panels, wing sections, landing gear assemblies, and avionics modules. Other agents, such as robotic assembly arms mounted on gantries, automated inspection drones performing quality checks, and an inventory management system tracking component locations, participate in the same production process. The overall aim is to synchronize the delivery and installation of these heavy, precision-critical components while maintaining strict safety standards and assembly sequence requirements, avoiding idle waiting of robotic arms or production bottlenecks caused by missing parts or timing mismatches.

The *Hierarchical Control Pattern* is particularly relevant for coordinating multiple MAPE-K instances in distributed environments. For example, as shown in the bottom part of Fig. 8, a three-layer hierarchical control pattern in the aircraft assembly facility coordinates the operations. At the bottom layer of this pattern, individual *PANCS*-equipped mobile robots handle component transport with continuous adaptation for dynamics, obstacle avoidance, and real-time efficient trajectory optimization, while other MAPE-equipped agents, such as robotic assembly arms, perform installation operations, and automated inspection drones verify positioning accuracy. At the middle layer, Zone Coordinators oversee specific assembly areas and their approach corridors. Each zone coordinator aggregates monitoring data from all robots and agents within its scope, analyzes traffic patterns and potential timing conflicts, plans coordination strategies including speed limits and right-of-way priorities, and executes these parameters by updating operational constraints for lower-level agents. For example, when multiple robots approach an assembly station, the zone coordinator detects potential congestion, computes de-confliction trajectories, and adjusts robot speeds to ensure synchronized arrival. At the top layer, a Facility Production Coordinator monitors overall assembly progress across multiple stations, analyzes production schedules and resource utilization, plans high-level resource allocation and task assignments across zones, and executes facility-wide operational policies. This coordinator determines station priorities and allocates robot fleets to different assembly operations.

In the information sharing configuration, as seen in the bottom part of Fig. 9, *PANCS*-equipped robots and other MAPE-based agents operate with greater autonomy through peer-to-peer data exchange rather than hierarchical coordination. Each agent's Monitor component continuously broadcasts relevant information,e.g.,. current position, velocity, intended destination, local environmental observations to nearby peers within communication range. When multiple robots navigate toward assembly stations through intersecting corridors, they share monitoring data to enable autonomous coordination without central direction. For instance, when a *PANCS*-equipped wing transport robot detects corridor congestion through its sensors, it broadcasts this observation to peers. Nearby landing gear delivery robots' Analyze components process this shared information alongside their own sensor data and autonomously adjust trajectories to avoid conflicts. During critical synchronized operations such as dual-robot wing positioning, two robots may intensively share real-time position, orientation, and force measurements. Each robot's controller uses the peer's shared data to continuously adjust its own control outputs, enabling coordinated motion that keeps the wing section level and stress-free without any explicit supervisory command.

These scenarios demonstrate how *PANCS*'s foundation on the MAPE–K ensures compatibility with heterogeneous agent ecosystems and enables its seamless alignment into complex, multi-agent adaptive

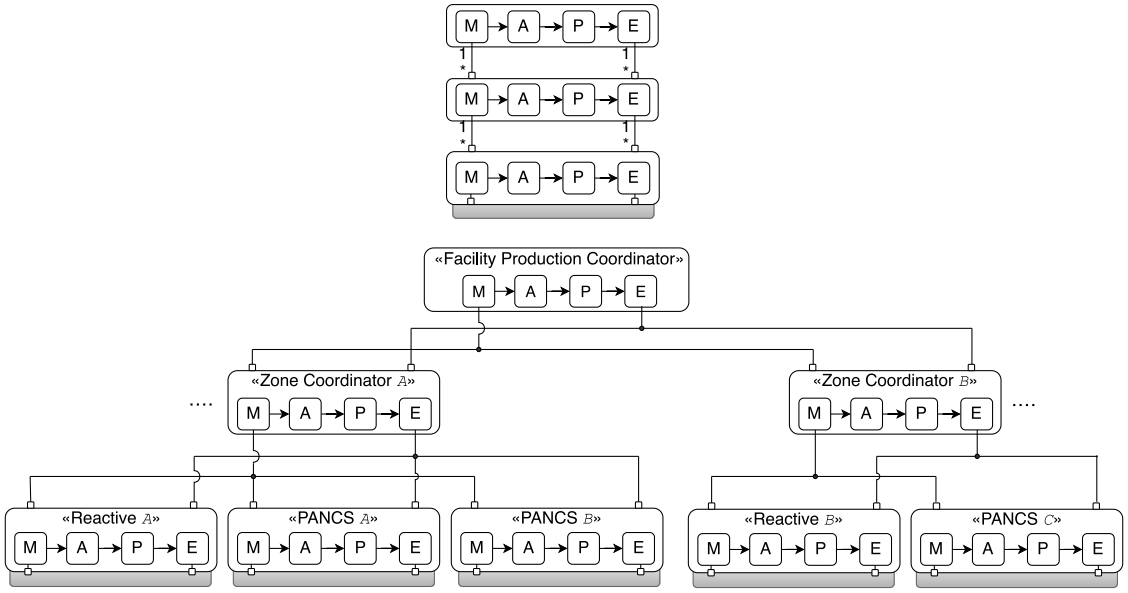


Fig. 8. Top: Hierarchical Control Pattern (Weyns et al., 2013). Bottom: Concrete instance of the pattern for coordinating the aircraft assembly operations.

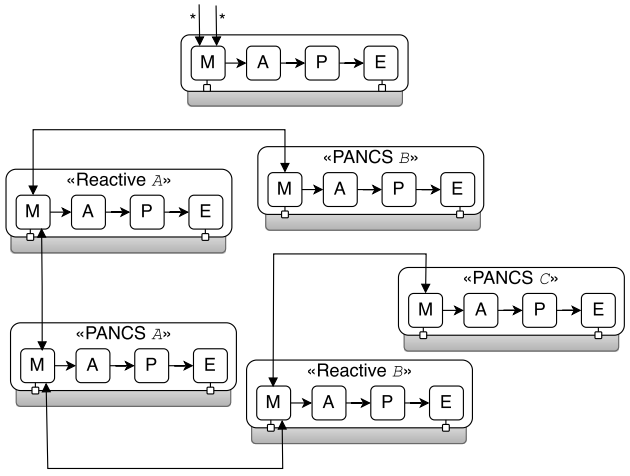


Fig. 9. Top: Information Sharing Pattern (Weyns et al., 2013). Bottom: concrete instance of the pattern for coordinating the aircraft assembly operations.

ecosystems through its modular component interfaces. Indeed, *PANCS*-equipped agents can participate in diverse coordination patterns without architectural modification, through only configuration changes or interface connections. This architectural alignment positions *PANCS* as a foundational element for building large-scale, decentralized CPS where continuous control meets adaptive coordination.

10. *PANCS* in action

Although the validation of *PANCS* was demonstrated in robotic contexts, its architectural modularity enables direct reuse across a variety of nonlinear CPS domains. To answer *RQ4*, this section illustrates how the same architectural structure applies across two other domains than robotics without requiring redesign. In particular, we provide detailed component-level mappings for each domain by specifying *Monitor* sensor selections, *Analyze* mechanism implementations for uncertainty types, and Constraint Assessment mappings to domain-specific constraints, *Plan* model structures and learning approaches, *Execute* actuator interfaces, and *Knowledge* organization schemas.

Healthcare. In the healthcare domain, *PANCS* can be instantiated for a drug delivery system that regulates blood pressure through hormone-based physiological dynamics. The nonlinear feedback mechanism described in Ottesen (1997, 2000) captures the coupled evolution of arterial and venous pressures and hormone concentration, introducing time delays and nonlinearities that make real-time self-adaptation challenging. The main goal is to maintain arterial pressure within safe physiological limits. Soft goals include avoiding overshoot and unsafe dosages, ensuring fault tolerance when sensors or actuators degrade, minimizing drug use, and preserving timeliness despite computational delays. To achieve these goals, the system relies on actuator commands that directly influence physiological states: the drug infusion rate, the rate of change of infusion, and timing parameters for execution. By adjusting these commands, the controller indirectly regulates arterial pressure and hormone levels while respecting safety constraints.

The *Env. Monitoring* subcomponent collects contextual data such as patient posture, ambient temperature, and motion intensity, which influence blood pressure regulation. The *MgS. Monitoring* subcomponent acquires physiological signals such as arterial pressure, venous pressure, and proxy hormone concentration, performing denoising, and missing-sample interpolation.

Internal uncertainty includes sensor drift or actuator degradation, detected through residual analysis between predicted and measured arterial pressure. External uncertainty corresponds to sudden hemodynamic changes due to movement or stress. Goal uncertainty arises when the target pressure cannot be safely achieved due to model mismatch or actuator limits. Diagnostic outcomes are forwarded to the *Constraint Assessment* and *Plan* components.

Constraint Assessment converts analysis results into quantitative constraint updates for the AMPC. For example, as an input constraint, if the infusion actuator efficiency drops, the maximum drug rate is scaled by a degradation factor to avoid overdosing. For the output constraint, when pressure sensors become noisy, the tolerance bounds could be widened relative to the variance magnitude of the sensor output to prevent false safety violations. These constraint updates are sent to the *Adaptive Config. Manager*, which applies them to the optimization problem. Indeed, the *Adaptive Config. Manager* reconfigures

AMPC parameters according to analysis results. When a sensor fault is detected, for example, it increases the measurement noise covariance Q_m to reduce the sensor's influence. When the physiological response slows down, the prediction horizon is extended, while penalty weights are adjusted to preserve real-time feasibility.

The *Action Translator* converts optimized control outputs into actuator commands such as infusion rate adjustments and timing parameters, while the *Execution Scheduler* ensures commands are applied at correct intervals, buffering premature actions or reusing valid commands during computational delays. Timing deviations and actuator feedback are reported back to the *Knowledge* base to support further adaptation. The *Knowledge* component organizes environmental, hardware, and software information in separate layers.

Dynamic Wireless Power Transfer (DWPT). DWPT is a technology that enables electric vehicles (EVs) to charge while moving along the road. Instead of stopping and plugging in, the road contains inductive coils embedded beneath its surface, powered by the electrical grid, that generate a magnetic field. The vehicle has a receiver coil that captures this energy and converts it into electricity to charge the battery. The coupling between the road coils and the vehicle coil changes continuously as the car moves, introducing nonlinearities and real-time control challenges (Gomes et al., 2025).

Within the *PANCS* view, the root goal of DWPT is stable power transfer to charge a moving vehicle. This goal can be AND-refined into two main subgoals: generating an alternating magnetic field on the road side, and capturing magnetic energy and converting it into battery-compatible direct current (DC) on the vehicle side. Soft goals include fault tolerance, high efficiency, robustness under varying conditions, and real-time performance for continuous adaptation. To achieve these goals, the control software dynamically adjusts switching frequency, phase angle, and voltage amplitude on the transmitter side. These adjustments compensate for run-time uncertainties such as changes in magnetic coupling caused by vehicle movement, coil misalignment, and variations in load conditions.

The *Env. Monitoring* subcomponent measures environmental and operational context, such as vehicle position and speed, and ambient conditions (like environmental temperature). The *MgS. Monitoring* subcomponent collects electrical quantities (current, voltage, and power) in both the road and vehicle sides, coil temperature, etc. Data are filtered and synchronized before being sent to analysis. The *Analyze* component performs analysis on monitored data, e.g., estimation of vehicle future speed, estimation of coupling, detecting overheating, detecting coils missalignment, and detecting deviations between expected and measured values to identify possible faults. These findings are passed to the *Constraint Assessment* and *Plan* layers. For example, in *Input Constraint Assessment*, if overheating is detected, the maximum coil current is reduced, while in the case of misalignment, the voltage limit increases. The *Adaptive Config. Manager* uses these updates to adjust control parameters in real time. For instance, if a fault is detected (e.g., coil overheating, converter failure), it may disable the faulty coil segment and reconfigure active coils to maintain partial charging. If thermal limits are reached, it temporarily relaxes efficiency goals to prioritize safety. The *Action Translator* applies new converter settings (switching frequency and phase commands) to the power electronics interface. The *Execution Scheduler* ensures these updates follow timing constraints, reporting delays or missed cycles back to the *Knowledge* base. The *Knowledge* component stores both static and runtime information, such as coil parameters, learned efficiency models, and adaptation rules linking sensor conditions to control updates. This allows new DWPT installations to reuse the same adaptation logic with only parameter updates.

These examples illustrate how the *PANCS* architecture can be reused across diverse CPS domains. This reuse is enabled by its clear separation of concerns following the *MAPE-K* loop, which improves modularity and maintainability and allows individual components (e.g., fault detection and constraint handling) to be independently developed and adapted. While domain-specific expertise is required to implement concrete mechanisms, the underlying architectural framework remains stable, reducing the engineering effort required to develop proactive self-adaptation for new nonlinear CPS domains. Rather than designing adaptation logic from scratch, engineers can follow PANCS's instantiation template, substituting domain-specific models and mechanisms into the established architectural slots. However, comprehensive empirical validation on physical systems remains important future work and is beyond the scope of this paper.

11. Discussion

The evaluation results presented in Sections 6 and 8 show that *PANCS* outperforms a reactive baseline across the majority of test scenarios in terms of mission success, mission efficiency, and safety. Beyond these quantitative improvements, the results yield several broader insights into the design and engineering of non-linear self-adaptive CPS, with implications for both researchers and practitioners, which are discussed in this section.

Implications for Researchers. *PANCS* draws on control-theoretic foundations. It delivers measurable benefits without requiring additional sensors or hardware upgrades. This highlights the often-underestimated leverage of software engineering in enhancing CPS qualities. *PANCS* demonstrates that the MAPE-K loop, traditionally associated with information systems, can be effectively extended to address the continuous dynamics, real-time constraints, and physical uncertainties characteristic of CPS.

To have interpretable, white-box adaptation logic to enable operators to understand, audit, and override adaptation decisions, Adaptive Config. Manager uses rule-based adaptation rather than learned policies. Therefore, *PANCS* generates structured text-based logs at each control cycle by recording decision-relevant information, including system state, active goals, detected uncertainties, applied constraints, optimization results, etc. These logs facilitate traceability and post-hoc analysis for developers, enabling debugging of unexpected behaviors and validation against operational policies. However, for end users and operators, a graphical visualization tool that interprets and presents log content would be necessary to achieve practical interpretability beyond the development phase.

Although the experimental validation was instantiated in a robotic setting, the architectural principles of *PANCS* are not domain-specific as exemplified in Section 10. It has also been shown, in Section 9, that *PANCS* could be used for separating concerns of nonlinear CPS based on *MAPE-K*. This enables not only the functionality of nonlinear CPS in broader alignment with other self-adaptive agents but also improved maintainability by making individual components (e.g., fault detection, constraint handling) easier to extend and test. These two sections provided the reusability perspectives of the proposed architecture to support further research on CPS self-adaptation. However, research challenges, including formal verification, comprehensive validation across multiple domains, and multi-agent coordination, remain open in this context.

Our experience also shows that architectural transparency matters. Structuring knowledge into static and dynamic layers across environment, hardware, and software dimensions, *PANCS* introduces a principled way to manage runtime models that balances efficiency with clarity. These qualities demonstrate that to develop proactive adaptation in CPS, design choices directly affect transparency and long-term evolvability. Making explicitly modular architecture not only improves maintainability but also makes adaptation outcomes easier to interpret, debug, and validate. This suggests that proactive adaptation frameworks in future research should be designed as explainable architectures, not just as optimization pipelines.

Moreover, the validation across varied uncertainty scenarios underscored the importance of progressive evaluation. Starting with controlled simplified simulations before moving to distributed co-simulation

helped isolate bottlenecks, refine assumptions, and strengthen claims of generalizability. This layered evaluation strategy is itself a lesson for engineering future CPS frameworks.

Implications for Practitioners. From a practical perspective, *PANCS* offers a modular architecture that supports incremental adoption, allowing practitioners to implement selected components as needed. The proposed co-simulation approach enables risk-free validation prior to deployment, while the structured knowledge component (static/dynamic × environment/hardware/software) supports maintainability and system evolution.

Despite its demonstrated benefits on facilitating practical implementation, practitioners should be aware of several considerations associated with the efforts needed for instantiation of *PANCS*. First, *PANCS* requires a reasonably accurate nonlinear dynamic model of the managed system at design time; the Kalman filter and successive linearization can compensate for moderate model-to-reality discrepancies, but a fundamentally inaccurate model cannot be corrected at runtime. Second, the AMPC formulation assumes that the system's dynamics are sufficiently smooth and slowly varying within each sampling interval to permit local linearization; systems with rapid mode switching may violate this assumption. Third, our real-time feasibility analysis (80% runtime budget usage) indicates that although all computations were completed within the sampling deadlines, the safety margin was sometimes narrow in high-complexity scenarios. This highlights an inherent trade-off between the prediction horizon, optimization fidelity, and responsiveness of execution, which must be carefully balanced for different CPS domains. It should also be noted that this analysis characterizes *PANCS*'s computational budget in isolation. In the distributed co-simulation setup, the managing system runs on a dedicated host, which eliminates contention with other processes. In physical deployments where the managing system shares hardware with sensor middleware, communication stacks, and actuation pipelines, system-level resource contention may reduce the effective available budget below the hardware's theoretical capacity. Analyzing such contention requires platform-specific real-time scheduling analysis at system integration time, which is appropriately conducted during deployment engineering rather than at the reference architecture stage. We recommend that practitioners profile the co-resident process loads and adjust the RTF target accordingly before deployment. Fourth, while our evaluation covered environments with varying levels of entropy and system uncertainty, the tested scenarios do not fully reflect the stochasticity, heterogeneity, and multi-agent interactions characteristic of many real-world deployments. Addressing these challenges requires further research into robust model learning, adaptive horizon tuning, and large-scale validation in diverse CPS contexts.

12. Threats to validity

We identify several threats to the validity of our evaluation and discuss mitigation strategies employed.

Internal Validity. The reactive baseline for comparison may not represent the best possible reactive approach, potentially inflating *PANCS* advantages. To mitigate, we considered standard MPC at the start of each evaluation for comparison purposes. Also, we implemented the reactive baseline following established practices in control engineering and self-adaptive systems literature, using the same sensing, actuation, and optimization infrastructure as *PANCS* to ensure fair comparison. However, we acknowledge that more sophisticated approaches exist and should be compared in future work. Another internal validity arises from simulation-to-reality gaps. Although we use Gazebo, a widely adopted robotics simulator, simulation-to-reality gaps remain. Physical phenomena like wheel slip, actuator backlash, and sensor drift may not be fully captured. To mitigate, we introduced stochastic noise models, model uncertainties, and fault scenarios to approximate real-world conditions. The distributed co-simulation architecture was specifically chosen to enable future substitution of simulated components with physical hardware. Moreover, the random generation of obstacle trajectories and initial conditions could inadvertently favor one approach over another if not properly controlled. Therefore, we mitigated it by using the same random seeds for paired experiments (reactive vs. *PANCS*) and conducted several replications per scenario configuration to account for variability. Statistical tests confirmed significance despite this variability.

External Validity. Evaluation involved single-robot instances, limiting conclusions about multi-agent scalability. However, multi-agent empirical evaluation builds necessarily on single-agent validation as the architecture must be well-understood and stable in isolation before the emergent dynamics of concurrent instances can be meaningfully studied. Section 9 provides architectural patterns for coordination, and the MAPE-K foundation supports decentralized operation by design. Empirical investigation of multi-agent *PANCS* deployments, including coordination overhead, conflict resolution, and scalability, is essential future work. Also, the primary evaluation focused on warehouse robotics, which may not represent all nonlinear CPS domains. To mitigate this threat, we provided architectural mappings for healthcare drug delivery and dynamic wireless power transfer systems, in Section 10, to demonstrate conceptual generalizability. However, comprehensive empirical validation on physical systems in these domains remains important future work and is beyond the scope of this paper. While we included sensor faults, gradual actuator degradation, and sudden actuator failures, other uncertainty types, like cyber-attacks, were not evaluated. However, the modular *Analyze* component design allows adding new uncertainty recognition mechanisms without architectural changes.

Construct Validity. The chosen metrics (MSR, SR, PE, ME, RTF) may not capture all relevant quality dimensions. For instance, energy consumption, user experience, and long-term wear were not measured. However, we selected metrics based on established robotic systems literature, covering functional correctness (MSR), safety (SR), efficiency (PE, ME), and feasibility (RTF). We acknowledge that additional and different domain-specific metrics may be relevant for particular applications.

13. Conclusion and future works

This paper has targeted a critical gap in self-adaptive CPS engineering: the lack of reusable architectural solutions for proactive adaptation in nonlinear CPS operating under real-time constraints. We presented *PANCS* to enable CPS to anticipate and prepare for future states rather than merely reacting to violations after they occur. Our experimental evaluations demonstrated that proactive adaptation through *PANCS* delivers improvements over reactive approach across multiple dimensions, including mission success ratio, safety, and efficiency, while maintaining real-time feasibility. This underscores the practical value of architectural self-adaptation investment in proactive adaptation. These gains were obtained without requiring additional sensors or hardware modifications, highlighting the often-underestimated leverage of software architecture in enhancing CPS quality. Beyond quantitative improvements, this work contributes to the broader software engineering discourse on self-adaptive systems by demonstrating that proactive adaptation in CPS is not solely a control problem but fundamentally a software architecture problem. The architectural reusability of *PANCS* was demonstrated through instantiation examples in other domains, showing how the same architectural principles and component structure can be adapted to diverse nonlinear CPS domains. Furthermore, the alignment with established MAPE-K patterns enables *PANCS* instances to participate in larger adaptive ecosystems.

While the current work has focused on validation within controlled experimental settings, the distributed co-simulation approach and consideration of real-world uncertainties (noise, model discrepancies, faults) represent significant steps toward practical deployment.

The results suggest that *PANCS* provides a viable path forward for engineering reliable, safe, and efficient self-adaptive CPS in domains where anticipatory decision-making can mitigate risks and improve operational quality. Accordingly, future work includes physical hardware-in-the-loop validation and real-world deployment of *PANCS*. Additionally, as noted in the paper, *PANCS* can serve as a building block in larger adaptive ecosystems. We therefore plan to investigate scenarios in which *PANCS*-equipped CPS collaborate and coordinate with other self-adaptive CPS.

CRediT authorship contribution statement

Farid Edrisi: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Conceptualization. **Diego Perez-Palacin:** Writing – review & editing, Writing – original draft, Supervision, Formal analysis, Conceptualization. **Mauro Caporuscio:** Writing – review & editing, Supervision. **Raffaela Mirandola:** Writing – review & editing, Supervision, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was partially supported by the Helmholtz Association with the KiKIT project and Helmholtz Grant 46.23 (Engineering Secure Systems) and by the German Research Foundation (DFG) - SFB 1608 - 501798263.

Appendix A. Frequent successive linearization

Employing a successive linearization approach improves approximation quality by preventing error accumulation. The linearization could be done in an open-loop fashion, where a single linearization is reused over time, or in a closed-loop and successive way, which is the approach adopted by *PANCS*. In the open-loop approach:

`Time k=0: Linearize at` $(s(0), u(0)) \rightarrow A^0, B^0$
`Time k=1: Predict` $(\hat{s}(1))$ `using` A^0, B^0 `(no measurement update)`
`Time k=2: Predict` $\hat{s}(2)$ `using` A^0, B^0 `(linearization now stale)`
`...`
`Result: Linearization becomes increasingly invalid as system state drifts: Error ACCUMULATES across timesteps.`

In successive linearization (*PANCS* approach):

`Time k=0: Linearize at` $(s(0), u(0)) \rightarrow A^0, B^0$, apply $u(0)$
`Time k=1: MEASURE actual` $s(1)$, `RE-LINEARIZE at` $(s(1), u(1)) \rightarrow A^1, B^1$
`Time k=2: MEASURE actual` $s(2)$, `RE-LINEARIZE at` $(s(2), u(2)) \rightarrow A^2, B^2$
`...`
`Result: Linearization always around CURRENT measured state: Error does NOT accumulate between timesteps.`

PANCS performs linearization at every control step, ensuring that the linear model A^k, B^k is always computed around the current operating point $s(k), u(k)$ obtained from sensor measurements. The Kalman filter further corrects prediction errors by incorporating actual measurements, preventing drift. To empirically validate this claim, we experimented with the running example, varying linearization update frequency while holding all other parameters constant. It is important to note that the linearization frequency is independent of the sampling time T_s where the sampling time T_s determines how frequently the MAPE-K loop executes and control actions are applied to the system and the linearization frequency determines how often the system model matrices (A^k, B^k) are recomputed during operation. In our experiments, we fix T_s for each experimental configuration and vary only the linearization update rate to isolate its effect on model accuracy.

Fig. A.10 shows the logarithmic results of the linearization error (blue) and computational load (red) versus linearization frequency at different sample times (T_s). We define the linearization error as follows; First, at each timestep k, we compute the instantaneous normalized error: $e^i(k) = \frac{|s^i_{\text{nLin}}(k) - s^i_{\text{Lin}}(k)|}{L^i}$ where $(.)_{\text{nLin}}$ denotes the actual system state from the nonlinear simulation, $(.)_{\text{Lin}}$ denotes the state predicted using the linearized model, L^i is acceptable error for each state i (we consider 5% of the state's overall scale). This normalization ensures that each term contributes proportionally to the overall error metric. Second, we compute the root mean square (RMS) of these errors across all timesteps as $\text{RMS}_e = \frac{1}{N}\sum_{k=1}^{N}\sqrt{\frac{1}{M}\sum_{i=1}^{M} e^i(k)^2}$ where M = *number of states* and N = *total number of timesteps (simulation Time/T_s)*. Based on Fig. A.10, the mean linearization error decreases with increasing update frequency, confirming that frequent re-linearization improves accuracy. It should be noted that the total execution time is always negligible for linearization across all frequencies (maximum = 0.02 sec for a 60-second run).

However, we acknowledge that within a single prediction horizon, linearization error can grow because predictions $\hat{s}(k+1), \hat{s}(k+2), \ldots, \hat{s}(k+p)$ are computed using a fixed linearization A^k, B^k without measurement updates. This linear growth is acceptable because in AMPC only the first action is executed. It means the receding horizon principle applies only $u(k)$, discarding the farther predictions that have higher error. Then, at $k+1$, the entire horizon is recomputed with fresh measurements and re-linearization, limiting exposure to accumulated intra-horizon error to a single timestep.

Appendix B. Simplified use case-model

The nonlinear model of the simplified use case follows the Ackermann kinematic equations, which is used to model a car-like vehicle with an Ackermann-steering mechanism. The equation adjusts the position of the axle tires based on the track width so that the tires follow concentric circles (Lynch and Park, 2017). The model describes the ground vehicle behavior, which consists of the position, attitude, and velocity variables.

$$S : \{x, y, \theta, v\}$$

where x and y represent the vehicle position coordinates, θ denotes the heading angle, and v is the linear speed. The vehicle dynamic behavior (i.e., the change of value) of each of its variables follows the equation:

$$\begin{aligned}
\dot{x}(t) &= v(t)\cos(\theta(t)) + n_x(t),\\
\dot{y}(t) &= v(t)\sin(\theta(t)) + n_y(t),\\
\dot{\theta}(t) &= \frac{v(t)\tan(\delta(t)/sc_\delta)}{VL} + n_\theta(t),\\
\dot{v}(t) &= 0.5T(t)/sc_T + n_v(t),
\end{aligned} \tag{B.1}$$

where t is the continuous time, $n_{(\cdot)}(t)$ describes the unknown stochastic noise affecting each state, and VL is the vehicle baseline length. The variables that the robot can use to control its operation are $\{\delta, T\}$. To include actuator faults, we introduce a set of scale-factor parameters as $sc = \{sc_\delta, sc_T\}$ where each $sc_i \in [0, 1]$ for $i \in \{\delta, T\}$ represents the multiplicative degradation factor associated with the corresponding actuator. Accordingly, the effective control input is expressed as $u = \{\frac{1}{sc_\delta}\delta, \frac{1}{sc_T}T\}$ where the nominal (fault-free) condition corresponds to $sc_i = 1$.

Consequently, the Linearization component described in Section 5.1.2 transforms the nonlinear equations in Eq. (B.1) into a linear approximation by computing the Jacobian matrix of partial derivatives as defined in Eq. (3). For example, the entry $A_{[1,3]}$ captures the sensitivity of the first state derivative ($\dot{x}(t)$) with respect to the current state of

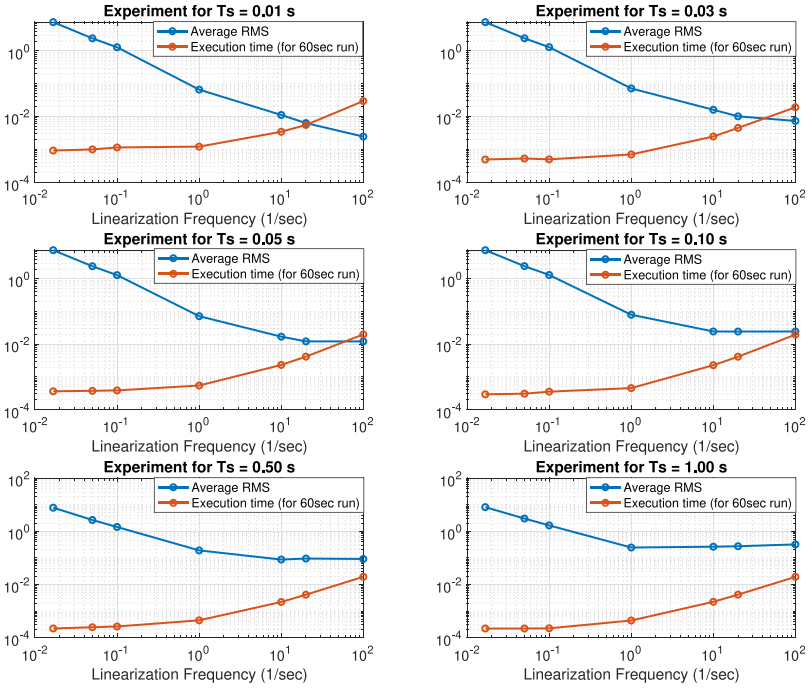


Fig. A.10. Logarithmic results of the linearization error (blue) and computational load (red) versus Linearization frequency at different sample times.

the third variable (θ (t)) at the time t^* when linearization is performed, resulting in $A_{[1,3]}(t^*) = \frac{\partial \dot{x}(t)}{\partial \theta(t)} \mid_{t=t^*} = -v(t^*)sin(\theta(t^*))$.

After linearization, the equations are discretized for time steps k using Eq. (4) (ignoring the higher order of A for small T_s). The resulting matrices and vectors are in a structure consistent with the form of Eq. (2) as:

$$A_d^k = \begin{bmatrix} 1 & 0 & -T_s v^k \sin(\theta^k) & T_s \cos(\theta^k) \\ 0 & 1 & T_s v^k \cos(\theta^k) & T_s \sin(\theta^k) \\ 0 & 0 & 1 & T_s \tan(\frac{\delta^k}{sc_\delta})/VL \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$B_d^k = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ T_s \frac{v^k}{VL} \sec^2(\delta^k) & 0 \\ 0 & 0.5T_s \end{bmatrix} \cdot \begin{bmatrix} \frac{1}{sc_\delta} & 0 \\ 0 & \frac{1}{sc_T} \end{bmatrix}, \quad C_d^k = I(4), \quad D_d^k = 0, \tag{B.2}$$

$$s_d(k) = \begin{bmatrix} x(k) \\ y(k) \\ \theta(k) \\ v(k) \end{bmatrix}, \quad u_d(k) = \begin{bmatrix} \delta \\ T \end{bmatrix}, \quad n_s(k) = \begin{bmatrix} n_x(k) \\ n_y(k) \\ n_\theta(k) \\ n_v(k) \end{bmatrix}.$$

Note that for considering actuators fault, a diagonal matrix from scale-factor values was multiplied by the input matrix. The model linearization executes once for each execution of the *Planner* since it updates values in A_d^k and B_d^k.

Appendix C. Running example use case- Model

The running example use case leverages the nonlinear model of the differential drive robot, following the kinematic equations of a two-wheeled robot with independent wheel actuation. The robot state is expressed as:

$S : \{x, y, \theta\}$,

where x and y denote the robot global position, and θ is the heading angle. The continuous-time dynamics are:

$$\begin{aligned} \dot{x}(t) &= \tfrac{r_w}{2}\left(\dot{\phi}_R(t)/sc_{\dot{\phi}_R} + \dot{\phi}_L(t)/sc_{\dot{\phi}_L}\right)\cos(\theta(t)) + n_x(t), \\ \dot{y}(t) &= \tfrac{r_w}{2}\left(\dot{\phi}_R(t)/sc_{\dot{\phi}_R} + \dot{\phi}_L(t)/sc_{\dot{\phi}_L}\right)\sin(\theta(t)) + n_y(t), \\ \dot{\theta}(t) &= \tfrac{r_w}{d}\left(\dot{\phi}_R(t)/sc_{\dot{\phi}_R} - \dot{\phi}_L(t)/sc_{\dot{\phi}_L}\right) + n_\theta(t) \end{aligned} \tag{C.1}$$

where r_w is the wheel radius, d is the track width, and $n_{(.)}(t)$ are noise terms. Similar to the simplified case, the effective control input vector is $u : \{\frac{1}{sc_{\dot{\phi}_L}}\dot{\phi}_L, \frac{1}{sc_{\dot{\phi}_R}}\dot{\phi}_R\}$ where $\dot{\phi}_L, \dot{\phi}_R$ are left and right wheels speed. Scale-factor parameters are shown as sc, where each $sc_i \in [0, 1]$ for $i \in \{\dot{\phi}_L, \dot{\phi}_R\}$ describes the multiplicative degradation factor associated with the corresponding actuator. After linearization and discretization of (C.1) for time steps k using Eqs. (3) and (4), the resulting matrices and vectors will be:

$$A_d^k = \begin{bmatrix} 1 & 0 & -T_s v^k \sin(\theta^k) \\ 0 & 1 & T_s v^k \cos(\theta^k) \\ 0 & 0 & 1 \end{bmatrix},$$

$$B_d^k = \begin{bmatrix} \frac{r_w T_s}{2}\cos(\theta^k) & \frac{r_w T_s}{2}\cos(\theta^k) \\ \frac{r_w T_s}{2}\sin(\theta^k) & \frac{r_w T_s}{2}\sin(\theta^k) \\ -\frac{r_w T_s}{d} & \frac{r_w T_s}{d} \end{bmatrix} \cdot \begin{bmatrix} \frac{1}{sc_{\dot{\phi}_L}} & 0 \\ 0 & \frac{1}{sc_{\dot{\phi}_R}} \end{bmatrix}, \quad C_d^k = I(3), \quad D_d^k = 0, \tag{C.2}$$

$$s_d(k) = \begin{bmatrix} x(k) \\ y(k) \\ \theta(k) \end{bmatrix}, \quad u(k) = \begin{bmatrix} \dot{\phi}_L \\ \dot{\phi}_R \end{bmatrix}, \quad n_s(k) = \begin{bmatrix} n_x(k) \\ n_y(k) \\ n_\theta(k) \end{bmatrix}$$

Appendix D. Experimental results

See Table D.5.

Data availability

Data will be made available on request.

Table D.5
Experimental results for *PANCS* and baseline approach (– is used for not finished tasks, where computation is not done).

MSR_{re}	MSR_{PANCS}	SR_{re}	SR_{PANCS}	RTF_{re}	RTF_{PANCS}	PE_{re}	PE_{PANCS}	ME_{re}	ME_{PANCS}	Uncertainty External	Internal
1.000	1.000	4.834	1.280	46.070	87.559	20.680	20.820	13.029	12.541	E	–
0.000	1.000	60.050	15.766	72.846	71.776	–	25.787	–	13.953	E	–
1.000	1.000	7.217	12.056	47.202	70.496	20.780	24.050	13.017	13.267	E	–
1.000	1.000	21.653	11.419	50.461	74.516	25.240	27.433	15.410	14.868	E	–
1.000	1.000	10.719	6.443	51.220	71.118	23.132	21.104	14.143	11.632	E	–
0.800	1.000	17.778	34.887	58.379	90.965	–	42.648	–	16.597	E	–
1.000	1.000	5.468	0.000	43.846	71.971	23.160	24.280	14.322	13.640	E	–
1.000	1.000	19.156	0.415	52.314	74.544	22.756	19.264	13.661	10.766	E	–
1.000	1.000	7.916	0.000	45.441	72.776	16.500	16.767	9.560	8.244	E	SF
0.600	1.000	61.892	21.322	70.815	86.075	–	22.696	–	11.303	E	SF
1.000	1.000	15.940	9.147	52.473	86.946	23.584	21.860	13.112	11.276	E	SF
1.000	1.000	24.188	11.453	49.427	81.564	36.927	26.187	21.119	13.781	E	SF
0.600	1.000	42.482	11.379	68.274	83.498	–	19.291	–	10.183	E	AGF
0.000	1.000	16.817	21.662	65.056	91.760	–	28.541	–	17.833	E	AGF
1.000	1.000	21.730	14.572	63.793	81.172	27.017	19.175	14.331	11.204	E	ASF
1.000	1.000	8.609	2.753	58.164	86.417	20.303	17.649	12.791	11.822	E	ASF
0.333	0.667	37.785	30.888	59.423	92.049	–	–	–	–	M	–
1.000	1.000	19.118	8.969	61.045	71.897	20.135	20.807	10.712	10.324	M	–
0.333	1.000	46.360	2.905	62.637	70.765	–	22.940	–	13.473	M	–
0.667	1.000	19.048	2.053	52.277	72.224	–	25.973	–	14.164	M	–
1.000	1.000	27.647	19.469	66.586	80.260	19.767	22.593	12.054	11.844	M	–
0.000	1.000	20.578	5.029	57.440	72.962	–	25.180	–	10.551	M	–
1.000	1.000	35.716	0.973	59.106	80.903	28.720	27.400	18.300	8.874	M	–
1.000	1.000	30.752	10.762	60.921	72.041	17.553	22.913	10.902	12.397	M	–
1.000	1.000	7.916	0.000	45.441	72.776	16.500	16.767	9.560	8.244	M	SF
0.000	1.000	57.643	0.000	59.345	72.781	–	22.927	–	12.764	M	SF
0.500	0.500	14.851	25.665	51.385	93.355	–	–	–	–	M	SF
1.000	1.000	31.694	34.005	56.021	88.611	45.740	33.220	23.980	16.090	M	SF
0.000	1.000	26.873	17.248	58.042	80.779	–	21.374	–	16.884	M	AGF
0.600	0.600	43.986	31.097	68.602	91.648	–	–	–	–	M	AGF
1.000	1.000	8.733	1.474	55.438	75.825	17.904	20.041	9.883	9.071	M	ASF
0.250	1.000	58.295	24.031	50.754	87.668	–	24.690	–	12.873	M	ASF
0.333	1.000	40.933	2.779	62.993	74.104	–	23.987	–	13.379	H	–
0.600	0.800	29.825	27.717	59.485	94.664	–	–	–	–	H	–
1.000	1.000	25.550	17.601	50.269	81.475	47.740	30.670	26.644	15.187	H	–
0.667	1.000	35.223	12.550	59.981	80.757	–	28.147	–	13.278	H	–
0.000	0.333	67.565	54.100	68.049	92.938	–	–	–	–	H	–
0.000	1.000	53.168	25.210	64.554	95.302	–	40.453	–	17.052	H	–
1.000	1.000	35.247	18.661	60.834	92.456	27.732	31.292	14.377	12.466	H	–
0.667	1.000	29.673	25.013	58.223	92.059	–	38.373	–	24.766	H	–
0.000	1.000	46.888	18.423	66.279	89.433	–	29.667	–	14.709	H	SF
0.333	0.333	44.167	29.694	56.756	81.052	–	–	–	–	H	SF
0.500	0.750	18.319	14.663	54.481	89.143	–	–	–	–	H	SF
0.200	1.000	30.233	9.937	60.179	86.757	–	22.136	–	11.067	H	SF
0.200	0.600	56.224	33.569	74.025	88.436	–	–	–	–	H	AGF
0.000	0.667	49.359	29.287	81.653	99.857	–	–	–	–	H	AGF
0.500	0.500	42.098	24.289	81.156	94.916	–	–	–	–	H	ASF
0.000	1.000	38.903	26.262	77.505	93.237	–	27.058	–	17.902	H	ASF

References

Adetola, V., DeHaan, D., Guay, M., 2009. Adaptive model predictive control for constrained nonlinear systems. Systems Control Lett. 58 (5), 320–326.

Angelopoulos, K., Papadopoulos, A.V., Souza, V.E.S., Mylopoulos, J., 2018. Engineering self-adaptive software systems: From requirements to model predictive control. ACM Trans. Auton. Adapt. Syst. 13 (1).

Ayala, I., Papadopoulos, A.V., Amor, M., Fuentes, L., 2021. ProDSPL: Proactive self-adaptation based on dynamic software product lines. J. Syst. Softw. 175, 110909.

Azari, M.S., Santini, S., Edrisi, F., Flammini, F., 2025. Self-adaptive fault diagnosis for unseen working conditions based on digital twins and domain generalization. Reliab. Eng. Syst. Saf. 254, 110560.

Barišić, A., Ruchkin, I., Savić, D., Mohamed, M.A., Al-Ali, R., Li, L.W., Mkaouar, H., Eslampanah, R., Challenger, M., Blouin, D., et al., 2022. Multi-paradigm modeling for cyber–physical systems: A systematic mapping review. J. Syst. Softw. 183, 111081.

Broy, M., Schmidt, A., 2014. Challenges in engineering cyber-physical systems. Computer 47 (2), 70–72.

Camilli, M., Mirandola, R., Scandurra, P., 2021. Runtime equilibrium verification for resilient cyber-physical systems. In: 2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems. ACSOS, IEEE, pp. 71–80.

Chen, Z., Jiao, W., 2022. A proactive self-adaptation approach for software systems based on environment-aware model predictive control. In: 2022 IEEE 22nd International Conference on Software Quality, Reliability and Security. QRS, pp. 992–1003.

Chen, Z., Li, J., Li, N., Jiao, W., Kang, E., 2024. Context-aware proactive self-adaptation: A two-layer model predictive control approach. ACM Trans. Auton. Adapt. Syst. Just Accepted.

de Lemos, R., et al., 2013. Software engineering for self-adaptive systems: A second research roadmap. In: Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers. Springer Berlin Heidelberg, pp. 1–32.

Edrisi, F., Perez-Palacin, D., Caporuscio, M., Giussani, S., 2023. Adaptive controllers and digital twin for self-adaptive robotic manipulators. In: 2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS, pp. 56–67.

Edrisi, F., Perez-Palacin, D., Caporuscio, M., Mirandola, R., 2025. Approaching proactive self-adaptation in nonlinear cyber-physical systems. In: 2025 IEEE/ACM 20th Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS, pp. 25–31.

Gesser, R.S., Lima, D.M., Normey-Rico, J.E., 2018. Robust model predictive control: Implementation issues with comparative analysis. IFAC-PapersOnLine 51 (25), 478–483.

Ghahremani, S., Giese, H., 2021. Hybrid planning with receding horizon: A case for meta-self-awareness. In: 2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C). IEEE, pp. 131–138.

Gomes, Z.M., Moussa, H., Gall, Y.L., Prado, E.O., Damm, G., Pinheiro, J.R., Ripoll, C., 2025. A nonlinear state-space model and control algorithm for a dynamic wireless power transfer system electric vehicle charger application. Control Eng. Pract. 158, 106270.

Gonzalez, R., Fiacchini, M., Alamo, T., Guzmán, J.L., Rodríguez, F., 2011. Online robust tube-based MPC for time-varying systems: A practical approach. Internat. J. Control 84 (6), 1157–1170.

Halima, R.B., Hachicha, M., Jemal, A., Kacem, A.H., 2023. Mape-k patterns for self-adaptation in cyber-physical systems. J. Supercomput. 79 (5), 4917–4943.

Hielscher, J., Kazhamiakin, R., Metzger, A., Pistore, M., 2008. A framework for proactive self-adaptation of service-based applications based on online testing. In: European Conference on a Service-Based Internet. Springer, pp. 122–133.

Iqbal, K., 2021. Introduction to control systems. Univ. Ark. At Little Rock.

Jetto, L., Orsini, V., 2020. A robust least squares based approach to min-max model predictive control. Internat. J. Robust Nonlinear Control 30 (13), 4807–4825.

Klös, V., Göthel, T., Glesner, S., 2018. Runtime management and quantitative evaluation of changing system goals in complex autonomous systems. J. Syst. Softw. 144, 314–327.

Krupitzer, C., Roth, F.M., VanSyckel, S., Schiele, G., Becker, C., 2015. A survey on engineering approaches for self-adaptive systems. Pervasive Mob. Comput. 17, 184–206, 10 years of Pervasive Computing' In Honor of Chatschik Bisdikian.

Luo, Y., Zhou, Y., Zhao, H., Jin, Z., Zhang, T., Liu, Y., Barthaud, D., Yu, Y., 2022. Online adaptation for autonomous unmanned systems driven by requirements satisfaction model. Softw. Syst. Model. 21 (4), 1295–1319.

Lynch, K.M., Park, F.C., 2017. Modern Robotics: Mechanics, Planning, and Control, first ed. Cambridge University Press, USA.

Mayne, D.Q., Rawlings, J.B., Rao, C.V., Scokaert, P.O.M., 2000. Constrained model predictive control: Stability and optimality. Automatica 36 (6), 789–814. http://dx.doi.org/10.1016/S0005-1098(99)00214-9.

Moreno, G.A., Cámara, J., Garlan, D., Schmerl, B., 2015. Proactive self-adaptation under uncertainty: a probabilistic model checking approach. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. pp. 1–12.

Moreno, G.A., Camara, J., Garlan, D., Schmerl, B., 2016. Efficient decision-making under uncertainty for proactive self-adaptation. In: 2016 IEEE International Conference on Autonomic Computing. ICAC, IEEE, pp. 147–156.

Moreno, G.A., Cámara, J., Garlan, D., Schmerl, B., 2018. Flexible and efficient decision-making for proactive latency-aware self-adaptation. ACM Trans. Auton. Adapt. Syst. (TAAS) 13 (1), 1–36.

Moreno, G.A., Papadopoulos, A.V., Angelopoulos, K., Cámara, J., Schmerl, B., 2017. Comparing model-based predictive approaches to self-adaptation: Cobra and PLA. In: Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS '17, IEEE Press, pp. 42–53.

Musil, A., Musil, J., Weyns, D., Bures, T., Muccini, H., Sharaf, M., 2017. Patterns for self-adaptation in cyber-physical systems. Multi-Disciplinary Eng. Cyber-Physical Production Syst.: Data Model. Softw. Solutions Handl. Complex Engineering Proj. 331–368.

Norby, J., Tajbakhsh, A., Yang, Y., Johnson, A.M., 2024. Adaptive complexity model predictive control. IEEE Trans. Robot. 40, 4615–4634.

Ottesen, J.T., 1997. Modelling of the baroreflex-feedback mechanism with time-delay. J. Math. Biol. 36 (1), 41–63.

Ottesen, J., 2000. Modelling the dynamical baroreflex-feedback control. Math. Comput. Modelling 31 (4), 167–173, Proceedings of the Conference on Dynamical Systems in Biology and Medicine.

Pandey, A., Moreno, G.A., Cámara, J., Garlan, D., 2016. Hybrid planning for decision making in self-adaptive systems. In: 2016 IEEE 10th International Conference on Self-Adaptive and Self-Organizing Systems. SASO, pp. 130–139.

Park, J., Delgado, R., Choi, B.-W., 2020. Real-time characteristics of ROS 2.0 in multiagent robot systems: An empirical study. IEEE Access 8, 154637–154651.

Pereira, G.C., Wahlberg, B., Pettersson, H., Mårtensson, J., 2023. Adaptive reference aware MPC for lateral control of autonomous vehicles. Control Eng. Pract. 132, 105403.

Rahideh, A., Shaheed, M.H., Huijberts, H.J.C., 2008. Stable adaptive model predictive control for nonlinear systems. In: 2008 American Control Conference. IEEE, Seattle, WA, USA, pp. 1673–1678. http://dx.doi.org/10.1109/ACC.2008.4586735.

Rakovic, S.V., Kouvaritakis, B., Cannon, M., Panos, C., Findeisen, R., 2012. Parameterized tube model predictive control. IEEE Trans. Autom. Control 57 (11), 2746–2761. http://dx.doi.org/10.1109/TAC.2012.2191174.

Rowell, D., 2002. State-space representation of LTI systems. 1–18.

Vilchez, E., Troya, J., Camara, J., 2024. Towards proactive decentralized adaptation of unmanned aerial vehicles for wildfire tracking. In: Proceedings of the 19th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS '24, Association for Computing Machinery, New York, NY, USA, pp. 56–62.

Wang, Y., Li, H.-X., Yang, H., 2024. Adaptive spatial-model-based predictive control for complex distributed parameter systems. Adv. Eng. Inform. 59, 102331.

Weyns, D., 2020. An Introduction to Self-Adaptive Systems: A Contemporary Software Engineering Perspective. John Wiley & Sons.

Weyns, D., Andersson, J., Caporuscio, M., Flammini, F., Kerren, A., Löwe, W., 2022. A research agenda for smarter cyber-physical systems. J. Integr. Des. Process. Sci. 25 (2), 27–47.

Weyns, D., Schmerl, B., Grassi, V., Malek, S., Mirandola, R., Prehofer, C., Wuttke, J., Andersson, J., Giese, H., Göschka, K.M., 2013. On patterns for decentralized control in self-adaptive systems. In: Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers. Springer, pp. 76–107.

Zacchia Lun, Y., D'Innocenzo, A., Smarra, F., Malavolta, I., Di Benedetto, M.D., 2019. State of the art of cyber-physical systems security: An automatic control perspective. J. Syst. Softw. 149, 174–216.

Zheng, Y., Li, Q., Li, S., 2024. Stability guaranteed model predictive control with adaptive Lyapunov constraint. IEEE Trans. Autom. Sci. Eng. 21 (1), 215–225. http://dx.doi.org/10.1109/TASE.2023.3241235.