

Runtime Self-Optimization Through Knowledge-Driven Design Space Reduction

Luca Brodo
luca.brodo@hshl.de
Hochschule Hamm-Lippstadt
Lippstadt, NRW, Germany

Giuseppe Scalora
giuseppe.scalora@hshl.de
Hochschule Hamm-Lippstadt
Lippstadt, NRW, Germany

Stefan Henkler
stefan.henkler@hshl.de
Hochschule Hamm-Lippstadt
Lippstadt, NRW, Germany

ABSTRACT

Modern Cyber-Physical Systems must operate in volatile environments, with conditions that are often unpredictable at design time, requiring engineering solutions that are able to learn and optimize their configuration at runtime. However, the dimensions of modern systems, which are often complex and highly interconnected, make self-optimization at runtime a challenging task. In this paper, we propose a novel approach to runtime self-optimization that leverages knowledge-driven design space reduction to enable efficient and formally correct optimization of complex systems. Contrary to naïve re-optimization strategies, which are often prohibitive, or heuristic search methods, which sacrifice optimality, our approach guarantees optimal solutions with heuristics-level runtime performance. Specifically, by storing and reusing proven-optimal configurations and intelligently pruning the design space based on learned knowledge, we are able to significantly reduce the search space, hereby enabling fast re-optimization. Upon changes in the operational domain, the knowledge base is efficiently queried and updated, with regions expansion and merging to maintain compactness. The key idea behind the approach is to use Zonotopes as the underlying representation, which offer a compact and efficient way to represent and manipulate high-dimensional design spaces. We evaluate the approach on a smart-farming drone scenario, comparing against common optimization methods demonstrating that our approach not only preserves optimality, but also improves performance up to ~56x compared to baseline methods with negligible memory and computation overhead.

CCS CONCEPTS

• **Software and its engineering** → **Search-based software engineering**.

KEYWORDS

Cyber-physical Systems, Knowledge-Driven Self-optimization

ACM Reference Format:

Luca Brodo, Giuseppe Scalora, and Stefan Henkler. 2026. Runtime Self-Optimization Through Knowledge-Driven Design Space Reduction. In *21st International Conference on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '26)*, April 13–14, 2026, Rio de Janeiro, Brazil. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3788550.3794861>

1 INTRODUCTION

Modern Cyber-Physical Systems (CPS), such as autonomous vehicles and drones, exemplify the challenges of engineering software systems that must operate safely and efficiently in dynamic and often uncertain environments. These systems are typically configured at design time for a specific *operational domain*, which defines the environmental, contextual, and task conditions under which they are expected to perform optimally. However, although these operational domains are carefully specified and tested at design time, real-world deployments inevitably expose these systems to conditions that deviate from the ones they were originally designed for. When these shifts in conditions happen—for example due to weather, new regulatory requirements, or resource aging—configurations that were optimal at deployment time may become suboptimal, potentially hindering the quality of service of the system.

To maintain optimal performance in the face of such changes, CPS must be endowed with the ability to *self-optimize* at runtime. This transforms them into self-optimizing systems, a specialized class of Self-Adaptive Systems (SAS) that not only maintain correct operation but also continuously seek the best possible configuration given the current context. The SAS community has long advocated for feedback-loop architectures, such as the *MAPE-K* pattern [20] or the OCM architecture [18, 19], that can give the ability to maintain desirable system properties by continuously monitoring context, analyzing trade-offs, planning adaptations, and executing them over a shared knowledge base. What remain challenging in practice is to provide the system with a mechanism to efficiently explore the design space at runtime to identify and deploy the optimal configuration according to the current operational context.

Existing optimization-based approaches must face a trade-off between optimality and runtime performance, rendering their applicability to runtime adaptation limited. The vast and multidimensional design space of modern SAS leads to combinatorial explosion, making exhaustive exploration infeasible [30]. Heuristic methods improve performance but at the cost of formal guarantees, often converging to local optima [5, 17]. Moreover, precomputed solutions lack robustness to dynamic operational design domain shifts—such as sudden weather changes or new regulatory constraints—rendering static configurations ineffective [10]. Consequently, a robust method for exact self-optimization at runtime is still missing.

To bridge this gap, we propose a knowledge-driven approach for runtime self-optimization which guarantees the optimality of exact methods while providing runtime performance rivaling the one of heuristics methods. The novelty of the approach is to leverage previously acquired knowledge to intelligently and correctly prune



the design space based on the operating conditions, redirecting the optimization efforts only on never-before explored regions, thus improving performance without sacrificing correctness. The key idea is to use zonotopes as underline representation for the knowledge, ensuring memory compactness and efficient merging and retrieval operations, leading to insignificant overhead for memory management as confirmed by our evaluation.

This work is structured as follows. Section 2 introduces a motivating example. Section 3 reviews related work. Section 4 presents our approach in detail, while Section 5 discusses implementation aspects. Finally, Section 6 evaluates our method and Section 7 concludes the paper and outlines future research directions.

2 MOTIVATING EXAMPLE: SELF-OPTIMIZE AUTONOMOUS DRONES

As part of a multi-year national project on sustainable autonomous farming, agricultural automation systems have been developed to optimize pesticide use, crop monitoring, and soil analysis. Specifically, fleets of autonomous drones are employed to perform tasks such as aerial imaging, targeted spraying, and environmental sensing in highly dynamic outdoor environments. A key challenge is to find optimal configurations for operational parameters—such as flight path, speed, and sensor settings—while adhering to a complex and evolving set of constraints.

Traditionally, when the weather or regulations change, each drone would need to re-optimize its configuration—adjusting parameters such as flight altitude, speed, spray rate, or sensor resolution—to maintain safety and performance. However, such full re-optimization at runtime is computationally expensive and inefficient, especially when similar conditions have already been encountered by other drones.

Instead, imagine that a drone operating in one region encounters a particular weather pattern—say, strong crosswinds and high humidity—and identifies the optimal configuration to maintain stability and minimize energy use. Later, when another drone in a different region experiences the same conditions, it should not have to start the optimization process from scratch. By leveraging the knowledge previously learned—stored as a mapping between environmental conditions and optimal configurations—the second drone can directly reuse this known optimal setup. Furthermore, if the new conditions are comparable but not identical, the design space can be intelligently pruned to focus only on the unexplored regions, significantly reducing the search effort required to find the new optimal configuration. This knowledge-driven reduction of the search space eliminates redundant computation, enabling rapid adaptation while preserving optimality.

This example illustrates the core motivation of our work: learning from past optimization experiences to enable knowledge-driven adaptation. By remembering and reusing proven-optimal configurations across drones and environmental contexts, the system reduces the design space it must explore at runtime. Rather than re-optimizing globally, it focuses computational effort only where new or previously unseen conditions occur—achieving faster adaptation without compromising optimality.

3 RELATED WORK

Research on self-adaptive systems (SAS) has long organized runtime adaptation around feedback-loop architectures, most prominently IBM's MAPE-K model [20, 22] and the OCM architecture [18, 19], both providing a structured approach for monitoring, analysis, planning activities, while emphasizing assurance, evidence, and traceability at runtime [9, 37]. In this tradition, many SAS employ heuristic policies or surrogate-driven search in the Analyze/Plan stages, yet the literature repeatedly highlights the difficulty of delivering foarmal guarantees at runtime [10].

A significant body of research has addressed the challenge of constrained optimization by pre-computing optimal policies offline through multi-parametric programming, most notably in the development of explicit Model Predictive Control (eMPC). A notable example is the work of Bemporat et al. [3], which laid the foundation for eMPC. Their paradigm solves a multi-parametric quadratic or linear program (mp-QP/mp-LP) at design time [3]. This process partitions the system's state space into a set of distinct polyhedral regions, each associated with a pre-calculated, optimal PieceWise Affine (PWA) control law [34]. The primary advantage of eMPC is its extremely low online computational footprint; at runtime, the control action is determined by a simple lookup to identify the current region and evaluate the corresponding affine function, making it highly suitable for resource-constrained systems with fast dynamics. However, this approach fundamentally relies on a static problem formulation, assuming that the system model, constraints, and objective function are fixed. Consequently, these pre-computed solution maps are brittle and cannot adapt to unforeseen runtime changes in the operational context—such as the shifting environmental or regulatory constraints defined by an Operational Design Domain (ODD). Furthermore, the complexity of the explicit solution often grows exponentially with the problem dimension, limiting its applicability. In stark contrast, our work does not pre-compute a global, static solution map but instead performs optimization *at runtime*, directly addressing the dynamism of the ODD. We achieve efficiency not through exhaustive pre-computation, but through a knowledge-driven mechanism that caches and reuses optimal solutions, enabling rapid, guaranteed-optimal adaptation to problems whose definitions evolve during operation.

Within the broader umbrella of runtime decision-making, *Search-Based Software Engineering (SBSE)* has been extensively explored for adaptation planning, using evolutionary algorithms, utility maximization, and reinforcement learning to navigate large configuration spaces online [8, 16]. These approaches are effective when objective landscapes are highly nonlinear and model knowledge is scarce, but they typically provide probabilistic or empirical assurances rather than hard guarantees, and seldom yield artifacts directly consumable by engineering governance. *Bayesian Optimization (BO)* has also been applied to online tuning, recent state-of-the-art methods such as SafeOpt and time-varying safe BO impose probabilistic safety while tracking drifting optima [4, 23, 33]. Despite strong sample efficiency, recent analyses question the transfer of theoretical safety assumptions to practice and highlight sensitivity to prior misspecification and the runtime overhead of posterior updates [39]. As a result, while SBSE and BO provide powerful mechanisms, their stochastic exploration complicates worst-case

timing guarantees and certification. Another line of work addresses the computational burden of online optimization by reusing information from previous solves, a technique known as warm-starting or re-optimization, particularly in the context of hybrid Model Predictive Control (MPC).

In this domain, a complex Mixed-Integer Program (MIP) must be solved at each control step. The work by Marcucci and Tedrake [25] presents a state-of-the-art method where, instead of solving each MIP from scratch, the entire Branch and Bound (B&B) search frontier from the previous time step is shifted forward in time and used to initialize the new optimization. By propagating dual variable information, this approach effectively generates tight initial bounds, drastically pruning the search tree for problems that are structurally similar across consecutive time steps. While highly effective for the incremental nature of receding-horizon control, such techniques are tightly coupled to the internal state of a specific solver (e.g., B&B) and are designed to reuse information from the immediately preceding problem instance [13]. Our approach, in contrast, operates at a higher level of abstraction. Instead of reusing a low-level solver frontier, we build a persistent, context-aware knowledge base of guaranteed-optimal configurations. This allows our system to learn over its lifetime and respond efficiently to discrete, significant shifts in the ODD, which may render the immediately preceding solution completely irrelevant.

Concurrently, the notion of an *ODD* has been formalized in standards for automated and autonomous systems (e.g., PAS 1883) and complemented by machine-readable schemas (ASAM OpenODD) that specify environmental, contextual, and task envelopes [2, 7]. These standards enable a principled articulation of when a function may operate, but typically stop short of prescribing optimization procedures that exploit ODD constraints at runtime. Several recent efforts have therefore focused on reducing or *pruning the runtime design space* to accelerate adaptation and improve assurance.

Constraint-based approaches exploit formal requirement models to eliminate infeasible configurations at runtime. For example, Neema et al. [27] introduced constraint-guided adaptation, where design-time constraints are combined with runtime parameters to shrink the operational search space. Similarly, monotonicity-based performance models allow discarding dominated configurations without explicit evaluation [35]. The Thallium framework [32] employs probabilistic model checking to compute best- and worst-case utility bounds for configurations, pruning transitions that cannot yield Pareto-optimal outcomes.

A complementary line leverages *machine learning* to approximate the performance or feasibility of configurations and prune unpromising options. Quin et al.'s ML2ASR [29] applies supervised learning to predict which adaptation choices are likely to satisfy goals, while DLASer+ [38] uses online deep learning to continuously refine pruning decisions without handcrafted features. Such methods achieve significant space reductions with low overhead, though their guarantees remain statistical. Finally, *evolutionary and search-based heuristics* inherently prune the design space through iterative selection. Frameworks such as PLATO [30] employ genetic algorithms to discard poor-performing candidates generation by generation, trading strict optimality for anytime decision-making under resource constraints.

To provide stronger runtime guarantees, our approach differs from the above by elevating the ODD from documentation to a live, machine-checkable feasible set that confines adaptation decisions; instantiating Analyze/Plan as a certifiable robust (MI)LP optimization over the moving intersection of design space and ODD with hard feasibility guarantees; and integrating solver artifacts (duals, infeasibility certificates) and ODD deltas as runtime evidence. In contrast to prior work that favors probabilistic safety or heuristic pruning, our method closes an assurance gap by delivering deterministic timing, zero-violation guarantees under calibrated uncertainty sets, and audit-ready artifacts, while still enabling data-driven refinement.

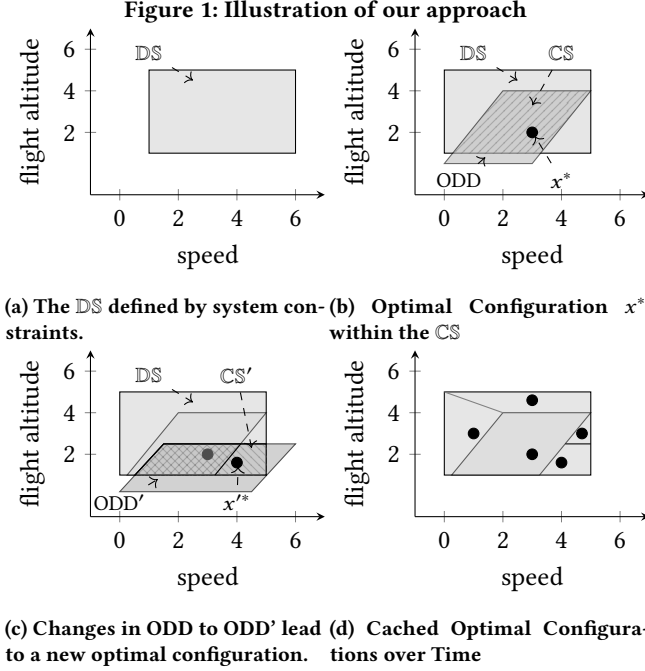
4 APPROACH

In this section, we present our approach for self-optimization. Our method is grounded in established software engineering paradigms for self-optimizing systems, leveraging a Digital Twin (DT) architecture and the MAPE-K loop to enable efficient online optimization.

4.1 Preliminaries

We introduce three key concepts that underpin our approach: the Design Space, the Operational Design Domain, and the Configuration Space.

- (1) *Design Space (DS)*: This represents the entire set of operational parameters and behaviors of a system, defined by its intrinsic physical and software limits. The *DS* is static and determined at design time. It represents everything the drone *can* do. Aligning with the definition in [10], the *DS* can be described as a Cartesian space with dimensions representing design decisions and values along those dimensions representing the possible choices. Points in the space represent concrete designs, which we refer to as *configurations*. Considering, for example, our smart farming drone, the *DS* encompasses all possible combinations of flight altitude (e.g., 1 to 5 meters) and speed (e.g., 0 to 5 m/s), as in Figure 1a.
- (2) *Operational Design Domain (ODD)*: The ODD defines the specific environmental, contextual, and task conditions under which the system is currently permitted to operate. Unlike the *DS*, the ODD is dynamic. It includes factors like weather conditions (e.g., wind speed is 10 m/s), regulatory constraints (e.g., stay clear of a no-fly zone), and resource status (e.g., battery level above 20%). This represents the conditions the system *operates within* at any given time. In Figure 1b, the ODD is illustrated as a polygon that intersects with the *DS*, highlighting the current operational constraints.
- (3) *Configuration Space (CS)*: As indicated by [10], in practice, at runtime, we solely focus on the part of the *DS* which contains configurations that are significant for the task at hand, which we refer to as *feasible*. We capture this intuition by indicating this portion of the *DS* as *CS* and formulating it as the intersection $CS := DS \cap ODD$. The *CS* represents the subset of configurations that *are feasible* (within the *DS*) and *optimal* (according to current ODD). This is the space within which the system can safely and effectively operate at any given time. For instance, while the drone's *DS* allows it to fly at 5 m/s, the current ODD (strong winds) might restrict



the CS to speeds below 2 m/s to ensure stability. If $CS = \emptyset$, it signifies that no feasible configuration exists for the current ODD given the system's capabilities.

4.2 Knowledge-Driven Runtime Self-Optimization

The goal of self-optimization is to find the configuration $\vec{x}$ within the current CS that optimizes a given performance metric, such as minimizing energy consumption or maximizing area surveyed. We denote this performance metric as a function f over the current ODD and CS. This is often formulated as an optimization problem of the form:

$$\begin{aligned} \min \quad & f(\vec{x} \mid ODD) \\ \text{s.t.} \quad & \vec{x} \in CS \end{aligned} \quad (1)$$

The solution, $\vec{x}_0^*$ represents the optimal configuration for the initial deployment conditions (Figure 1b). At design-time, engineers define the DS based on system capabilities and constraints, and establish the *initial* ODD, i.e. the condition the system is *expected* to operate under. An optimal configuration is computed by solving the optimization problem (1) according to the specified DS and ODD. The resulting optimal configuration is then deployed to the system, ensuring that the it starts in a state that is both feasible and optimized for its initial operating conditions.

The central challenge for the deployment of CPS lies in the fact that the ODD is not static, but rather, it is highly dynamic and can change frequently based on a variety of factors. The expected ODD conceptualized at design time rarely fully capture the range of conditions the system will encounter in real-world operation. Furthermore, the ODD are often results of regulations and policies that can change over time. For instance, in our smart farming drone

example, the ODD may change due to environmental factors (e.g., sudden rain, temperature fluctuations), as well as due to the introduction of new regulations (e.g., new no-fly zones) or operational demands (e.g., urgent need to survey a specific area). The main result of these changes can significantly alter the CS, potentially rendering the previously optimal configuration, $\vec{x}_0^*$, suboptimal or even infeasible. Figure 1c illustrates such a scenario, where the ODD changes due to a new regulation which requires lower flight altitudes, hereby shifting the CS to a new region. In this case, the system must re-optimize at *runtime* to find a new optimal configuration, $\vec{x}_1^*$, that is feasible within the new CS as well as optimal for the current ODD.

After new ODD are observed and the CS is updated, a naive approach would be to re-solve (1) over the new CS_i . However, this could be computationally expensive and prohibitive, especially for resource-constrained devices. Our approach improves this process by using a *Knowledge* base to guide re-optimization efficiently through *Knowledge-Driven Re-Optimization*. Intuitively, if a previous optimal configuration $\vec{x}_{i-1}^*$ remains within the new CS_i , we can drastically prune the search. As illustrated by the checkered region in Figure 1c, any configuration previously found to be suboptimal to $\vec{x}_{i-1}^*$ remains so if it is still feasible. The search can therefore focus on the unexplored regions of the new CS_i , or conclude that $\vec{x}_{i-1}^*$ is still optimal if no such regions exist. In essence, the system avoids re-evaluating configurations that are known to be poor performers. If $\vec{x}_{i-1}^*$ becomes infeasible, the system must find a new optimum. Instead of a full exploration, we leverage the Knowledge base: previously explored configurations and their performance are used to “warm-start” the optimization solver, accelerating convergence to a new optimal solution.

This iterative ODD-driven optimization process enables continuous improvement and full utilization of the design space. As illustrated in Figure 1d, the system progressively explores different ODDs and learns a map of optimal configurations across the entire DS. With each new ODD encountered, accumulated knowledge allows more efficient pruning of the search space, leading to compounding performance gains over time.

5 IMPLEMENTATION

The *knowledge-driven re-optimization* approach introduced in Section 4.2 imposes stringent requirements on the underlying knowledge representation and requires an architecture that can effectively and correctly manage and utilize this knowledge. Section 5.1 describes the architectural foundation of our implementation, which is based on a Digital Twin (DT) framework that employs the MAPE-K loop to structure the optimization process and manage the knowledge base.

For knowledge to be efficiently saved and retrieved at runtime, as well as shared, merged, and reasoned about coherently across distributed systems, it cannot be a mere collection of ad-hoc data points; it must be captured in a formal, unambiguous structure. Moreover, this representation must support efficient operations for querying and updating the knowledge base, such as determining unexplored regions of the design space, extending existing knowledge entries, and merging overlapping or adjacent entries to minimize fragmentation. Section 5.2 details our implementation of such a

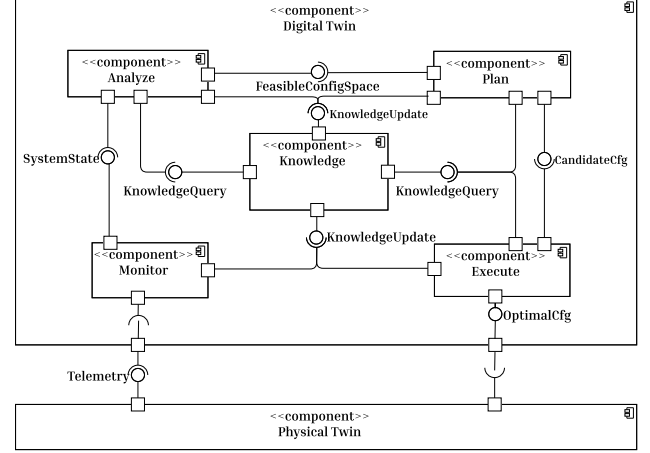
knowledge base and the rationale behind choosing *zonotopes* as our representation for regions of the $\mathbb{CS}$. In essence, zonotopes strike a compelling balance between expressiveness and computational efficiency, enabling both a compact and continuous representation of complex feasible regions as well as efficient computation of key set operations needed for our caching strategy. The procedure for knowledge-driven re-optimization is then detailed in Algorithm 1, which leverages the zonotope representation to avoid redundant computation by storing the results of previous optimization queries, mapping regions of the design space to their optimal configurations. Its novelty lies in its ability to not only serve exact hits but also to intelligently prune the search space for new queries, extending cached regions to accelerate convergence. Finally, Sections 5.4 and 5.5 describe the system and environmental models used in Section 6 to evaluate our approach.

5.1 Architectural Foundation

As illustrated in Figure 2, our architecture is based on the interplay between a Physical Twin (PT)—the operational system—and its virtual counterpart, the DT. The DT structures the adaptation process using the MAPE-K loop, hereby acting as *managing systems*, providing a systematic framework for monitoring, analysis, planning, and execution, all supported by a shared Knowledge base. The PT, on the other hand, serves as the *managed system*, executing the configurations determined by the DT. This separation of concerns allows for a clear delineation between the system’s operational aspects and its adaptive capabilities, facilitating more effective management and optimization. More specifically, we separate the real-time critical operations, which are executed by the PT, from the more computationally intensive tasks of self-optimization and learning, which are handled by the DT. We maintain a continuous feedback loop between the PT and DT, where the PT provides information about its status and the environment to the DT (the *Monitor* stage), and the DT, in turn, computes optimal configurations that are then provided to the PT (the *Execute* stage). The flexibility provided by the DT allows to consider not only data provided by the PT, but also external data sources (e.g., weather forecasts, regulatory databases) as well as human-in-the-loop inputs in case of regulation changes. The computation of the optimal configurations occurs at the *Analyze* and *Plan* stages within the DT. After observing a shift in environmental conditions, this data is analyzed to assess its influence on the current configuration and, if required, an optimization problem is formulated and solved at the *Plan* stage to identify a new optimal configuration. This optimization is informed by the *Knowledge* base, which accumulates previous configurations, enabling more efficient and effective planning over time (c.f. Figure1). The resulting optimal configuration is then communicated back to the PT for execution, completing the adaptation cycle. The PT stores these configurations locally to ensuring that the deployment of new configurations can occur within hard deadlines, effectively decoupling real-time operations from the DT’s optimization processes.

This design is particularly advantageous for CPS operating in dynamic environments, such as our smart farming drone example, where the DT can leverage its computational resources to perform complex analyses and optimizations without impeding the PT’s ability to respond safely and in real-time to environmental changes.

Figure 2: Architecture of a PT and DT with MAPE-K loop.



Furthermore, the learning process is further enhanced by the DT’s ability to aggregate data from multiple sources. This key capability elevates the learning process beyond individual systems, enabling collective intelligence across a fleet of CPS. The DT acts as a gateway, allowing individual systems to contribute their locally validated findings to a shared, global knowledge base. The insights gained from various ODDs encountered by different systems are shared and merged, creating a *cumulative memory*. This collective experience can then be synchronized back to the entire fleet, ensuring the operational knowledge of one system benefits all others. For instance, a newly deployed drone can be immediately bootstrapped with the hard-won experience of its peers, avoiding costly re-discovery of optimal behaviors.

5.2 Zonotope-Based Cache for Online Knowledge-Driven Design Space Pruning

To realize the efficient, knowledge-driven re-optimization described in Section 4.2, the *Knowledge* base of our MAPE-K loop is implemented as a *cache*. This knowledge base avoids redundant computation by storing the results of previous optimization queries, mapping regions of the design space to their optimal configurations. Its novelty lies in its ability to not only serve exact hits but also to intelligently prune the search space for new queries, extending cached regions to accelerate convergence.

As underlying representation for our knowledge we use *zonotope* [1, 31].

Zonotopes can exactly represent any N-dimensional hyperrectangle, which results from parameter bounds of the form $a_i \leq x_i \leq b_i$, however they can also represent more general convex shapes that may arise from intersecting multiple of these constraints. This representation is particularly advantageous in our settings, where the ODD and $\mathbb{CS}$ are typically defined by a set of (mixed-integer) linear constraints of the same form due to their tractability, as exemplified by [6, 12, 14, 21, 32, 36]. Zonotopes can compactly and continuously represent such multi-dimensional regions as well as the result of their intersections or unions, which usually yield complex feasible

regions. Compared to simpler shapes like hyperrectangles, the flexibility of zonotopes helps avoid excessive memory fragmentation in case of these complex regions. For example, if the intersection of the $\mathbb{DS}$ with the ODD results in a t-shaped feasible region, a zonotope can represent this shape more compactly than a collection of hyperrectangles, which would require multiple entries. Furthermore, zonotopes support efficient computation of key set operations needed for our caching strategy: the set difference $(A \setminus B)$ and the convex hull of the union $(A \cup B)$ can be computed efficiently, allowing us to determine unexplored regions and merge existing entries.

The core of our approach is the QueryAndOptimize procedure, outlined in Algorithm 1. This procedure is run whenever the system needs re-optimization from a shift in the ODD. The procedure starts by computing the current $\mathbb{CS}$ as the intersection of the $\mathbb{DS}$ with the constraints derived from the current ODD (line 1). Note that at this step a function $\Gamma : \mathbb{R}^n \rightarrow \mathbb{R}^m$ is used to map the ODD into the same constraint space as the $\mathbb{DS}$. The function Γ shall represent the domain knowledge of how environmental conditions impact the feasible configurations of the drone fleet. For instance, if the wind speed exceeds a certain threshold, Γ would impose a meaningful constraint on the maximum allowable speed and altitude in the $\mathbb{DS}$ to ensure safe navigation for the drones. Similarly, low visibility conditions would lead to a minimal camera resolution requirement to ensure adequate imaging capabilities.

The memory C is then queried to identify any saved entry that intersect with the current $\mathbb{CS}$ and share *comparable* ODD (line 2). In this context, ODD comparability is defined as two ODDs that will result in the same set of constraints when mapped to the design space through the function Γ .

This is efficiently checked by ensuring that either $e \subseteq \Gamma(ODD)$ or $\Gamma(ODD) \subseteq e$.

The rationale behind this condition is discussed in detail in Section 5.3. The union of these regions represents the already explored $\mathbb{CS}$, or $\mathbb{CS}_c$, and represents the portion of the $\mathbb{DS}$ that has already been explored (line 3). The unexplored region, $\mathbb{CS}_u$, is then computed as the set difference between the current $\mathbb{CS}$ and the region stored in memory (line 4). If the unexplored region is empty (line 5), it indicates a cache hit, and the best solution from the cached regions is returned (line 6). If there are unexplored regions, a solver to find the optimal solution is invoked only on this subset (line 9). If there are cached regions, the best solution among them is compared to the new solution (lines 10-15). If the new solution's objective value is within a specified arbitrary tolerance (τ_{ext}) of the cached best, the region saved in memory is extended to include the new design space, and the previous solution is returned (lines 12-14). Otherwise, a new entry is created for the unexplored region with its corresponding optimal configuration and added to the memory (lines 17-18). Before new regions are added, lines 19-22 check for potential merges with existing entries; if suitable candidates are found, these are tried to be merged with existing entries to reduce fragmentation and improve future query efficiency. This merging can be done only if two regions are ODD-comparable and their geometric union does not introduce excessive over-approximation. Upon a successful merge, their zonotopes are unified, and the superior of the two optimal configurations (i.e., the one with the minimum objective value) is retained as the representative for the newly formed

Algorithm 1: QueryAndOptimize

Input: ODD, Memory C
Output: Optimal configuration x^* , objective value $f(x^*)$

```

1  $\mathbb{CS} \leftarrow \mathbb{DS} \cap \Gamma(ODD)$ ;
2  $\mathcal{R} \leftarrow \{e \in C \mid e \cap \mathbb{CS} \neq \emptyset \wedge \text{COMPARABLE}(\Gamma(ODD), e)\}$ ;
3  $\mathbb{CS}_c \leftarrow \bigcup_{e \in \mathcal{R}} e$ ;
4  $\mathbb{CS}_u \leftarrow \mathbb{CS} \setminus \mathbb{CS}_c$ ;
5 if  $\mathbb{CS}_u = \emptyset$  then
6    $(x^*, f(x^*)) \leftarrow \text{BESTSOLUTION}(\mathcal{R})$ ;           // Cache Hit
7   return  $x^*, f(x^*)$ ;
8  $(x_{\text{new}}^*, f(x_{\text{new}}^*)) \leftarrow \text{SOLVE}(\mathbb{CS}_u)$ ; // Optimize only on
   the unexplored space
9 if  $\mathcal{R} \neq \emptyset$  then
10   $(x_c^*, f(x_c^*)) \leftarrow \text{FINDBESTSOLUTION}(\mathcal{R})$ ;
11  if  $f(x_{\text{new}}^*) \geq f(x_c^*) \cdot (1 - \tau_{\text{ext}})$  then
12     $\text{EXTEND}(C, \mathcal{R}, x_c^*)$ ;
13    return  $x_c^*, f(x_c^*)$ ;
14  $e_{\text{new}} \leftarrow \text{CREATE}(\mathbb{CS}_u, x_{\text{new}}^*, f(x_{\text{new}}^*), \Gamma(ODD))$ ;
15  $e \leftarrow \text{FINDMERGECANDIDATES}(C, e_{\text{new}})$ ;
16 if  $e \neq \emptyset$  then
17    $(x_{\text{new}}^*, f(x_{\text{new}}^*)) \leftarrow \text{MERGE}(e, e_{\text{new}})$ ;
18 else
19    $\text{ADD}(C, e_{\text{new}})$ ;
20 return  $x_{\text{new}}^*, f(x_{\text{new}}^*)$ ;

```

entry. This approach ensures that the memory remains both relevant and efficient, adapting to new information while leveraging past computations to minimize redundant optimization efforts.

5.3 Discussion

Some comments are in order. Firstly, the use of an arbitrary tolerance parameter τ_{ext} in line 10 allows for a controlled trade-off between strict optimality and cache efficiency. By permitting slight deviations (e.g. $\tau_{\text{ext}} \ll 1\%$) from the optimal objective value, the system can reduce the frequency of cache misses and the associated computational overhead, while still maintaining a high level of performance. The choice is motivated by the observation that in many practical scenarios, especially in CPS applications, the cost of re-optimization can be significant compared to the marginal gains from finding a negligibly better solution.

Secondly, the consolidation of cache entries (not shown in the pseudocode for brevity) is performed as a background operation to minimize fragmentation and improve cache hit rates. This involves checking for adjacent or overlapping regions in the cache and combining them into a single entry, thereby reducing fragmentation.

Thirdly, similar to any context-sensitive mechanism, a potential concern with our approach is *thrashing* [11], i.e. excessive re-optimization triggered by minor, high-frequency fluctuations in the ODD. Since the cache is indexed by both geometric overlap and ODD comparability (line 2), frequent and minor changes in the ODD can lead to a high rate of cache misses, as the system

may fail to find suitable cached entries even if the geometric design space has not changed significantly. While standard control systems techniques of *hysteresis* and *dwell-time logic* can be implemented in the *Analyze* stage to prevent re-planning unless an ODD parameter changes by a significant threshold or persists in a new state for a minimum duration, these techniques alone may lead to suboptimal performance. Furthermore, the removal of the semantic condition on the ODD, while potentially reducing thrashing, would compromise the correctness of the solutions served by the memory. Without this condition, the cache becomes context-agnostic, treating all cached regions as if they were derived from the same underlying problem and the algorithm will make invalid assumptions based purely on geometric overlaps. This would lead to *incorrect hits*. For example, a drone's optimal configuration (high speed, high altitude) might be found within a large $\mathbb{CS}_{\text{daylight}}$. If the ODD shifts to night-time, the resulting $\mathbb{CS}_{\text{night}}$ may be a geometric subset of $\mathbb{CS}_{\text{daylight}}$, but the high-speed daylight optimum might be sub-optimal during night-time.

Our approach, addresses the problem of thrashing not by weakening the cache's correctness guarantees, but rather by managing the re-optimization trigger at the MAPE-K level. For this we consider encoding the environmental conditions into an abstraction function Γ . The *ODD abstraction function* Γ can be employed to map continuous environmental parameters to discrete, meaningful regimes (e.g., raw wind speed is classified into "Calm", "Light", or "Strong"). This is motivated by the understanding that minor fluctuations in environmental conditions often do not necessitate a change in the drone's configuration. For example, a slight increase in wind speed from 5 to 5.3 mph may not require a different flight strategy and can be effectively treated as the same regime. By grouping similar conditions into regimes, we can reduce the frequency of re-optimization and make the system more robust to minor fluctuations in the environment. The system then detects the ODDs overlaps rather than the raw value, ensuring stability within a regime and triggering re-optimization only upon a significant state transition or threshold break. This approach provides practical robustness against environmental noise without corrupting the cache's logic or compromising its ability to guarantee optimality for a given operational context.

5.4 System Model and Problem Formulation

Our system model is centered around a fleet of autonomous drones tasked with an agricultural mission. In this case, we focus on spraying specific pesticide formulations across a field. The field is divided into a set of discrete tasks, where each task corresponds to covering a single row.

The $\mathbb{DS}$ of the drone is the tuple $\mathbf{x}_e = (s, a, c, sr, P, ssr)$, where s is the speed, a is the max allowed altitude, c is the camera resolution, sr is the spray rate, P is the power factor limit, and ssr is the sensor sampling rate. Each of these parameters has specific bounds and constraints based on the drone's physical capabilities and mission requirements. For instance, speed and altitude are constrained by safety regulations and environmental conditions, while battery capacity and payload weight are limited by the drone's design specifications. A *fleet* consists of M equal drones.

To solve the mission planning problem, we formulate it as a multi-agent task assignment problem aimed at minimizing total energy consumption, while adhering to battery constraints and task assignment constraints (field coverage in our case). The goal is to find a single, globally optimal configuration $\mathbf{x}^*$ for the entire fleet and an assignment of tasks to drones that completes the mission with the least amount of energy. Since the energy model is non-linear, we employ a linear approximation to make the problem tractable for a Mixed-Integer Linear Program (MILP) solver.

The configuration parameters $\mathbf{x}$ are treated as continuous decision variables within the bounds defined by the current design space $\mathbb{CS}_f$. The total energy cost $E_{\text{total}}(\mathbf{x})$ is the sum of the energy to complete each of the N tasks with configuration $\mathbf{x}$:

$$E_{\text{total}}(\mathbf{x}) = \sum_{j=1}^N E(\mathbf{x}, \text{task}_j) \quad (2)$$

where $E(\mathbf{x}, \text{task}_j)$ is the non-linear energy cost function, growing quadratically with respect to the speed. We linearize this function using a first-order Taylor expansion around a reference point $\mathbf{x}_{\text{ref}}$ (e.g., the center of the design space):

$$E(\mathbf{x}, \text{task}_j) \approx E(\mathbf{x}_{\text{ref}}, \text{task}_j) + \nabla E(\mathbf{x}_{\text{ref}}, \text{task}_j)^T (\mathbf{x} - \mathbf{x}_{\text{ref}}) \quad (3)$$

The gradient ∇E is computed numerically using finite differences around the reference point $\mathbf{x}_{\text{ref}}$, which is the center of the design space. This results in a linear objective function. We define two decision variables for the MILP: a continuous vector $\mathbf{x}$ representing the drone configuration and a binary matrix z where $z_{ij} = 1$ if drone i is assigned to task j , and 0 otherwise. The latter is used to ensure that each task is assigned to exactly one drone, which is expressed as

$$\sum_{i=1}^M z_{ij} = 1 \quad \forall j \in \{1, \dots, N\} \quad (4)$$

We subject the problem to an additional battery constraint, which can be optionally triggered through the implementation, the constraint basically constrains the energy consumption within battery capacity limits defined as:

$$\sum_{j=1}^N E(\mathbf{x}, \text{task}_j) \leq \text{usableWh} \quad (5)$$

If not triggered, the simulation will simply just opt to abstract away the battery constraints assuming the power drawing alternates seamlessly among the available replacement batteries on board.

The minimal configuration $\mathbf{x}^*$ is then found by solving the following optimization problem at each re-optimization step i :

$$\begin{aligned} \min \quad & (3) \\ \text{s.t.} \quad & \vec{\mathbf{x}} \in \mathbb{CS}_{f,i} \text{ and } (4), (5) \end{aligned} \quad (6)$$

5.5 Environmental Model

The environmental model captures the dynamic characteristics of the ODD that influence the system's performance. We provide a realistic representation of the environment where we generate a dynamic weather sequence over the simulation period, incorporating meteorological phenomena like diurnal cycles, persistence for smooth transitions, and sporadic weather fronts for sudden shifts.

This model is used to assess the system’s adaptability and performance under more life-like, continuously evolving conditions.

This realistic model generates time-series for temperature (T_t), wind speed (w_t), and humidity (H_t) using a mean-reverting autoregressive process of order 1 (AR(1)) for temporal persistence. For a generic weather variable $X_t \in \{T_t, w_t, H_t\}$, the state at time step t is given by:

$$X_t = \mu_{X,t} + \phi_X(X_{t-1} - \mu_{X,t-1}) + \sigma_X \epsilon_{X,t} \quad (7)$$

where ϕ_X is the persistence coefficient, σ_X is the process noise magnitude, and $\epsilon_{X,t} \sim \mathcal{N}(0, 1)$ is Gaussian noise. The time-varying mean $\mu_{X,t}$ captures daily patterns and sudden shifts:

$$\mu_{X,t} = X_{\text{base}} + \Delta X_{\text{diurnal}}(h_t) + \delta_{X,t} \quad (8)$$

Here, X_{base} is a seasonal baseline, $\Delta X_{\text{diurnal}}(h_t)$ is a sinusoidal function of the local hour h_t to model diurnal cycles, and $\delta_{X,t}$ is a stochastic shock term representing a weather front, which is zero unless a front is active. Visibility (V_t) is then derived as a non-linear function of humidity and wind speed: $V_t = V_{\text{base}} - f_{\text{haze}}(w_t) - f_{\text{RH}}(H_t) + \sigma_V \epsilon_{V,t}$, where haze decreases exponentially with wind and visibility loss is quadratic with high humidity. This model ensures a weather sequence with realistic properties such as temporal smoothness, daily patterns, and occasional abrupt shifts.

6 EVALUATION

This section presents the evaluation of our proposed knowledge-driven mechanism for run-time self-optimization under dynamic ODD changes. To assess the effectiveness of our approach, we propose an evaluation based on the smart farming scenario introduced in Section 2. The fleet’s decision variables include speed, altitude, camera resolution, spray rate, power limit factor representing the power capability of the drone given certain weather conditions and finally sensor sampling rate. The fleet’s main goal is to cover and monitor the entire agricultural field efficiently while adapting to the changing weather conditions. The main optimization objective is to minimize energy consumption during the operation, while covering the whole field area (c.f. Section 5.4). We used the environmental model described in Section 5.5 to simulate various environmental conditions and their impact on the drone’s decision-making process. We consider three representative weather scenarios, each corresponding to a distinct season of the year—winter, spring and summer—at a fixed latitude of 52.5°N (Berlin). For each scenario, we set the day of year $d \in \{90, 180, 330\}$ and the seasonal mean near-surface temperature $T_{\text{base}} \in \{5, 15, 25\}$. We simulated 10h of operation for each scenario, with ODD changes sampled every 5 minutes. The simulation was written with Python, using the PuLP library [26] with Gurobi [15] as the solver. It was run on an Intel Ultra 9 185H with 64 GB of RAM memory using the standard Python implementation for set-based operations and a custom implementation for the Zonotope operations.

We study two instantiations of the ODD abstraction: Γ_1 (coarse, thresholded regimes) and Γ_2 (fine-grained). These regimes influence cache comparability and thus cache hits and latency. Γ_1 groups similar conditions to stabilize decisions; Γ_2 retains higher fidelity representing a richer set of mappings of weather conditions to constraints. We set the extension tolerance to $\tau_{\text{ext}} = 0$ (0% allowable degradation in energy/coverage when extending cached regions).

6.1 Research Questions and Metrics

The evaluation is guided by the following research questions (RQs):

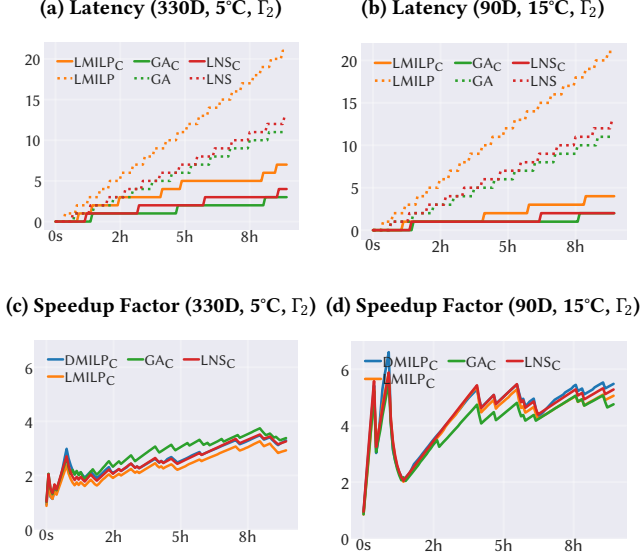
- RQ1 (Cache Effectiveness):** How effectively does the caching mechanism reduce decision latency compared to full re-optimization and no-cache baselines?
- RQ2 (Cache Overhead):** What is the computational and memory overhead of maintaining and querying the cache?
- RQ3 (Cache Hit Rate):** How frequently does the cache yield hits under varying ODD change patterns and can we infer the conditions that lead to higher hit rates?
- RQ4 (Adaptation Quality):** Does the cache maintain the same quality of configurations as the no-cache counterparts?
- RQ5 (Sensitivity and Scalability):** How resilient is the caching mechanism to model inaccuracies (drifts), and how does fleet size impact cache management and the overall scalability of the DT architecture?

We present the quantitative metrics used to address these RQs in Table 1 and Table 2, capturing the performance, efficiency, and feasibility of our approach. We report cache hit rate (%), latency percentiles (P50/P95/P99), and speedup (mean latency baseline / caching). Efficiency is captured by total cache footprint (KB), entries, per-entry size and final $\mathbb{CS}$ coverage (%). Solution quality is assessed by average energy consumption (Wh) subject to field coverage constraints. In order, the *cache hit rate* (%) measures the proportion of decision requests served from the cache, calculated as $(\text{hits} / (\text{hits} + \text{misses})) \times 100$, and serves as the primary indicator of cache effectiveness. The *total latency*, measured from an ODD change to the generation of a new configuration, is reported at the median (P50), 95th (P95), and 99th (P99) percentiles. P50, P95, and P99 latency are metrics that show how fast a system responds to requests by looking at percentiles, not just an average global value: P50 (median) indicates that half of all requests are faster than the value shown and half are slower; P95 means 95% of requests are completed faster than this time, with the remaining 5% taking longer; the same applies to P99 as 99% respectively, giving a somewhat accurate idea about the worst-case scenarios. The *speedup factor* (SF) expresses the ratio between average decision latency without and with caching, indicating overall performance gain. To assess resource usage, we report the *cache memory footprint* (MF) in Kilobytes and the *cache size* as the number of unique stored solutions. Finally, we evaluate *performance objectives*, measured as energy efficiency (energy/ha, lower is better) and field coverage (100% of the field must be covered) as objective constraint, across all strategies to assess potential trade-offs between speed of the approach versus quality.

6.2 Compared approaches

In order to validate our approach, we choose a suite of alternative and comparative strategies for handling dynamic ODD changes and optimization challenges, spanning a spectrum of methodologies, from full re-optimization to heuristic and learning-based methods. Each approach is applied to the same scenarios in two variants, namely with and without our caching mechanism, to isolate the impact of our proposed method. The following approaches are selected: (i) *Discrete MILP* (DMILP): The $\mathbb{CS}$ is discretized by generating a set of evenly spaced values for each decision variable within

Figure 3: Cumulative Latency (s) and Speedup Factor over two simulation hours for scenarios 330D, 5°C and 90D, 15°C under Γ_2 . The DMILP strategy is omitted for improved visibility.



the domain's bounds, and then exhaustively searched using a MILP solver; (ii) *Continuous MILP (MILP)*: The same formulation as before, but with continuous decision variables and piecewise-linear approximations for non-linear functions; (iii) *SBSE approaches*: Two meta-heuristic optimization algorithms, typically used in SBSE, namely Genetic Algorithm (GA) and Large Neighborhood Search (LNS), are implemented to solve the same problem.

To provide clarity in the results, we denote the cached version of each approach with a subscript C , e.g., DMILP $_C$ for the discrete MILP with caching. This notation helps to distinguish between the base methods and their enhanced versions with the caching mechanism.

6.3 Results

We summarize findings along RQ1–RQ4 and analyze the impact of ODD abstraction (Γ) on cache dynamics.

RQ1: Cache Effectiveness (Latency and Speedup). Across all scenarios, our memory-driven approach exhibits substantial latency reductions with flattening trends during operation.

As reported in Table 1, cached strategies consistently outperform their uncached counterparts, with speedup factors and cache hit rates increasing over time as the system learns and serves already established configurations from memory. As depicted in Figures 3a and 3b, the cached approaches show a trend of flattening cumulative latency curves, indicating that most ODD changes are served directly with insignificant delays. This trend is more evident in Figure 3c and 3d, showing how the speedup factor compounds over time and increases as the simulation progresses, peaking at $56.5\times$ for DMILP $_C$ under Γ_1 and at $\sim 9\times$ for DMILP $_C$ under Γ_2 . This,

along with the increase in hit rate indicates that the system is able to learn from past scenarios and leverage that knowledge to optimize future decisions. At the end of the simulation, the amount of explored $\mathbb{DS}$ reaches nearly 73% for Γ_1 and 60% for Γ_2 , indicating that a significant portion of the design space has been effectively cached and reused. Notably, for the more fine-grained ODD abstraction Γ_2 , the peak value for the speedup factor has been achieved during the mid-phase of the simulation. This is a consequence of its fine-grained nature, which leads to a larger number of uncomparable ODDs regimes and thus higher cache misses. Nevertheless, even under Γ_2 , the caching mechanism delivers significant speedups, demonstrating its robustness across different ODD abstractions.

RQ2: Cache Overhead. The cache remains compact across strategies with a final number of entries of 9 and 14 (for Γ_1 and Γ_2), showing total footprint ~ 8 –16 KB, and per-entry 0.86–1.13 KB. This is summarized in Table 2. The small memory overhead confirms the feasibility of our caching approach for resource-constrained platforms. Query times are negligible (< 1 ms) due to the efficient Zonotope representation, which also allows for rapid comparability checks and extensions as per Algorithm 1, which we measured to be seven for Γ_1 and three for Γ_2 .

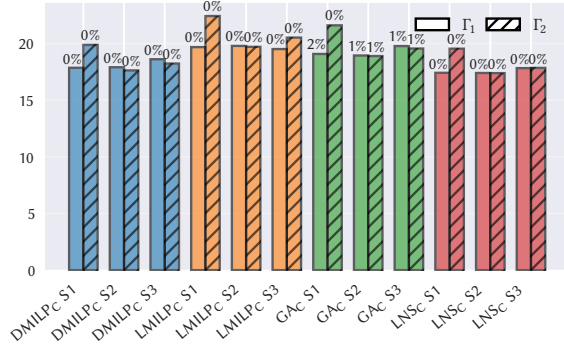
RQ3: Hit Rate and Learning Dynamics. As already mentioned above, the number of hits and the overall learning dynamics highly depend on the ODD abstraction Γ . As shown in Table 1, a coarser abstraction like Γ_1 yields higher hit rates ($> 92\%$), as it groups similar ODD states into larger discretizations. A finer abstraction like Γ_2 results in lower hit rates (~ 71 – 89%) due to the increased number of distinct ODD states. These patterns are consistent with the design of our comparability test in Algorithm 1 and the regime-based stabilization discussed in Section 5.2. Crucially, the cache footprint is nearly identical in both (c.f. Table 2), so choosing Γ_1 vs. Γ_2 is a timeliness–fidelity policy decision rather than a memory constraint.

RQ4: Adaptation Quality (Energy, Coverage). Across all scenarios and strategies, caching preserves solution quality. Figure 4 shows that for strategies delivering optimal solutions (DMILP, MILP), the average energy consumption remains identical between cached and uncached versions. For heuristic strategies (GA, LNS), the average energy remains within 1% of the uncached baselines (e.g., 17.4–19.8 Wh under Γ_1 ; 17.4–22.4 Wh under Γ_2). Furthermore, final coverage is unchanged for a given scenario (e.g., 29.4%, 54.5%, and 72.7% in Γ_1 ; 18.3%, 50.9%, and 58.3% in Γ_2), indicating that the cache serves guaranteed-optimal or extension-equivalent configurations without sacrificing objective values.

RQ5 (Sensitivity and Scalability). To evaluate the *system sensitivity* to model variations, we tested its resilience by simulating errors in the energy consumption model used by the DT (Equation 5). To this end, we introduced controlled perturbations for the PT, i.e. $E_{\text{real}}(x) = E_{\text{model}}(x) \times (1 + \delta)$, where $\delta \in \{0\%, 10\%, 20\%\}$, simulating drifts due to hardware aging (e.g., battery degradation, rotor wear) or unmodeled environmental factors (e.g. humidity). For each δ , we run a three hours long simulation to measure its influence on the system. As shown in Figure 5, the prediction error, measured as the difference between predicted and actual energy consumption, increases proportionally with δ . Notably, since the cache maintains

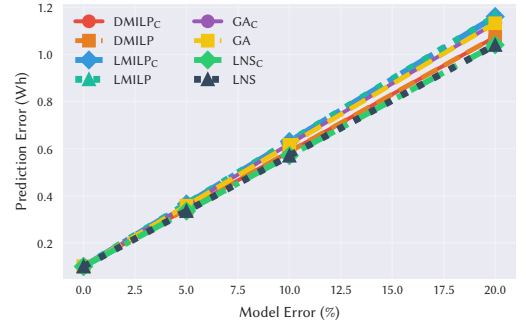
Table 1: Overview of the simulation results. Scenarios are denoted as S1(330D, 5°C), S2(180D, 25°C), and S3(90D, 15°C).

Strategy (Scenario)	Hit Rate		Speedup		Latency (ms)								Baseline		Coverage	
	(%)				P50		P95		P99		Avg.		Latency (ms)		(%)	
	Γ_1	Γ_2	Γ_1	Γ_2	Γ_1	Γ_2	Γ_1	Γ_2	Γ_1	Γ_2	Γ_1	Γ_2	Γ_1	Γ_2	Γ_1	Γ_2
GA (S1)	92	71	12.6	3.4	0.44	0.61	96.7	106.5	99.6	109.7	7.77	29.44	97.90	99.59	29	18
LNS (S1)	92	70	12.4	3.3	0.48	0.68	109.9	115.8	113.5	117.5	8.81	33.54	108.78	109.29	29	18
LP (S1)	92	70	11.0	2.9	1.01	1.52	183.0	225.1	190.8	261.8	14.93	61.11	164.93	178.90	29	18
MILP (S1)	92	70	12.6	3.3	0.81	1.34	49645.2	36161.8	50458.7	57066.2	3430.64	10145.93	43305.89	33438.56	29	18
GA (S2)	93	89	13.3	8.0	0.50	0.87	101.0	106.0	103.6	109.7	7.29	12.20	96.96	97.78	54	50
LNS (S2)	93	89	13.6	8.2	0.53	0.90	110.2	114.4	114.5	118.6	8.01	13.21	108.98	108.80	54	50
LP (S2)	93	89	12.6	7.6	1.21	1.84	182.0	190.3	186.9	198.8	13.42	22.59	168.87	171.79	54	50
MILP (S2)	97	89	14.6	7.7	1.01	1.42	49727.8	30933.0	50650.0	51433.7	3202.56	4080.09	46646.42	31579.62	54	50
GA (S3)	98	82	43.8	4.8	0.52	0.98	0.6	116.3	82.2	124.5	2.22	20.72	97.24	98.57	72	58
LNS (S3)	98	83	44.4	5.3	0.56	1.06	0.7	120.1	90.8	124.8	2.46	20.69	109.10	109.27	72	58
LP (S3)	98	83	41.5	5.1	1.11	2.02	1.8	202.5	153.8	240.3	4.27	35.72	177.39	180.72	72	58
MILP (S3)	98	84	56.5	5.5	0.85	1.66	1.5	33893.1	40727.0	55119.2	839.81	6231.44	47454.93	34153.19	72	58

Figure 4: Average energy consumption per scenario and caching strategy, for Γ_1 and Γ_2 . Numbers above bars show the difference compared to the baseline**Table 2: Resulting Cache Characteristics.**

Algorithm	# Entries		Entry Size (KB)		Total Size (KB)	
	Γ_1	Γ_2	Γ_1	Γ_2	Γ_1	Γ_2
GA	9	14	0.86	0.86	7.74	12.04
LP	9	14	1.13	1.13	10.17	15.82
LNS	9	14	0.86	0.86	7.74	12.04
DMILP	9	14	0.86	0.86	7.74	12.04

solution quality (c.f. RQ4), cached or baseline strategies exhibit identical error trends, confirming that the caching mechanism does not amplify model inaccuracies. Even for model errors up to 20%, the solutions remained feasible, however, with increased energy consumption due to suboptimal configurations being selected based on inaccurate predictions. This tolerance to model error (up to 20%) exceeds realistic field conditions (e.g. battery aging: 5–10% per year [28], rotor wear: 5–10% [24]), demonstrating robustness against gradual component degradation.

Figure 5: Impact of model error (δ) on the system in terms of prediction error on energy consumption.

To address *scalability*, we evaluated the DT’s performance as the fleet size M increased from 10 to 100 agents (10× scale-up). While the solver’s complexity for task assignment grows with M —reflected in the baseline DMILP latency increasing from ~35 s (10 drones) to ~350 s (100 drones)—the design space reduction process remains invariant to M , as it is driven by ODD volatility rather than the number of agents. Critically, the caching mechanism maintained effectiveness at scale: hit rates remained consistent at ~71% across all strategies, and speedup factors of 3.4× were observed even for the 100-drone fleet. Cache lookup times remained consistently below 1 ms, and the synchronization payload for sharing new entries across the fleet was measured at ≈ 0.86 KB per entry, independent of M . This validates that the zonotope-based knowledge representation scales gracefully with fleet size, as each cache entry represents a *region* in $\mathbb{DS}$ rather than individual agent assignments.

Discussion and Practical Considerations. While the evaluation clearly shows that: (i) our approach achieves large and consistent latency reductions; (ii) memory overhead is negligible (< 20 KB); (iii) solution quality is maintained; some comments on the validity are in order. We already highlighted that the choice of ODD abstraction Γ has effects on the performance of the approach. The choice of

this abstraction is a design choice that should be made based on the specific application requirements.

As already shown in Table 1, the choice of optimization strategy (exact vs. heuristic) can impact solution quality. Specifically, heuristic strategies may not always provide optimal solutions as ODDs are first encountered, therefore the solution stored in the cache may not be optimal. This might lead to less optimal configurations being served from the cache. While in our evaluation the difference is found to be negligible, an hybrid approach could be considered, where exact methods are used to periodically validate and update cached solutions, ensuring long-term optimality while still benefiting from the lower speed advantages of heuristics.

An important aspect that must be addressed is the presence of model deviations, as real systems and their operational environments may not perfectly adhere to their models. In general, incorrect model assumptions are a potential problem for many approaches (e.g. [3, 25, 39]), as they can lead to suboptimal or unfeasible configurations. While the sensitivity analysis in RQ5 shows that the proposed approach tolerates model errors of up to 20% while maintaining feasibility, these deviations may still compromise optimality and long-term cache validity, therefore requiring corrective action. To handle real-world volatility, we propose integrating an explicit runtime feedback mechanism into the MAPE-K loop. At the *Monitor* stage, the system continuously compares the predicted performance of the deployed configuration, $f(x^* | \text{ODD})$, with the observed performance, $\hat{f}(x^* | \text{ODD})$. If the resulting residual exceeds a predefined threshold δ , i.e. $|\hat{f}(x^* | \text{ODD}) - f(x^* | \text{ODD})| > \delta$, the corresponding cached configuration is conservatively flagged as low-confidence, indicating that it may no longer be optimal due to (i) model inaccuracies or system degradation, or (ii) environmental effects incorrectly captured by the current ODD abstraction.

One possible approach is to save different entries for different system conditions, hereby storing multiple configurations for the same ODD, but different system states. This would avoid invalidating cache entries due to system drifts, but will increase memory usage and knowledge management complexity and will not handle cases where the model is updated to consider additional factors (e.g. the impact of humidity on energy consumption). Alternatively, a larger than δ deviation can initiate a cache re-validation policy, ensuring that future predictions are representative of the current system state. Low-confidence entries are *invalidated*, i.e. excluded from serving future decisions until their validity is restored. While the PT temporarily executes a possibly suboptimal, but guaranteed-feasible fallback configuration to maintain safe operation, the DT asynchronously re-evaluates the invalidated cache entries by performing a re-optimization over the current configuration space and updates them to re-establish their validity, using the previous solution as a warm start to accelerate convergence. This ensures that cached solutions are never reused outside their validity assumptions, preserving correctness guarantees whenever cache entries are served, while enabling continuous operation.

Incorporating this runtime feedback, however, introduces two primary challenges. First, *computational overhead*: while zonotope operations are efficient, frequent model re-learning could lead to increased memory-management operations as well as fragmentation. This could be mitigated by scheduling re-evaluations during

low-demand periods or by prioritizing entries based on their usage frequency. Furthermore, the DT could proactively monitor model drift by tracking the distribution of residuals over time. If a trend of increasing residuals is detected, the DT can preemptively re-evaluate and update cache entries before they are explicitly flagged, introducing a form of predictive maintenance for the knowledge base. This background process is further facilitated by the fact that modern solvers provide sensitivity analysis tools that can be used to predict the impact of model changes on the optimal solution, which can be leveraged to assess whether a cached solution remains valid under model deviations, potentially avoiding unnecessary re-validation of cache entries. Second, *data consistency*: in a multi-agent fleet, different drones may observe different residuals for the same ODD due to local noise. The DT must therefore employ a consensus-based or probabilistic filter (e.g., a Kalman filter) before invalidating global cache entries to prevent "thrashing" where the knowledge base is constantly invalidated by outliers.

7 SUMMARY & OUTLOOK

This work presented a knowledge-driven approach to runtime self-optimization that reduces online search efforts by reusing proven-optimal configurations across shifting ODDs. Our approach, architecturally grounded in a MAPE-K/Digital Twin framework, treats the feasible configuration space as the dynamic intersection of a static design space and the current ODDs and employs a zonotope-based knowledge cache that correctly prunes explored regions and focuses optimization efforts on the unexplored ones. Our evaluation in a smart-farming drone scenario demonstrated that, compared to naïve re-optimization or purely heuristic search, our method preserves optimality guarantees while delivering substantial latency reductions during reconfiguration, providing up to 56x speedup with minimal memory and computation overhead. Nevertheless, several avenues for future work remain. Firstly, as already established in Section 6, the choice of ODD abstraction Γ significantly influences performance. Selecting an appropriate abstraction remains a design decision, contingent on specific application requirements, and further research could explore adaptive or context-aware abstraction strategies. Furthermore, an exhaustive ablation of cache policies (e.g., tolerance τ_{ext} , eviction, and merge heuristics) was not conducted, and theoretical bounds on merge-induced over-approximation and worst-case latency under adversarial ODD switching remain to be formalized. Finally, the effect of this approach on real-time systems, has yet to be validated and will be the subject of future work.

In summary, knowledge-driven design space reduction offers a practical approach to efficient and optimal runtime self-optimization for systems operating in dynamic ODDs. With stronger theoretical foundations, federated knowledge sharing, and real-system validation, we anticipate that this paradigm can be extended to a broader range of adaptive CPS.

ACKNOWLEDGMENTS

Supported by the Ministry of Economic Affairs, Industry, Climate Action and Energy of the State of North Rhine-Westphalia. Co-funded by the European Union. Grant number EFRE-20800498.

REFERENCES

- [1] Matthias Althoff. 2013. Reachability Analysis of Nonlinear Systems using Conservative Polynomialization and Non-Convex Sets. 173–182. <https://doi.org/10.1145/2461328.2461358>
- [2] ASAM e.V. 2022. ASAM OpenODD.
- [3] Alberto Bemporad, Manfred Morari, Vivek Dua, and Efstratios N. Pistikopoulos. 2002. The explicit linear quadratic regulator for constrained systems. *Automatica* 38, 1 (2002), 3–20. [https://doi.org/10.1016/S0005-1098\(01\)00174-1](https://doi.org/10.1016/S0005-1098(01)00174-1)
- [4] Felix Berkenkamp, Angela P. Schoellig, and Andreas Krause. 2018. Bayesian Optimization with Safety Constraints: Safe and Automatic Parameter Tuning in Robotics. *The International Journal of Robotics Research* 37, 7 (2018), 671–683.
- [5] Ilhem Boussaid, Patrick Siarry, and Mohamed Ahmed-Nacer. 2017. A survey on search-based model-driven engineering. *Automated Software Engg.* 24, 2 (June 2017), 233–294. <https://doi.org/10.1007/s10515-017-0215-4>
- [6] Luca Brodo, Giuseppe Scalora, and Stefan Henkler. 2025. Towards a Digital Twin Framework for Secure and Efficient Cyber-Physical Transportation Systems. In *Proceedings of the 1st Workshop on Modeling and Verification for Secure and Performant Cyber-Physical Systems* (UC Irvine Student Center, Irvine, CA, USA) (*MoVe4SPS '25*). Association for Computing Machinery, New York, NY, USA, Article 1, 6 pages. <https://doi.org/10.1145/3735948.3736156>
- [7] BSI Standards. 2020. PAS 1883: Operational Design Domain (ODD) taxonomy for an Automated Driving System (ADS)—Specification.
- [8] Tao Chen, Miqing Li, Ke Li, and Kalyanmoy Deb. 2020. Search-Based Software Engineering for Self-Adaptive Systems: Survey, Disappointments, Suggestions and Opportunities. *arXiv preprint arXiv:2001.08236* (2020).
- [9] Betty H. C. Cheng et al. 2014. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In *Models@run.time*. Springer International Publishing, Cham, 101–136.
- [10] Rogério de Lemos, Holger Giese, Hausi Müller, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha Villegas, Thomas Vogel, Danny Weyns, Luciano Baresi, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Ron Desmarais, Schahram Dustdar, Gregor Engels, and Jochen Wuttke. 2013. *Software Engineering for Self-Adaptive Systems: A Second Research Roadmap*. 1–32. https://doi.org/10.1007/978-3-642-35813-5_1
- [11] Peter J. Denning. 1968. Thrashing: its causes and prevention. In *Proceedings of the December 9–11, 1968, Fall Joint Computer Conference, Part I* (San Francisco, California) (*AFIPS '68 (Fall, part I)*). Association for Computing Machinery, New York, NY, USA, 915–922. <https://doi.org/10.1145/1476589.1476705>
- [12] Kleber Gama and Sari Al-Haj. 2017. A utility-based approach for self-adaptation of IoT systems. In *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 136–142. <https://doi.org/10.1109/SEAMS.2017.19>
- [13] Gerald Gamrath, Benjamin Hiller, and Jakob Witzig. 2015. Reoptimization Techniques for MIP Solvers. In *14th International Symposium on Experimental Algorithms (SEA 2015) (Lecture Notes in Computer Science, Vol. 9125)*, Paolo G. Franciosa, Daniele Frigioni, Tiziano Leidi, and Camil Demetrescu (Eds.). Springer, 181–192. https://doi.org/10.1007/978-3-319-20086-6_15
- [14] Gustavo Goldschmidt, Helder G. D. L. F. Junior, Rafael M. D. F. e Silva, and Thais Batista. 2017. Delta-oriented adaptive programming for self-adaptive systems. In *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 114–124. <https://doi.org/10.1109/SEAMS.2017.12>
- [15] Gurobi Optimization, LLC. 2024. *Gurobi Optimizer Reference Manual*. <https://www.gurobi.com> Version 11.0.
- [16] Mark Harman, Edmund Burke, John Clark, and Xin Yao. 2012. Dynamic adaptive search based software engineering. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (Lund, Sweden) (ESEM '12)*. Association for Computing Machinery, New York, NY, USA, 1–8. <https://doi.org/10.1145/2372251.2372253>
- [17] Mark Harman, Yue Jia, and Yuanyuan Zhang. 2015. Achievements, Open Problems and Challenges for Search Based Software Testing. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. 1–12. <https://doi.org/10.1109/ICST.2015.7102580>
- [18] Christian Henke, Matthias Tichy, Tobias Schneider, Joachim Böcker, and Wilhelm Schafer. 2008. System Architecture and Risk Management for Autonomous Railway Convoys. 1–8. <https://doi.org/10.1109/SYSTEMS.2008.4518986>
- [19] Martin Hirsch, Stefan Henkler, and Holger Giese. 2008. Modeling collaborations with dynamic structural adaptation in mechatronic UML. In *Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-Managing Systems (Leipzig, Germany) (SEAMS '08)*. Association for Computing Machinery, New York, NY, USA, 33–40. <https://doi.org/10.1145/1370018.1370026>
- [20] IBM Corporation. 2005. *An Architectural Blueprint for Autonomic Computing*. Third Edition.
- [21] Pooyan Jamshidi, Aakash Ahmad, and Claus Pahl. 2017. Uncertainty-aware self-adaptation in the cloud. In *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 11–21. <https://doi.org/10.1109/SEAMS.2017.3>
- [22] Jeffrey O. Kephart and David M. Chess. 2003. The Vision of Autonomic Computing. *IEEE Computer* 36, 1 (2003), 41–50. <https://doi.org/10.1109/MC.2003.1160055>
- [23] J. Li, M. Zagorowska, G. De Pasquale, A. Rupenyan, and John Lygeros. 2024. Safe Time-Varying Optimization based on Gaussian Processes with Spatio-Temporal Kernel. In *Proceedings of NeurIPS 2024*. (accepted). Vancouver, Canada, 9–15 August. preprint: <https://arxiv.org/abs/2409.18000> [arXiv:2409.18000](https://arxiv.org/abs/2409.18000), available 27 Sep 2024.
- [24] Harry Liuhan and Toby Bryce-Smith. 2025. Experimental Characterisation of Low Reynolds Number Propeller Performance from Erosion on a Triangular Aerofoil. (Oct. 2025). <https://doi.org/10.21203/rs.3.rs-7875205/v1> Experimental MAV/UAUV propeller erosion; reports that leading-edge erosion yields an average 9.4% reduction in thrust-to-power metric (CT/CP) across test points..
- [25] Tobia Marcucci and Russ Tedrake. 2020. Warm Start of Mixed-Integer Programs for Model Predictive Control of Hybrid Systems. *arXiv:1910.08251v2 [eess.SY]*. [arXiv:1910.08251 \[eess.SY\]](https://arxiv.org/abs/1910.08251)
- [26] Stuart Mitchell et al. 2011. PuLP: A Linear Programming Toolkit for Python. <https://coin-or.github.io/pulp/> Version 2.7.
- [27] Sandeep Neema, Theodore Bapty, and Gabor Karsai. 2003. Constraint-guided self-adaptation. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE)*. IEEE, 283–288.
- [28] M. Pervaiz, H. Ali, et al. 2023. Assessment of the calendar aging of lithium-ion batteries for a long-term—Space missions. *Frontiers in Energy Research* (2023). <https://doi.org/10.3389/fenrg.2023.1108269> Reports calendar-aging capacity degradation normalized to % per year; notes rates around 10%/year at elevated temperature or SOC (chemistry-dependent)..
- [29] Felipe Quin, Weiqing Shen, Rui Andrade, and Alex Silva. 2018. ML2ASR: Machine learning for adaptation space reduction in self-adaptive systems. In *Proceedings of the 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 76–87.
- [30] Alex Ramirez, Betty H.C. Cheng, and Antonio Carzaniga. 2011. PLATO: a genetic algorithm approach to reconfiguration planning. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 95–104.
- [31] Joseph K. Scott, Davide M. Raimondo, Giuseppe Roberto Marseglia, and Richard D. Braatz. 2016. Constrained zonotopes: A new tool for set-based estimation and fault detection. *Automatica* 69 (2016), 126–136. <https://doi.org/10.1016/j.automatica.2016.02.036>
- [32] Paul Stevens and Hadi Bagheri. 2020. Thallium: Pruning runtime adaptation spaces with utility bounds. In *Proceedings of the 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 1–12.
- [33] Yanan Sui, Alkis Gotovos, Joel Burdick, and Andreas Krause. 2015. Safe Exploration for Optimization with Gaussian Processes. In *Proc. ICML*. 997–1005.
- [34] Petter Tøndel, Tor Arne Johansen, and Alberto Bemporad. 2001. An algorithm for multi-parametric quadratic programming and explicit MPC solutions. In *Proceedings of the 40th IEEE Conference on Decision and Control (CDC)*, Vol. 2. 1199–1204. <https://doi.org/10.1109/CDC.2001.980302>
- [35] Mirco Tribastone. 2014. A monotonicity-based approach to performance modeling. *IEEE Transactions on Software Engineering* 40, 5 (2014), 453–467.
- [36] M. L. Villota, E. P. Rivera, C. A. C. Bastidas, J. F. G. G. de B, and P. R. F. Cunha. 2018. Context-aware runtime reconfiguration model for self-adaptive service-based systems. In *Proceedings of the 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 154–160. <https://doi.org/10.1145/3194133.3194146>
- [37] Danny Weyns. 2019. *Software Engineering of Self-adaptive Systems*. Springer International Publishing, Cham, 399–443. https://doi.org/10.1007/978-3-030-00262-6_11
- [38] Danny Weyns, Mauro Caporuscio, and Carlo Ghezzi. 2022. DLASer+: Online deep learning adaptation space reduction for self-adaptive systems. In *Proceedings of the 17th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 45–56.
- [39] Yibo Zheng, Alessio Benavoli, Marco Zaffalon, et al. 2024. On Safety in Safe Bayesian Optimization. *arXiv preprint arXiv:2403.12948* (2024).