Towards MPC-driven Software Adaptation: A Dual-Layer Approach Combining ICNN-based Modeling and Delta-based Tuning

Yitong Shi
Institute of Science Tokyo
Tokyo, Japan
shi.y.0a6a@m.isct.ac.jp

Chenyu Hu
Institute of Science Tokyo
Tokyo, Japan
chenyuhu59@gmail.com

Mingyue Zhang
Southwest University
Chongqing, China
myzhangswu@swu.edu.cn

Nianyu Li
ZGC National Laboratory
Beijing, China
li_nianyu@pku.edu.cn

Jialong Li*
Waseda University
Institute of Science Tokyo
Tokyo, Japan
lijialong@fuji.waseda.jp

Kenji Tei
Institute of Science Tokyo
Tokyo, Japan
tei@comp.isct.ac.jp

***Abstract*—Proactive self-adaptation using Model Predictive Control (MPC) is well studied for software systems operating in dynamic environments. However, its practical adoption is limited by two key challenges. First, the modeling gap: traditional MPC requires state-space models that are difficult to construct for complex software systems, often demanding extensive domain expertise and incurring high computational costs. Second, the tuning gap: configuring MPC to balance abstract and competing objectives—such as performance, resource efficiency, and control stability—typically relies on manual, expert-driven tuning, impeding autonomous operation. To address these challenges, we propose a dual-layer MPC framework. At the lower layer, an Input Convex Neural Network (ICNN) is used to learn complex nonlinear dynamics directly from data, enabling tractable optimization and reducing the modeling burden. At the upper layer, a delta-driven controller manager adaptively tunes the lower-layer MPC by monitoring deviations in system behavior and estimating the impact of different cost components on overall utility, thereby automating the tuning process. We evaluate our approach on SIMDEX, a job scheduling simulator, where it demonstrates superior performance over both the traditional requirements-oriented MPC framework CobRA and the non-adaptive ICNN-based controller, achieving a better balance of performance, resource efficiency, and control stability.**

***Index Terms*—Self-adaptive software system, Proactive adaptation, Model predictive control**

I. Introduction

Modern software systems must operate reliably in complex, dynamic environments, making self-adaptation a critical capability [1], [2]. Control theory offers a principled foundation for engineering such systems, providing mathematically-grounded techniques to ensure that software maintains desired properties despite uncertainty [3]. Among these techniques, Model Predictive Control (MPC) [4], [5] has emerged as a particularly powerful approach for proactive adaptation. By predicting future system behavior, MPC can compute optimal actions in advance, steering the system away from undesirable states [6].

However, MPC originated in the control of physical systems (e.g., chemical plants, robotics), and its direct application to software engineering presents fundamental challenges. Unlike physical systems, which are governed by well-understood laws of physics, software systems exhibit complex, emergent dynamics that are difficult to capture with first-principles models. This discrepancy creates two primary hurdles for applying MPC effectively in the software domain. The first challenge is the modeling gap. Traditional MPC relies on state-space models [7]–[10], but constructing an accurate model for a software system requires extensive domain expertise. Software dynamics are often highly nonlinear and high-dimensional. Simplified linear models lack the expressive power to capture this complexity, leading to poor control performance. Conversely, hand-crafting accurate nonlinear models is exceptionally difficult and computationally expensive. The second challenge is the tuning gap. The objectives for a software controller—such as balancing performance, resource cost, and user experience—are often more abstract and dynamic than those for physical systems. Tuning an MPC controller to effectively manage these competing objectives is a complex task that typically demands significant manual effort, expert intuition, and iterative, post-deployment adjustments [11]. This reliance on manual tuning acts as a major barrier to creating truly autonomous, "expert-free" self-adaptive software systems.

To address these two fundamental challenges, this paper proposes a dual-layer MPC framework tailored for self-adaptive software system. For the modeling gap, our lower-layer controller leverages an Input Convex Neural Network (ICNN) [12]. The ICNN learns complex, nonlinear system

*Corresponding Author: Jialong Li

2151-0849/25/$31.00 ©2025 IEEE
DOI 10.1109/ASEW67777.2025.00052

dynamics directly from data, eliminating the need for manual model derivation. Critically, its guaranteed convexity ensures that the resulting optimization problem is tractable and avoids local minima. For the tuning gap, we introduce a novel, upper-layer controller manager. This manager automates the difficult tuning process by adopting a delta-driven strategy. In our approach, 'delta-driven' specifically refers to a mechanism where adjustments to the controller's parameters (such as cost function weights and the prediction horizon) are determined by the observed deviations ('deltas') in system performance—like tracking errors or suboptimal resource utilization—and the calculated impact of different cost factors on the overall system utility. It intelligently balances trade-offs between tracking error, control costs, and other factors, enabling expert-free deployment.

The main contributions of this paper are as follows:

- We propose an ICNN-based MPC framework that closes the modeling gap for self-adaptive software system by learning complex dynamics from data while ensuring tractable optimization.
- We design a delta-driven controller manager that closes the tuning gap by automating the controller's configuration, which dynamically balances competing objectives at runtime by reacting to performance feedback and cost implications.
- We conduct an empirical evaluation on the SIMDEX simulation platform, demonstrating that our proposed approach outperforms traditional methods in a dynamic job-dispatching scenario.

The remainder of this paper is structured as follows. Section II provides an overview of the foundational concepts, including the MAPE-K model, MPC, and ICNN. Section III elaborates on our proposed methodology, detailing the ICNN-based MPC controller and the delta-driven controller manager. In Section IV, we present our experimental evaluation, including the case study setup and a discussion of the results. Finally, Section V introduces related work and Section VI concludes the paper and outlines future research directions.

II. Background

A. Model Predictive Control

Model Predictive Control (MPC) is a control strategy that leverages a dynamic system model to predict future behavior [4], [5]. At each time step, MPC solves a constrained optimization problem over a finite time horizon to determine an optimal sequence of control actions. This approach, known as receding horizon control, enables MPC to proactively handle system constraints and optimize performance.

A commonly used system model is the linear state-space representation:

$$x_{t+1} = Ax_t + Bu_t \tag{1}$$

$$y_t = Cx_t + Du_t \tag{2}$$

where x_t is the system state, u_t is the control input, and y_t is the system output at time t.

However, real-world systems often operate under uncertainty and disturbances, leading to discrepancies between the model and actual behavior. To mitigate this, MPC is typically combined with a state estimator such as the Kalman filter [13], which uses real-time measurements to correct state predictions and improve control performance under uncertainty. The state update equation is:

$$\hat{x}_{t+1} = A\hat{x}_t + Bu_t + K(y_t - \hat{y}_t) \tag{3}$$

where $\hat{x}$ and $\hat{y}$ denote the estimated state and output, respectively, and K is the Kalman gain matrix that optimally weights the prediction error $(y_t - \hat{y}_t)$ to refine the state estimate.

B. Input Convex Neural Networks

Input Convex Neural Networks (ICNNs) are a class of neural networks specifically designed to ensure that their output is a convex function of the input [12]. This distinctive property combines the high expressive power of deep learning with the tractability and optimization guarantees of convex functions. To enforce convexity, ICNNs adopt specific architectural constraints: (1) all weights beyond the first hidden layer must be non-negative; and (2) all activation functions must be convex and non-decreasing.

These constraints ensure that the network's output, $f(x)$, satisfies the convexity condition for any inputs x_1, x_2 and any $\lambda \in [0, 1]$:

$$f(\lambda x_1 + (1-\lambda)x_2) \leq \lambda f(x_1) + (1-\lambda) f(x_2) \tag{4}$$

This property is significant because, for any convex function, a local minimum is also a global minimum. As a result, optimization becomes more efficient and avoids the pitfalls of local minima. Moreover, the ICNN architecture is particularly well-suited for tasks involving partial input optimization—optimizing a subset of inputs while holding others fixed.

III. Dual-Layer Model Predictive Control

A. Overview

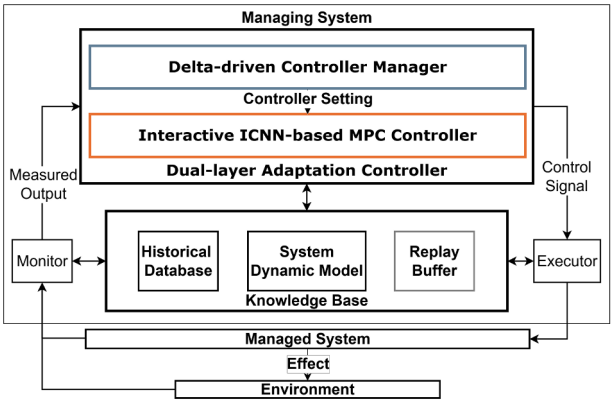

Fig. 1: Proposal Overview.

Figure 1 illustrates the conceptual architecture of our proposal, which follows the classical three-component model of self-adaptive software systems: the *environment*, the *managing system*, and the *managed system* [3], [14]. The environment refers to the external context in which the system operates. The managed system is the functional core, executing the primary tasks of the software and responding to changes in the environment. The managing system oversees adaptation: it monitors the system's behavior, analyzes changes, and adjusts control strategies to ensure the system meets its goals.

Zooming into the managing system, it follows the classical MAPE-K (Monitor, Analyze, Plan, Execute, Knowledge) framework [15], which structures the feedback loop. Specifically, the *Monitor* collects runtime information, the *Knowledge Base* stores prior experience and models, the *Dual-layer Adaptation Controller* performs both *Analyze* and *Plan* functions, and the *Executor* applies adaptation decisions to the managed system.

Within each control cycle, the upper-layer Delta-driven Controller Manager selects high-level control configurations based on system state and environmental feedback gathered by the Monitor and retrieved from the Knowledge Base. These configurations are passed to the lower-layer interactive ICNN-based MPC controller, which performs fine-grained control by predicting future system behavior using the stored dynamic model and computing an optimal control signal that satisfies system constraints. After execution, the controller manager compares the predicted outputs with the actual outputs reported by the Monitor. If discrepancies are significant or retraining is scheduled, new data is collected into the Replay Buffer. Once the buffer is filled, it is combined with historical data from the Historical Database, and the System Dynamic Model is updated accordingly, enabling continuous learning and model refinement over time.

B. ICNN-based MPC controller

1) System Dynamic Model: The system dynamics model [16] describes how a system evolves over time and serves as a core component of MPC. The accuracy of this model's predictions is directly correlated with the actual control performance of the controller. In this paper, we adopt a learning-based approach to model the system dynamics. The model is initially trained offline and is subsequently refined through periodic fine-tuning.

To ensure tractability during optimization, we leverage the property that any local minimum of a convex function is also a global minimum. This significantly reduces the computational complexity of solving the control optimization problem and fundamentally avoids issues associated with local optima.

We employ an ICNN architecture, as illustrated in Figure 2, which is composed of:

- Two hidden layers (Z_1, Z_2),
- Two passthrough layers (D_1, D_2),
- Two activation functions σ (Rectified Linear Unit (ReLU) functions are used throughout to maintain simplicity and ensure convexity).

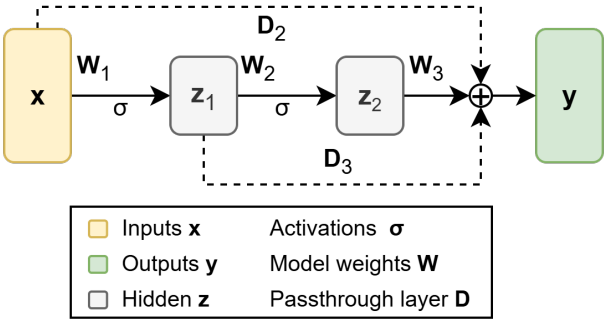

Fig. 2: Input-Convex Feed-Forward Neural Network (ICNN)

The network's final output $\hat{y}$ is computed as:

$$\hat{y} = D_2 + D_3 + Z_2 + b \tag{5}$$

where b denotes the bias term.

This network strictly adheres to the ICNN formulation by ensuring all weights are non-negative and the activation functions are both non-decreasing and convex.

2) Control Logic: The controller follows the classical MPC framework, achieving multi-objective optimization by adjusting multiple control inputs. At each control cycle, the controller leverages the system dynamics model to predict the system's future behavior over a finite prediction horizon. It then solves an online optimization problem to determine the control input sequence $\mathbf{u}_t$ that minimizes a predefined cost function while satisfying various system constraints.

The cost function typically consists of three components: (1) the state tracking error term penalizes deviations from the reference trajectory; (2) the control input magnitude term discourages excessive control efforts; and (3) the control input variation term promotes smooth transitions in control actions.

$$\begin{aligned}
\underset{\mathbf{J}_t}{\text{minimize}} \quad & J_t = \sum_{i=1}^{N_p} \left[(\mathbf{y}_{t+i} - \mathbf{y}_{\text{ref}})^\top \mathbf{Q} (\mathbf{y}_{t+i} - \mathbf{y}_{\text{ref}})\right] \\
& \quad + \mathbf{u}_t^\top \mathbf{R}\, \mathbf{u}_t + \Delta \mathbf{u}_t^\top \mathbf{S} \Delta \mathbf{u}_t, \\
\text{subject to} \quad & \mathbf{y}_{t+1} = f(\mathbf{y}_t, \mathbf{u}_t), \\
& \mathbf{y}_{\min} \le \mathbf{y}_{t+i} \le \mathbf{y}_{\max}, \quad i = 1, ..., H \\
& \mathbf{u}_{\min} \le \mathbf{u}_{t+j} \le \mathbf{u}_{\max}, \quad j = 0, ..., H-1 \\
& \Delta \mathbf{u}_{\min} \le \Delta \mathbf{u}_t \le \Delta \mathbf{u}_{\max}
\end{aligned} \tag{6}$$

where:

- $\mathbf{y}_{t+i} \in \mathbb{R}^n$ is the predicted output vector at time $t+i$ from the system dynamics model;
- $\mathbf{y}_{\text{ref}} \in \mathbb{R}^n$ denotes the reference trajectory (i.e., the desired output);
- $\mathbf{u}_t \in \mathbb{R}^m$ represents the control input vector at time t;
- $\Delta \mathbf{u}_t = \mathbf{u}_t - \mathbf{u}_{t-1}$ is the control input rate vector;
- $\mathbf{Q} \succeq 0 \in \mathbb{R}^{n \times n}$, $\mathbf{R} \succ 0 \in \mathbb{R}^{m \times m}$, and $\mathbf{S} \succ 0 \in \mathbb{R}^{m \times m}$ are positive (semi-)definite weighting matrices;

- $N_p \in \mathbb{Z}^+$ denotes the prediction horizon length;
- $f(\cdot)$ represents the system dynamics model. In our case, this is the ICNN, which predicts the next system output $\mathbf{y}_{t+1}$ from the current output $\mathbf{y}_t$ and control input $\mathbf{u}_t$.

C. Delta-driven controller manager

Algorithm 1 Delta-Driven Control

Input: *Current controller parameters* Q, R, S, N_p, *Performance indicators* I, *Historical data* D
Output: *Adjusted* Q, R, S, N_p

```
    # Initialize
1:  Q, R, S, N_p ← lower-layer MPC controller
2:  I ← Monitor
3:  D ← Knowledge Base
    # Online Update Module
4:  error ← Get_Error(I, D)
5:  IF error > threshold :
6:      replay_buffer ← I, D
7:  END IF
8:  IF replay_buffer.size == replay_buffer.capacity :
9:      D_train ← replay_buffer.pop(0), Knowledge Base
10:     train_model_online(D_train)
11: END IF
    # Cost Function Adaptation Module
12: utility_loss ← Get_Utility_Loss(I, D)
13: nc_Q, nc_R, nc_S ← f(utility_loss)
14: Q, R, S ← Normalization(Q, R, S)
15: Q, R, S ← Q + nc_Q, R + nc_R, S + nc_S
    # Horizon Adaptation Module
16: ce ← Get_Control_Effect(I, D)
17: cc ← Get_Computation_Cost(I, D)
18: IF error > threshold || cc/ce > threshold :
19:     Decrease N_p
20: ELSE IF error < threshold || cc/ce < threshold :
21:     Increase N_p
22: END IF
23: lower-layer MPC controller ← Q, R, S, N_p
```

The core of MPC lies in solving a finite-horizon optimization problem to generate control commands. This problem is typically formulated as shown in Equation 6, where N_p denotes the prediction horizon, and $\mathbf{Q}$, $\mathbf{R}$, and $\mathbf{S}$ are the weighting matrices for the state tracking error, control input magnitude, and control input variation terms, respectively. To better understand and improve MPC performance, we decompose the design into two key factors: (1) the selection of the prediction horizon and (2) the setting of the weighting matrices in the cost function.

Regarding the choice of prediction horizon is particularly challenging in self-adaptive software systems. A longer horizon can improve the global optimality of control decisions, enhance stability, and mitigate oscillations or periodic disturbances. However, it also significantly increases computational complexity. In environments with high variability or limited model accuracy, longer horizons may introduce greater uncertainty, leading to delayed, unnecessary, or incorrect adaptations. Conversely, a shorter horizon reduces computational cost and improves responsiveness to disturbances. In some cases, it may even yield better control performance. However, excessively short horizons can result in overreacting to short-term noise, leading to instability, overshooting, or oscillatory behavior.

For the setting of the weighting matrices in the cost function, the MPC cost function typically consists of three main components. The state tracking error term governs tracking performance and prioritization among multiple objectives, driving the system to follow the reference trajectory as closely as possible. The control input magnitude term penalizes large control actions to reduce operational costs; increasing its weight can improve robustness by reducing sensitivity to high-frequency noise, though it may also slow the system's responsiveness. The control input variation term penalizes rapid changes in control input, helping to suppress abrupt or oscillatory control behaviors and promote smoother system dynamics.

Considering the two design factors discussed above, we propose a delta-driven control algorithm. As illustrated in Algorithm 1, the algorithm takes as input the current controller parameters, performance metrics returned by the monitor, and historical data stored in the knowledge base. In the initialization phase (Lines 1–3), the current controller parameters (Q, R, S, N_p) are obtained from the lower-layer controller; performance metrics (I) are collected from the monitor; and historical data (D) are retrieved from the knowledge base. In the online update module (Lines 4–11), the model error is calculated based on historical data and current performance metrics. If the error exceeds a predefined threshold, relevant data are stored in a replay buffer. Once the buffer is full, historical data from the knowledge base are sampled and combined with new data in the buffer to update the system dynamics model. In the cost function adaptation module (Lines 12–15), the utility loss is computed over a recent time window. To guide the adjustment, this total loss is decomposed into three sources: *System Performance Loss* (related to the $\mathbf{Q}$ term), *Control Actuation Cost* (related to the $\mathbf{R}$ term), and *Control Change Overhead* (related to the $\mathbf{S}$ term). A mapping function f then translates these individual loss components into negative contribution adjustments nc_Q, nc_R, nc_S. The update follows a two-step process: first, the current weights are normalized to prevent their values from drifting over time. Second, the calculated negative contributions are added to these normalized weights, directly modifying their magnitudes based on recent performance feedback. In the horizon adaptation module (Lines 16–21), the controller dynamically balances prediction accuracy against model uncertainty and computational cost. The prediction horizon N_p is decreased when long-term predictions are deemed unreliable (i.e., when 'error' is high) or inefficient (i.e., when the ratio of computational cost to control effect, 'cc/ce', is high, indicating diminishing returns). Conversely, N_p is increased when the system is stable

and control is efficient, allowing the controller to optimize over a longer term.

Note that f can be implemented as either an analytical function or a black-box model. In this paper, f is implemented as a linear function. Additionally, the moving average of utility is used as a proxy for estimating control effect. Specifically, we decompose the utility loss of the managed system into three components, defined as follows:

$$\begin{aligned} &\textit{System Performance Loss} = U_{actural} - U_{max} \\ &\textit{Control Actuation Cost} = \mathrm{C}_{control} \times (U_{max} - U_{expected}) \\ &\textit{Control Change Overhead} = C_{change} \end{aligned} \tag{7}$$

IV. Experimental Evaluation

The evaluation aims to address the following two research questions:

- **RQ2: Effectiveness of the ICNN-based MPC method.** Compared to traditional MPC, does the ICNN-based MPC method achieve better control performance in software systems?
- **RQ1: Effectiveness of the delta-driven controller manager.** Compared to using only the lower-layer MPC controller, does the delta-driven control provided by the upper-layer controller manager improve overall control performance?

A. Target Software System: SIMDEX

We use SIMDEX [17], a simulator for dispatching computing jobs across multiple workers. SIMDEX provides two configurable control actions to manage dynamically changing job loads in real time: (1) adjusting the number of active workers, which corresponds to modifying the number of job processing queues; and (2) adjusting the worker's overtime duration, defined as the maximum job processing time per time step. We normalize the overtime duration to a value between 0 and 1, where 0 indicates no overtime and 1 represents the maximum allowable overtime.

To monitor the system's status, we use six evaluation metrics:

- **Work rate**: The proportion of time all active workers are engaged in processing jobs within a time step, calculated as the average actual working time per worker divided by the average scheduled time.
- **Punctuality**: The ratio of jobs arriving in a given time step that are completed within the same time step.
- **Moving average delay**: The average job delay calculated over a sliding window of recent time steps.
- **Moving max delay**: The maximum job delay observed within the same sliding time window.
- **Total average delay**: The average delay across all time steps throughout the experiment.
- **Total max delay**: The maximum delay recorded during the entire experiment.

B. Experimental Settings

Cost function. The cost function, unified of control inputs, system outputs, evaluation metrics, and a utility equation, is defined as follows:

$$\begin{aligned} U &= U_{actural} - C_{control} - C_{change} - C_{computation} \\ U_{actural} &= \textit{working\_rate} * \textit{worker\_num} * (1 + \text{overtime} * 2) \\ &\quad * (1.5 * \textit{punctuality} + 0.5) \\ U_{max} &= \textit{worker\_num} * (1 + 2 * \textit{overtime}) * \textit{working\_rate} \\ U_{expected} &= \textit{worker\_num} * (1 + 2 * \textit{overtime}) \\ C_{control} &= \textit{worker\_num} * (1 + 4 * \textit{overtime}) \\ C_{change} &= 0.5 * \Delta\textit{worker\_num} + 0.25 * \Delta\textit{overtime} \end{aligned} \tag{8}$$

SIMDEX Scenario settings. Table I summarizes the SIMDEX experiment settings. The system has two control inputs (number of workers [1, 4] and overtime rate [0, 1]) and two outputs (working rate and punctuality). The prediction horizon is fixed at 5, with an additional variable horizon range [1, 10]. Weight matrices Q, R, and S, along with the reference outputs, are also listed.

Name	Type	Value
Worker_num	Control Input	[1, 4]
Overtime Rate	Control Input	[0, 1]
Working Rate	System Output	[0, 1]
Punctuality	System Output	[0, 1]
Fixed Predict horizon N_f	Controller Parameter	5
Variable Predict horizon N_v	Controller Parameter	[1, 10]
Reference System Output	Controller Parameter	[1.0 1.0]
Initial state error weight Q	Controller Parameter	[1.5 1.0]
Initial control input weight R	Controller Parameter	[0.1 0.1]
Initial control rate weight S	Controller Parameter	[0.2 0.2]

TABLE I: Parametor Settings

C. Implementation

We evaluate three methods: the traditional requirements-oriented MPC approach CobRA [7], and the ICNN-based MPC with/without delta-driven tuning.

For CobRA, the system dynamics are modeled using MATLAB's System Identification Toolbox. A linear state-space model is employed, as described in Section 2. The controller is implemented in Python using NumPy and CVXPY for matrix operations and for formulating and solving the optimization problem.

For ICNN-based MPC, the system dynamics model is defined and trained using the PyTorch library. The ICNN architecture, described in Section 3, includes two hidden layers (Z_1, Z_2) and two passthrough layers (D_1, D_2). The model function is given by:

$$
\begin{aligned}
D_2 &= W_{D_2} x, \\
D_3 &= W_{D_3} x, \\
Z_1 &= \sigma(W_1 x + b_{Z_1}), \\
Z_2 &= \sigma(W_2 Z_1 + b_{Z_2}), \\
f(x) &= D_2 + D_3 + W_3 Z_2 + b_y \\
&= W_{D_2} x + W_{D_3} x + W_3 Z_2 + b_y,
\end{aligned} \tag{9}
$$

where σ is the ReLU function, i.e., $\sigma(a) = \max(0, a)$. The weights W_1, W_2, W_3 are constrained to be non-negative to ensure convexity. Passthrough weights W_{D_2}, W_{D_3} are typically unconstrained but may also be restricted for strict convexity. Bias terms b_{Z_1}, b_{Z_2}, b_y are unconstrained. For the controller, the optimization problem is solved iteratively using gradient descent with the Adam optimizer, as implemented in PyTorch.

D. Experiment Results

For RQ1, we compared our proposed baseline method (ICNN w/o D, in red) with the traditional MPC method (CobRA, in blue), as shown in Figure 3 and Table II. The results demonstrate that our baseline method achieves a significantly higher working rate, with an average improvement of 23.8%, and a slight improvement in punctuality by 1.36%. CobRA's poorer performance is mainly due to its imprecise system model, which cannot adequately capture future uncertainty and thus fails to adapt effectively, especially when the system load fluctuates around the 5000s mark. Furthermore, an analysis of utility loss reveals that the cumulative negative contributions of the cost function terms Q, R, and S were reduced by 45.27%, 14%, and 25.94%, respectively. This was achieved while also reducing computation overhead by 6.5%, leading to a 72.59% increase in the overall cumulative utility.

For RQ2, our full delta-driven method (ICNN w D, in green) was compared against the baseline (ICNN w/o D, in red). As detailed in Table II, our full method achieves a 14.8% reduction in the average number of workers and a 6.4% reduction in average overtime. It also improves the average working rate by 21.8%, despite a minor 2.33% decrease in average punctuality. A detailed breakdown provides further insights into how these gains are achieved. For instance, the method dynamically adapts to the workload by reducing the prediction horizon to save computation during periods of low load (around 4000s) and increasing the weight on control cost to achieve a net gain during periods of high fluctuation (around 5000s and 6000s). Moreover, as seen around 1000s in Figure 5, the delta-driven controller actively suppresses tracking error by adjusting weights in the cost function, causing the performance curves to diverge favorably from the baseline. Ultimately, this approach reduces the cumulative negative contributions of the Q, R, and S terms by 49.74%, 58.55%, and 73.86%, respectively. While the controller introduces a slight 0.7% overhead, the MPC computation cost is reduced by 8%, resulting in a final improvement of 33.48% in total cumulative utility.

E. Discussion and Limitations

The experiments in RQ1 and RQ2 demonstrate that our method, which introduces an ICNN-based system model, more effectively captures the complex dynamics of software systems compared to traditional linear state-space models, thereby improving overall control performance. Additionally, the modeling process is fully data-driven and supports online updates at runtime, eliminating the need for manual modeling and enabling expert-free deployment. Additionally, the proposed delta-driven control mechanism allows the high-level controller manager to dynamically adjust control parameters based on the observed deviations (delta)—like decomposed utility loss, system prediction error, proportion change of computation cost to control effect. This facilitates intelligent trade-off management among conflicting objectives such as tracking accuracy and control cost. Although the auto-tuning mechanism introduces some additional overhead, its share of the total control cost remains small. Moreover, dynamic prediction horizon adjustment helps reduce computational overhead, minimizing the impact on overall cost.

This study has several limitations. First, we simplify the system's state representation by assuming the Markov property. For systems with long-term dependencies (e.g., historical accumulation or delayed effects) or partial observability, the prediction accuracy of Markov-based models may degrade. Second, the controller manager follows a reactive strategy whose responsiveness is limited by the control cycle interval. This may lead to issues such as overshooting or undershooting. Third, the controller manager adjusts MPC behavior based on negative contributions derived from utility loss. The effectiveness of this adjustment depends on the accuracy of utility loss estimation. In certain systems, utility loss may not be directly observable or may lack a linear relationship with the actual negative contribution.

V. Related Work

MPC has been widely applied to adaptive software systems, yet two core challenges remain: the modeling gap and the tuning gap.

The modeling gap arises from the difficulty of constructing accurate system models for software systems, which are often complex, dynamic, and partially observable. Traditional MPC approaches rely on linear state-space models derived from expert knowledge [18] or system identification techniques [8], [9], both of which require domain expertise and offer limited generalizability. To address this, learning-based alternatives have been proposed, such as data-driven nonlinear models [19], but they often introduce non-convex optimization problems with high computational costs. More recent works adopt ICNNs [12] to ensure global convexity and tractable optimization [20]. However, these methods typically model only the system state while retaining manual control logic, limiting their applicability in software systems with unobservable or implicit internal dynamics. To address this, we introduce ICNN to model the entire control-relevant input–output behavior of the software system, enabling a data-driven, expert-

TABLE II: STATISTICS OF RESULTS

	avg. working rate	avg. punctuality	avg. worker num.	avg. overtime rate	cum. Q-related NegCont.	cum. R-related NegCont.	cum. S-related NegCont.	cum. Comp.Cost	cum. Utility
CobRA	0.4512	0.9607	1.9320	**0.1902**	-207.31	-360.0	-126.6	512.33	2060.74
ICNN w/o D	0.5589	**0.9738**	1.3106	0.2699	-113.45	-309.6	-93.75	**475.47**	3556.75
ICNN w D	**0.6812**	0.9511	**1.1165**	0.2524	**-57.017**	**-128.3**	**-24.5**	437.47+41.35	**4747.78**

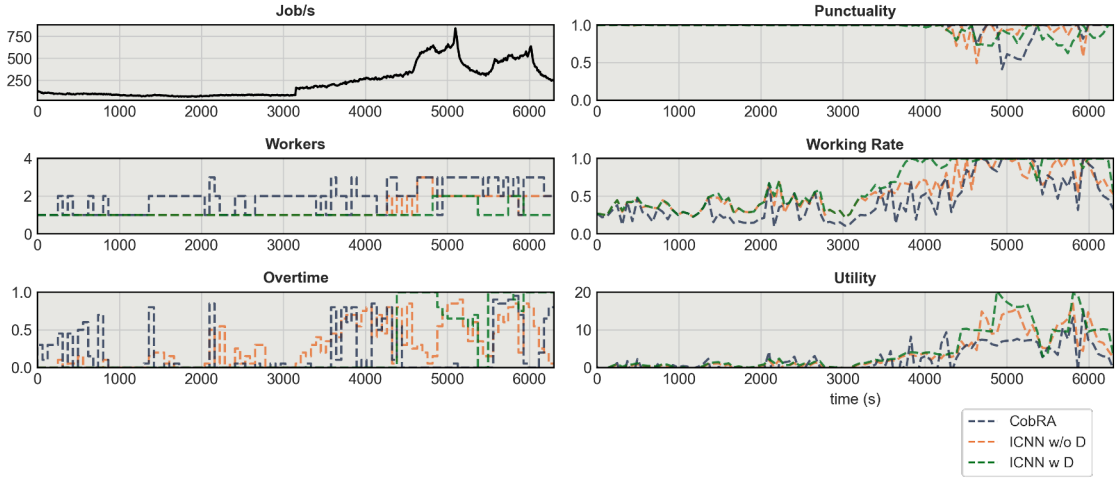

Fig. 3: Experiment Results

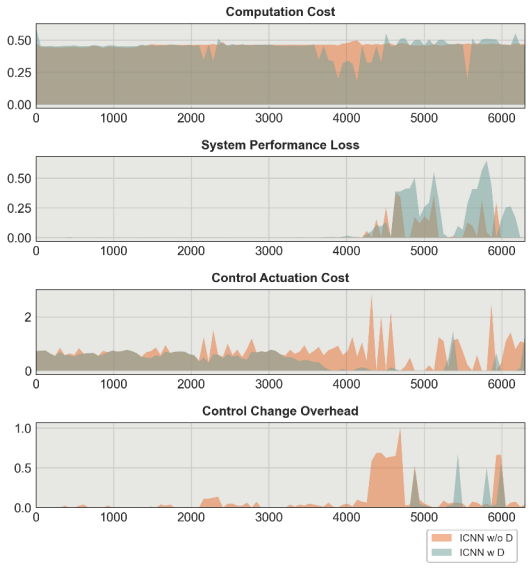

Fig. 4: Split Utility Loss

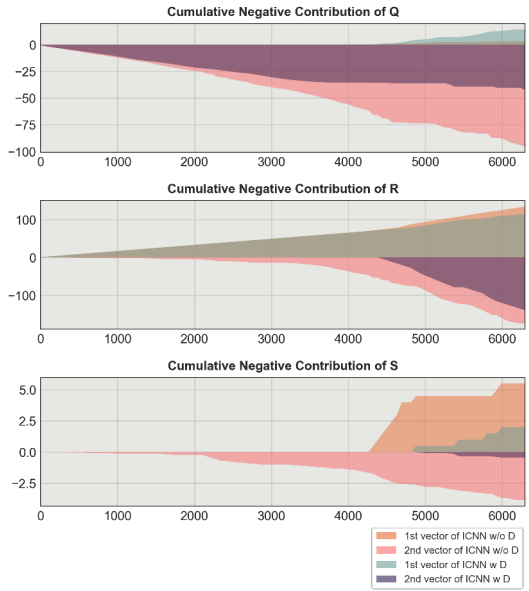

Fig. 5: Cumulative Negative Contribution

free, and globally optimizable approach suitable for complex, partially observable environments.

The tuning gap refers to the difficulty of dynamically adapting MPC parameters—such as prediction horizons and cost weights—in changing environments. Reinforcement learning (RL)-based approaches have been used to tune prediction horizons [21], [22] and cost weights [23], but they require observable states and well-defined rewards, which are often unavailable in software systems. Rule-based or trigger-based methods [24]–[26] offer another alternative by adjusting parameters based on predefined thresholds, but these rely

heavily on expert configuration and lack generality. Multi-MPC architectures [27], [28] allow for model switching and adaptive control strategies, yet require scenario-specific designs. A promising direction is hierarchical MPC, as in [29], [30], where an upper-layer controller adjusts the lower-layer controller's horizon based on survival constraints; however, the tuning objective remains limited to hardcoded rules. To bridge this gap, we propose a delta-driven controller manager that performs online adjustment of both prediction horizon and cost weights, using observed utility loss as feedback—enabling fine-grained, real-time tuning without expert-defined rules or reward functions.

VI. Conclusion and Future Work

This paper presents a dual-layer MPC framework for self-adaptive control in software systems. By incorporating ICNNs, the method simplifies the modeling process and better captures complex system dynamics compared to traditional linear models. The high-level controller manager adjusts control parameters in real time based on observed utility loss, enabling efficient adaptation while supporting online model updates. Experiments on the SIMDEX platform confirm that the proposed approach improves both modeling accuracy and control effectiveness in dynamic environments.

For future work, we plan to enhance the learning capabilities of the framework. One direction is to extend the ICNN model to an Input-Convex Recurrent Neural Network (ICRNN) to capture temporal dependencies more effectively. Another is to build a model that maps utility loss to negative contributions in the cost function, allowing for predictive tuning of control parameters.

References

[1] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Architecture-based runtime software evolution," in *Proceedings of the 20th international conference on Software engineering*. IEEE, 1998, pp. 177–186.

[2] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[3] D. Weyns, *An introduction to self-adaptive systems: A contemporary software engineering perspective*. John Wiley & Sons, 2020.

[4] B. Kouvaritakis and M. Cannon, "Model predictive control," *Switzerland: Springer International Publishing*, vol. 38, no. 13-56, p. 7, 2016.

[5] S. Kouro, P. Cortés, R. Vargas, U. Ammann, and J. Rodríguez, "Model predictive control—a simple and powerful method to control power converters," *IEEE Transactions on industrial electronics*, vol. 56, no. 6, pp. 1826–1838, 2008.

[6] A. Filieri, M. Maggio, K. Angelopoulos, N. D'ippolito, I. Gerostathopoulos, A. B. Hempel, H. Hoffmann, P. Jamshidi, E. Kalyvianaki, C. Klein *et al.*, "Control strategies for self-adaptive software systems," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 11, no. 4, pp. 1–31, 2017.

[7] K. Angelopoulos, A. V. Papadopoulos, V. E. Silva Souza, and J. Mylopoulos, "Model predictive control for software systems with cobra," in *Proceedings of the 11th international symposium on software engineering for adaptive and self-managing systems*, 2016, pp. 35–46.

[8] Z. Chen and W. Jiao, "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)*. IEEE, 2022, pp. 992–1003.

[9] I. Ayala, A. V. Papadopoulos, M. Amor, and L. Fuentes, "Prodspl: Proactive self-adaptation based on dynamic software product lines," *Journal of Systems and Software*, vol. 175, p. 110909, 2021.

[10] L. Wang, J. Xu, H. A. Duran-Limon, and M. Zhao, "Qos-driven cloud resource management through fuzzy model predictive control," in *2015 IEEE International Conference on Autonomic Computing*. IEEE, 2015, pp. 81–90.

[11] M. Schwenzer, M. Ay, T. Bergs, and D. Abel, "Review on model predictive control: An engineering perspective," *The International Journal of Advanced Manufacturing Technology*, vol. 117, no. 5, pp. 1327–1349, 2021.

[12] B. Amos, L. Xu, and J. Z. Kolter, "Input convex neural networks," in *International conference on machine learning*. PMLR, 2017, pp. 146–155.

[13] A. Simpkins, "System identification: Theory for the user, (ljung, l.; 1999)[on the shelf]," *IEEE Robotics & Automation Magazine*, vol. 19, no. 2, pp. 95–96, 2012.

[14] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM transactions on autonomous and adaptive systems (TAAS)*, vol. 4, no. 2, pp. 1–42, 2009.

[15] P. Arcaini, E. Riccobene, and P. Scandurra, "Modeling and analyzing mape-k feedback loops for self-adaptation," in *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2015, pp. 13–23.

[16] K. Ogata, *System dynamics*. © 2004, 1998, 1992, 1978 Pearson Education, Inc., 2004.

[17] M. Kruliš, T. Bureš, and P. Hnětynka, "Simdex: a simulator of a real self-adaptive job-dispatching system backend," in *Proceedings of the 17th Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2022, pp. 167–173.

[18] E. Incerto, M. Tribastone, and C. Trubiani, "Software performance self-adaptation through efficient model predictive control," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 485–496.

[19] M. A. Hosen, M. A. Hussain, and F. S. Mjalli, "Control of polystyrene batch reactors using neural network based model predictive control (nnmpc): An experimental investigation," *Control Engineering Practice*, vol. 19, no. 5, pp. 454–467, 2011.

[20] F. Bünning, A. Schalbetter, A. Aboudonia, M. H. de Badyn, P. Heer, and J. Lygeros, "Input convex neural networks for building mpc," in *Learning for dynamics and control*. PMLR, 2021, pp. 251–262.

[21] E. Bøhn, S. Gros, S. Moe, and T. A. Johansen, "Reinforcement learning of the prediction horizon in model predictive control," *IFAC-PapersOnLine*, vol. 54, no. 6, pp. 314–320, 2021.

[22] Y. Liu, S. Chen, J. Shi, and N. Zheng, "The optimal horizon model predictive control planning for autonomous vehicles in dynamic environments," in *2024 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2024, pp. 2421–2428.

[23] B. Zarrouki, V. Klös, N. Heppner, S. Schwan, R. Ritschel, and R. Voßwinkel, "Weights-varying mpc for autonomous vehicle guidance: a deep reinforcement learning approach," in *2021 European Control Conference (ECC)*. IEEE, 2021, pp. 119–125.

[24] G. Laguna, G. Mor, F. Lazzari, E. Gabaldon, A. Erfani, D. Saelens, and J. Cipriano, "Dynamic horizon selection methodology for model predictive control in buildings," *Energy Reports*, vol. 8, pp. 10 193–10 202, 2022.

[25] A. Ma, K. Liu, Q. Zhang, T. Liu, and Y. Xia, "Event-triggered distributed mpc with variable prediction horizon," *IEEE Transactions on Automatic Control*, vol. 66, no. 10, pp. 4873–4880, 2020.

[26] P.-B. Wang, X.-M. Ren, and D.-D. Zheng, "Robust nonlinear mpc with variable prediction horizon: An adaptive event-triggered approach," *IEEE Transactions on Automatic Control*, vol. 68, no. 6, pp. 3806–3813, 2022.

[27] M. Soliman, O. Malik, and D. T. Westwick, "Multiple model predictive control for wind turbines with doubly fed induction generators," *IEEE Transactions on Sustainable Energy*, vol. 2, no. 3, pp. 215–225, 2011.

[28] K. Alexis, G. Nikolakopoulos, and A. Tzes, "Switching model predictive attitude control for a quadrotor helicopter subject to atmospheric disturbances," *Control Engineering Practice*, vol. 19, no. 10, pp. 1195–1207, 2011.

[29] Z. Chen, J. Li, N. Li, W. Jiao, and E. Kang, "Context-aware proactive self-adaptation: A two-layer model predictive control approach," *ACM Transactions on Autonomous and Adaptive Systems*, 2025.

[30] Z. Chen, J. Li, N. Li, and W. Jiao, "Reliable proactive adaptation via prediction fusion and extended stochastic model predictive control," *Journal of Systems and Software*, vol. 217, p. 112166, 2024.