technische universität
dortmund

Anytime ROS 2: Timely Task Completion in Non-Preemptive Robotic Systems

Harun Teper, Daniel Kuhse, Yun-Chih Chen,
Georg von der Brüggen, Zhishan Guo, Jian-Jia Chen
TU Dortmund University, Department of Computer Science, Dortmund, Germany
National Tsing Hua University, Department of Computer Science, Hsinchu, Taiwan
North Carolina State University, Department of Computer Science, Raleigh, USA
Lamarr Institute for Machine Learning and Artificial Intelligence, Germany

Citation: XX.XXXX/XXXXXXX

Preprint Version. Citation Info: XX.XXXX/XXXXXXX

BibTeX:

```
@inproceedings{teper2026anytime,
  author={H. {Teper} and D. {Kuhse} and Y.-C. {Chen}
          and G. {von der Br\"{u}ggen} and Z. {Guo} and J.-J. {Chen}},
  booktitle={32nd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)},
  title={Anytime ROS 2: Timely Task Completion in Non-Preemptive Robotic Systems},
  year={2026}
}
```

©2026 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

cs 12 computer science 12

Anytime ROS 2: Timely Task Completion in Non-Preemptive Robotic Systems

Harun Teper*, Daniel Kuhse*, Yun-Chih Chen†, Georg von der Brüggen*, Zhishan Guo‡ and Jian-Jia Chen*§
*TU Dortmund University, Germany, †National Tsing Hua University, Taiwan,
‡North Carolina State University, USA, §Lamarr Institute, Germany
{harun.teper, daniel.kuhse, georg.von-der-brueggen, jian-jia.chen}@tu-dortmund.de,
yunchih@cs.nthu.edu.tw, zguo32@ncsu.edu

Preprint Version. Citation Info: XX.XXXX/XXXXXXX

***Abstract*—Ensuring timely and predictable data propagation is essential in real-time robotic systems, yet they increasingly incorporate computationally expensive, long-running tasks, such as deep neural networks. Integrating these tasks into the Robot Operating System 2 (ROS 2) is challenging due to the non-preemptive design of the ROS 2 *executor*, which can lead to high task interference and unpredictability. This paper presents a structured approach to transform these long-running tasks into a series of smaller, more manageable subtasks. The transformation inherently introduces preemption points, which enable an *anytime functionality* where tasks can be terminated early while delivering progressively better results. The approach leverages the existing ROS 2 *actions* framework to provide a reliable task-cancellation mechanism and feedback channel. This solution requires no modification to the existing ROS 2 codebase and remains compatible with existing ROS 2 end-to-end latency analyses. The evaluation of Anytime RRT* and AnytimeYOLO demonstrates that timely results can be achieved via flexible cancellations.**

***Index Terms*—Anytime Algorithms, End-to-End Latency, Real-Time Systems, Robot Operating System 2, Scheduling**

I. Introduction

Many modern robotic systems, such as autonomous driving platforms [48], must ensure timely and predictable data propagation to maintain safe operation. These systems often incorporate computationally expensive tasks like sampling-based planning and vision-based sensing. However, due to their high computational demand, integrating these tasks while maintaining the required timing guarantees is challenging, especially in dynamic environments. Moreover, such computationally expensive tasks often require integration with external accelerators like GPUs, whose non-preemptive execution must be carefully managed to maintain predictability and timeliness.

To ease the integration effort and utilize existing software solutions, instead of designing from scratch, modern systems are typically built using modular architectures [26], [48]. A popular choice is the Robot Operating System 2 (ROS 2) [41], which provides the corresponding tools and libraries.

The ROS 2 executor is a user-space scheduler that manages tasks through a non-preemptive scheduling mechanism. Once the executor chooses a task instance for its thread to execute, it runs to completion without interruption from instances of other tasks it manages. The non-preemptive ROS 2 scheduling approach (i) results in reduced scheduling overhead, (ii) provides a predictable execution pattern that eases the scheduling analysis, and (iii) allows high processor utilization.

However, while this approach is effective for short, non-blocking tasks, it creates significant challenges when integrating computationally expensive tasks due to the non-preemptive execution behavior. Specifically, a long-running task can monopolize an executor thread, causing high interference for other tasks and potentially preventing them from meeting their timing constraints. Furthermore, this non-preemptive design inherently prevents a task from being stopped early, even if the system workload is high, an intermediate result is already *good enough*, or saving energy is a priority.

One potential solution is to offload these long-running tasks to separate, unmanaged operating system threads. These tasks run independently of the ROS 2 executor, enabling preemption and reducing interference with other tasks. However, unmanaged threads are not controlled by the ROS 2 executor, making it difficult to analyze and guarantee timing properties for the overall system. The use of unmanaged threads in ROS 2 systems was highlighted by Fan et al. [18] as a common practice to bypass the limitations of the ROS 2 executor, but they did not provide any solution for the resulting issues.

Our goal is to solve the challenges of long-running tasks in ROS 2 systems without relying on unmanaged threads. Instead, we work within the constraints of the ROS 2 executor. We propose to break up long-running tasks into a series of smaller, more manageable subtasks, which can already reduce interference and improve system responsiveness. However, assuming these tasks can also produce intermediate results of increasing quality, common in iterative algorithms for planning or perception, we can go one step further.

Specifically, we can transform the long-running task into an *anytime algorithm* [19], which allows termination at *any time* while delivering progressively better results with increased runtime. In dynamic situations, where optimal results may not always be required, anytime algorithms help to maintain predictability while delivering usable results. They also allow for dynamic resource allocation, enabling the system to adapt to changing conditions and requirements.

In this work, we show how to transform long-running tasks into anytime algorithms within the ROS 2 framework, adhering to its non-preemptive scheduling model. We provide a generic solution that can be implemented in any ROS 2 system without relying on external software packages or changing the ROS 2 source code, ensuring compatibility and easy integration.

Preprint Version. Citation Info: XX.XXXX/XXXXXXX

Specifically, we leverage *ROS 2 actions*, a client-server communication interface designed for long-running tasks, to manage anytime algorithms in ROS 2. They enable us to create a client to initiate computation on the action server via a request, to receive feedback on the computation status, and to cancel and terminate the computation if needed. That is, the computation and management of the anytime algorithm are executed in the action server, while the controls to start, stop, and query the anytime algorithm are handled by the action client, providing the necessary modular interfaces [19].

As ROS 2 actions have not yet been analytically explored in the literature, we also extend the existing timing analysis for end-to-end latencies in ROS 2 when converting a long-running task to an anytime algorithm executed via ROS 2 actions.

Challenges and Contributions. In this paper, we explore the transformation of long-running tasks into anytime algorithms for ROS 2 systems. We strive to keep our design simple, such that our solution can be adopted with minimal integration effort, **requiring no modification to the existing ROS 2 codebase**. Specifically, we address several key *challenges* for using *ROS 2 actions* to facilitate task interruptions and timely completions in a client-server approach:

- **Resolving Architecture Limitations.** We enable the execution of long-running tasks as anytime algorithms within the constraints of the ROS 2 executor's non-preemptive scheduling, improving system responsiveness and reducing task interference. Specifically, we present our solutions for single-threaded executors in Section V and extensions for multi-threaded executors and accelerators in Section VI. Our implementation is based solely on the interfaces and task types provided directly by ROS 2, providing a generic solution for integrating most anytime algorithms, *requiring no change to the underlying ROS 2 codebase*. Our source code is publicly available.[1]
- **Trade-offs.** Design trade-offs, such as the *task granularity* and *result computation*, are discussed in Section VII.
- **Formal Guarantees.** We analyze the impact of our transformation on the end-to-end latencies and task interference of ROS 2 systems in Section VIII. This includes the analysis of chains that include the transformed anytime algorithm and those that do not, focusing on the effects on timing guarantees and system performance.
- **System Validation.** We explore and evaluate different ROS 2 anytime architectures using RRT* path planning and an AnytimeYOLO object detector. In Section IX, we show how the architectures and configurations affect cancellation delays, efficiency, and task interference, and how the anytime algorithms perform for both time-based and quality-based cancellation triggers.

Paper Organization: In addition to the sections mentioned above, Sections II and III briefly introduce anytime algorithms and ROS 2, respectively. Section IV defines the studied problem. Section X and Section XI discuss future research and related work. Section XII concludes the paper.

[1]https://github.com/tu-dortmund-ls12-rt/Anytime-Development

II. Anytime Algorithms

This section provides an overview of anytime algorithms' fundamental benefits and their implications for real-time robotic systems. Optimization and iterative approximation algorithms, such as robot path planning via Monte Carlo Tree Search [13] and trajectory planning for autonomous racing cars [45], are often among the most computationally demanding tasks within autonomous driving software stacks [6].

Implementing these algorithms in a straightforward manner can lead to long execution times, which can interfere with the timely execution of other critical tasks in the overall system. Simple solutions are *time-budgeted* (or contract) algorithms [46], which are given a fixed time budget in advance to return a result, limiting their execution time.

A more flexible approach (compared to the budgeted approach) is to use *anytime* algorithms [14], [59], which can return the best intermediate result calculated so far at *any time* during their execution. Furthermore, the quality of the result typically improves with longer computation times, reaching a bounded optimal result if run to completion. This anytime property allows them to adapt to changing time requirements, use quality measures to track the result's quality, and ensure timely responses. Furthermore, if more resources than expected are available at runtime, anytime algorithms can adjust their computation accordingly, potentially yielding better results than static, time-budgeted algorithms.

To efficiently deploy anytime algorithms in practice, the underlying software framework must provide the necessary interfaces to (1) start the algorithm, (2) stop the computation, (3) monitor the algorithm's state, and (4) query intermediate results at any time. These interfaces allow for controlling the algorithm's execution and for dynamically adapting to changing time or performance requirements.

However, integrating anytime algorithms into an existing framework is not straightforward. Ideally, the underlying framework should provide the possibility to interrupt the computation and return the best result so far at any moment. Yet, for ROS 2, due to its non-preemptive executor, a design change is required to enable this functionality.

Thus, we need to identify which algorithms can be suitably adapted into anytime algorithms, and how to best implement them within the constraints of the target framework. For example, the aforementioned algorithms that feature an iterative structure can be converted into anytime algorithms by wrapping the algorithm in time-interruptible code [19]. More broadly, any sampling-based or iterative algorithm that can produce meaningful intermediate results is a natural candidate for anytime execution. In autonomous driving, this includes algorithms such as RRT* [24] for path planning, particle filters [15] for localization, and iterative model predictive control (MPC) solvers [43], in addition to the various anytime neural networks discussed in the related work.

The underlying accelerator devices (GPUs, FPGAs, or TPUs) also pose further challenges for (i) designing the computation function on the accelerator, and (ii) interacting with the functions on the CPU that manage anytime algorithms.

Preprint Version. Citation Info: XX.XXXX/XXXXXXX

III. ROS 2 Fundamentals

This section provides a high-level overview of the ROS 2 system model. Here, we focus on the aspects relevant to deploying anytime algorithms in ROS 2, while details are provided later on where needed. More detailed introductions to ROS 2 from a real-time perspective can be found in the literature [9], [11], [53], [55]. We use the LTS release ROS 2 Humble [41] as the reference implementation.

In ROS 2, *nodes* represent the system components. Each node contains a set of *tasks*, which define the node's functionality. ROS 2 provides periodically activated *time-triggered tasks* (*timers*) and *event-triggered tasks* (see Section V-C), which are activated by incoming events. Each task's function, called a *callback*, is executed when the task is scheduled.

The underlying *Data Distribution Service* (DDS) middleware facilitates communication between event-triggered tasks via three *communication patterns*. (i) For *one-way communication*, tasks can publish messages via *publishers* to shared *topics*, while the event-triggered tasks subscribed to the topic immediately receive and process these messages. (ii) *Services* provide a *request-response communication* pattern between two event-triggered tasks, a *server* and a *client*, which activate each other. (iii) *Guard conditions* are not intended for inter-node communication but allow *event-driven control* of tasks within the same node via custom (external) *trigger* signals.

The execution behavior of a ROS 2 system is managed by a set of *executors*, which are best suited for running short, non-blocking tasks. Each node (and thus its tasks) is assigned to **exactly one** (either single- or multi-threaded) executor. Task priorities are unique on each executor. An executor has two alternating phases: (i) *polling points* where the *wait set* of currently ready tasks is collected, and (ii) *processing windows* where these tasks are executed non-preemptively in priority order. This behavior is explained in detail in Section V-A.

Callback groups prevent concurrent access to shared resources in a *multi-threaded executor* since, at any time, only one task of a callback group can be executed. While there are other types of callback groups, we only utilize *mutually-exclusive callback groups*, as detailed in Section VI-A.

IV. Problem Definition

The problem of supporting anytime algorithms in ROS 2 is broken down into five smaller, more manageable subproblems to derive a more detailed understanding of the challenges. We start with three subproblems related to the single-threaded executor, which are addressed in Section V. The latter two subproblems are related to the multi-threaded executor and accelerator devices, addressed in Section VI.

1. Enabling Anytime Computation. Because of the non-preemptive nature of the ROS 2 executor, with its polling points and processing windows, long-running tasks block the execution of other tasks, preventing timely responses to new inputs and thus making anytime computation infeasible. Hence, given the limitations of the ROS 2 executor, we need to split long-running tasks into segments, introducing preemption points for interruptions and returning intermediate results.

2. Anytime Task Management. The original long-running task may have been directly activated either by a timer or an event, and simply produced an output. Yet, after splitting a task into smaller segments, a task activation mechanism is needed to manage the execution flow, handle cancellations, compute intermediate results, and return the best result so far.

3. Anytime Interfaces. Leveraging ROS 2's modularity through its communication infrastructure, we discuss and choose an appropriate communication interface for anytime algorithms, providing a generic and reusable solution.

4. Concurrency and Synchronization for Shared Resources. While multi-threaded environments enable parallelization, efficient synchronization of shared resources is required to provide fast cancellation and real-time responsiveness. We explore solutions for the ROS 2 multi-threaded executor that (i) allow safe concurrent access to shared resources without compromising performance, and (ii) utilize the natively provided ROS 2 callback groups for synchronization.

5. Accelerator Synchronization and Delays. We investigate the efficient integration of accelerator devices, such as GPUs or FPGAs, into the anytime ROS 2 framework, as they are commonly used for computationally intensive tasks. We discuss different synchronization mechanisms between the CPU-based managing functions in ROS 2 and the accelerator-based computation functions to minimize delays and overheads.

V. Anytime ROS 2 for Single-Threaded Executors

This section discusses the integration of anytime algorithms into ROS 2, focusing on providing clear examples of the potential problems and how we address them. This section considers the case where the anytime algorithm is completely managed by and executed on a single-threaded executor.

A. Enabling Anytime Computation

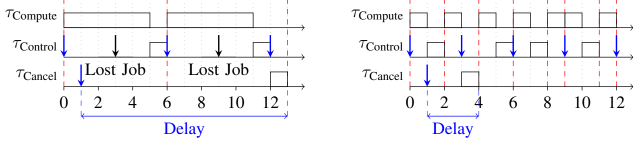
We first provide a detailed overview of the scheduling mechanism of the ROS 2 executor, followed by a discussion on how to adapt long-running tasks to enable timely cancellations.

Executors determine the set of eligible task instances (jobs), the so-called *wait set*, and decide which job to execute next. The wait set is managed in two distinct, alternately recurring phases: *polling points* and *processing windows*.

During a *polling point*, the ROS 2 executor collects any *ready* jobs and adds (at most) one job per task to the wait set. Timers are ready if their period has expired since they were last executed. However, ROS 2 discards additional timer jobs if multiple periods have elapsed since the last execution. For event-triggered tasks, the executor samples the task if there are pending events, taking the oldest if multiple are pending.

In a *processing window*, the tasks in the wait set are selected based on their priorities and executed non-preemptively. By design, ROS 2 prioritizes tasks based on their types: $P_{\text{Timer}} > P_{\text{Subscription}} > P_{\text{Server}} > P_{\text{Client}} > P_{\text{Guard}}$ (i.e., Timer has the highest priority). Tasks of the same type are prioritized based on the order in which they were added to the executor.

From a concrete schedule, we determine the delays involved in cancellation requests for the single-threaded executor.



(a) A long-running task blocks the executor, causing scheduling delays and lost jobs.

(b) Splitting the computation into smaller segments reduces delays and eliminates lost jobs.

Fig. 1: Scheduling delays and lost jobs in the single-threaded executor and the impact of splitting the computation. Blue arrows indicate executed jobs, black arrows indicate lost jobs, and red dashed lines indicate polling points.

Example 1 (Figure 1a). *Consider three tasks* $\tau_{Compute} > \tau_{Control} > \tau_{Cancel}$, *where* τ_{Cancel} *processes the cancellation request. The red-dashed lines indicate the polling points, the blue arrows indicate the time of activation, while the black arrows indicate lost jobs. The task* $\tau_{Control}$ *is activated every three time units, while the task* τ_{Cancel} *is activated at time* 1.
Lost jobs. *In ROS 2, if a timer's period elapses multiple times since the task was last executed, only one job is added to the wait set, and the other jobs are discarded (lost). In our example, the task* $\tau_{Control}$ *is activated at* 0, 3, 6, 9, *and* 12. *However, executing* $\tau_{Compute}$ *results in a long processing window and the activations at time* 3 *and* 9 *are lost.*
Long Delays. *Although the task* τ_{Cancel} *is activated at time* 1, *indicated by the blue arrow, it is not sampled and added to the wait set until time* 6. *Then, it is executed in the next processing window, where* $\tau_{Compute}$ *and* $\tau_{Control}$ *have higher priority. Hence,* τ_{Cancel} *starts at time* 12 *and finishes at time* 13, *resulting in a total delay of* 12 *time units.*

Example 1 shows two main sources for delays of cancellation requests: (i) the remaining time for the current processing window, and (ii) the time until the cancellation request is executed in the following processing window. Furthermore, long execution times of tasks can result in timer jobs being lost if the timer's activation periods elapse multiple times before another job of the timer is added to the wait set.[2]

Proposed Solution: We split the *computation* of an anytime algorithm into multiple *segments*, which are processed one at a time (i.e., each segment is an individual job). This shortens the processing windows, allowing other tasks (including cancellation requests) to be sampled more frequently and processed promptly. We first illustrate the idea with Example 2.

Example 2 (Figure 1b). *Consider Example 1 when* $\tau_{Compute}$ *is split into multiple segments of one time unit. Now, every second processing window contains both the* $\tau_{Compute}$ *and* $\tau_{Control}$ *tasks, with no lost jobs. The* τ_{Cancel} *task, activated at time* 1, *is sampled at the polling point at time* 2, *executed in the next processing window at time* 3, *and finishes at time* 4, *reducing the total delay to* 3 *time units.*[3]

[2]Event-triggered tasks like subscriptions can also lose jobs if they receive too many messages, causing their FIFO buffer to discard the oldest ones.

[3]We only show one case for brevity. The maximum delay can be observed if the task is released right after any polling point in this example.

The example shows two key benefits of this approach: (1) reduced delays for other tasks, and (2) potentially fewer lost jobs. Our approach introduces preemption points, allowing other tasks to suspend, resume, or cancel the computation, and enabling it to compute and return intermediate results.

B. Anytime Task Management

The solution in Section V-A enables running iterative algorithms as anytime algorithms by splitting their computation into multiple jobs. For example, an anytime neural network can now be split layer by layer, an RRT*-based planner iteration by iteration, or an MPC solver step by step. However, in comparison to the previous long-running task, which was a single job activated by one time-triggered or event-triggered task, the new approach requires a more sophisticated scheduling mechanism to manage the multiple jobs effectively.

To this end, two more **requirements** must be addressed:

1. **Correct Execution Flow:** The correct execution flow of the original anytime algorithm upon such fine-grained 'splits' must be maintained, even when segments need different amounts of time to compute.
2. **State Management:** The current state of the anytime algorithm must be passed to the next segment and to the result computation with minimal overhead. This is especially important for algorithms that handle large amounts of data or have many short-running segments.

Proposed Solution: We use ROS 2's *guard conditions*, an event-driven signaling mechanism providing a flexible and customizable solution to activate tasks within a ROS 2 node. Regarding Requirement 1, guard conditions can be activated by other tasks based on any logical function, allowing for more dynamic and responsive task management.

Regarding Requirement 2, since guard conditions operate within a single ROS 2 node, they can directly access its shared data structures, allowing each segment to access and update the algorithm's state without additional data transfer.

Implementation Variations: The solution we propose, shown in Figure 2, can be implemented in multiple ways using guard conditions. Its core is the management task (shown in blue), which is responsible for deciding which task to activate next. It receives external signals (shown in orange) to *start* or *cancel* the algorithm. It also sends status messages as *feedback*, and returns intermediate results upon *finish*. In addition to the management task, there are three main computation functions: *compute*, *result*, and *return*: *compute* performs one segment of the anytime algorithm, updating the current *internal state*, *result* computes the current intermediate result based on the internal state, and *return* sends the intermediate result back.

Depending on the given requirements and the specific algorithms, the concrete implementation of this architecture can vary. For example, the management task can be implemented as a single guard condition that activates itself after each execution of the compute, result, and return functions. Alternatively, all functions can be implemented as separate guard conditions that activate each other, though special care must be taken to avoid potential race conditions.

Preprint Version. Citation Info: XX.XXXX/XXXXXXX

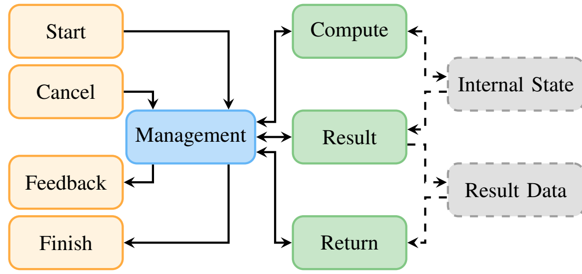


Fig. 2: Management architecture of anytime algorithms in ROS 2, showing the functions (management, compute, result, return) and data structures (internal state, result data) used to coordinate the segmented computation.

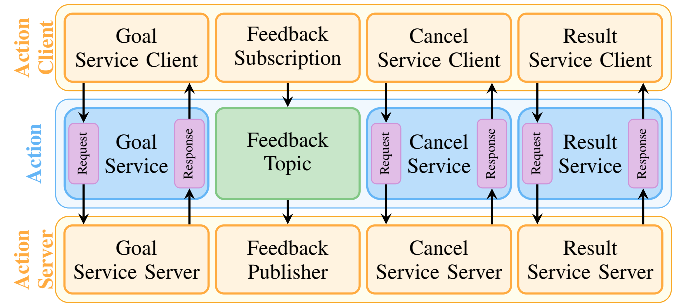


Fig. 3: Overview of ROS 2 Actions, illustrating the goal, result, and cancel services, and the feedback topic between the action client and server used to manage the anytime computation.

Preprint Version. Citation Info: XX.XXXX/XXXXXXX

In summary, guard conditions provide a flexible mechanism to manage the different tasks of anytime algorithms, allowing the solution to be tailored to the specific algorithm and the system requirements. In Section VII, we present two different designs for realizing our approach and discuss their trade-offs, such as the granularity of the *compute* and *result* tasks.

C. Communication Interface Design

Our solution requires an effective communication interface to interact with the anytime algorithm. Specifically, the interface should feature mechanisms to *start*, *cancel*, *finish*, and *query* the anytime algorithm, allowing for fine-grained control. Moreover, this interface should be generic, requiring minimal effort to integrate anytime algorithms into ROS 2.

ROS 2 provides three communication interfaces to choose from: *topics*, *services*, and *actions* [8]. We first discuss each option and then explain why ROS 2 *actions* are the most suitable choice for implementing anytime algorithms.

Comparison of ROS 2 Interfaces: *Topics* enable one-way, broadcast-style communication between *publishers* and *subscriptions*. They are not suitable for anytime algorithms, as their lack of a direct response mechanism complicates fine-grained control and coordinated execution.

Services provide a *request-response* communication pattern between a *server* and a *client* through two event-triggered tasks that activate each other. Specifically, a client sends a request to a server, which responds after processing the request. Managing start, cancellation, and response of the anytime algorithm would require multiple individual clients and servers, complicating the implementation and increasing the risk of errors. Services are also not designed to provide frequent messages for monitoring the computation state, making it difficult to implement adaptive decision-making.

Actions are one of ROS 2's core communication types, designed for long-running tasks. They are built upon services and topics, providing a higher-level interface where an action client can send a *goal* (e.g., the parameters for running the algorithm) to an action server for processing. Importantly, two native ROS 2 action features make them particularly suitable: (1) the ability for clients to terminate the goal early, and (2) the option to provide *feedback* for adaptive decision-making.

Proposed Solution: Actions provide a standardized interface for the necessary callbacks that initiate, monitor, and terminate anytime algorithms. Compared to manually implementing anytime algorithms with topics and services, integrating anytime algorithms with ROS 2 actions leverages their built-in features, increasing reliability and maintainability.

The interactions between action clients and action servers consist of four main components: the *goal service*, *feedback topic*, *result service*, and *cancel service*. After sending a goal to the action server, the action client waits for the server to accept the goal. If accepted, the server starts processing the goal and responds with references to the relevant feedback topic, result service, and cancel service. Throughout the computation of the goal, the action server tracks the state of the goal (which is either *Executing*, *Canceling*, *Canceled*, or *Finished* after the goal is accepted). The action server sends periodic feedback messages through the feedback topic, allowing the action client to monitor the progress of the goal. Once the goal is fully processed, the result is sent back to the action client via the result service. Importantly, the action client can cancel the goal at any time using the cancel service, prompting the action server to halt computation and return the latest result computed so far. After cancellation, the action server uses the result service to send the intermediate result back to the action client.

We illustrate the communication between an action client and an action server in Figure 3, which includes the goal service, feedback topic, result service, and cancel service. For simplicity, we show one action client and one action server.

We aim to implement ROS 2 actions and our management approach within the single-threaded ROS 2 executor, avoiding unmanaged threads. This allows us to apply existing scheduling analysis techniques for ROS 2 to provide timing guarantees for integrated anytime algorithms. In Section VII, we present two concrete architectures that leverage ROS 2 actions and our task management approach, and discuss their design trade-offs. Then, in Section VIII, we show how our task transformation, management integration, and use of ROS 2 actions affect existing scheduling analysis techniques for ROS 2, allowing us to provide timing guarantees for anytime algorithms integrated into ROS 2. We also analyze the benefits our transformation provides for other tasks in the system.

Preprint Version. Citation Info: XX.XXXX/XXXXXXX

D. Anytime Algorithm Integration Summary

We address subproblems (1)∼(3) (Section IV) to enable the integration of anytime algorithms into ROS 2 as follows:

1. Anytime Computation: We propose splitting the computation of an anytime algorithm into multiple smaller *segments*, scheduling one segment at a time. This keeps the (non-preemptive) processing window short, allowing the executor to sample and process tasks (e.g., cancellations) promptly.

2. Task Management: We suggest using guard conditions to dynamically manage the activation of tasks within the anytime algorithm. Furthermore, we propose a management architecture that coordinates the execution flow of the anytime algorithm, enabling the algorithm to start, cancel, and return intermediate results effectively while maintaining the correct execution flow and minimizing state management overhead.

3. Communication Interface Design: We recommend integrating anytime algorithms with ROS 2 actions. Actions provide a standardized interface for initiating, monitoring, and terminating anytime algorithms, leveraging native ROS 2 features for goal cancellation and feedback querying to enable the effective management of anytime algorithms.

VI. Multi-Threading and Accelerator Extensions

We discuss the integration of anytime algorithms with multi-threaded executors (Section VI-A) and the integration with accelerator devices (Section VI-B), addressing subproblems (4) and (5) from Section IV, respectively.

A. Synchronization in Multi-Threaded Executors

For the single-threaded executor, all tasks are processed sequentially by the same thread, and hence no synchronization between the tasks is needed. The ROS 2 multi-threaded executor enables parallel execution of tasks, potentially increasing overall system performance. However, this comes at the cost of increased complexity, as concurrently running tasks may access shared data structures, requiring proper synchronization mechanisms to ensure data integrity.

In our case, the shared data structures entail the *internal state* of the algorithm and the *intermediate results* that are computed by the *computation* tasks. Thus, we need to ensure that these tasks are synchronized to avoid race conditions. Furthermore, the external inputs to the *management task*, such as *start*, *cancel*, *return*, and *finish*, need to be synchronized to ensure that the action server state is consistent. As we solely rely on the ROS 2 executor, without the use of external threads, we focus on synchronization using native ROS 2 mechanisms.

We synchronize in multi-threaded executors using ROS 2 callback groups (see Section III). They ensure sequential processing of tasks in the same group. In the following, we discuss at a general level which tasks should be grouped and how shared data structures can be synchronized, as the exact implementation depends on the concrete algorithm in use.

Proposed Solution: We use callback groups to (1) keep the action server state consistent and (2) synchronize the shared data structures efficiently while enabling parallel execution.

The *management callback group* includes all tasks that change the action server state and must be processed sequentially to ensure consistency, i.e., the action server tasks processing the external inputs *start*, *cancel*, *return*, and *finish*.

Furthermore, there are three *computation* tasks that access two shared data structures: ***compute*** reads from and writes to the *internal state*, ***result*** reads from the *internal state* and writes to the *result data*, and ***return*** reads from the *result data*.

We now present two approaches to group the *computation tasks*, each with different synchronization trade-offs.

1) Each task is assigned to a separate callback group. Hence, the tasks can run in parallel, which is especially useful if the *segments* and *result computation* are computationally intensive. However, this approach requires a separate synchronization mechanism, external to the ones provided by ROS 2, for the *internal state* and the *result data*.
   The *internal state* is highly algorithm-dependent and may include complex data structures. Thus, we suggest using this approach only if the *internal state* is simple enough to be synchronized using common techniques.
   The *result data*, which includes the content that is returned when the algorithm is finished or canceled, can be stored as a ROS 2 message, making it readily available whenever the algorithm finishes. Synchronization can then be enabled using common techniques such as double buffering or atomic data structures.
2) Alternatively, the *compute* and *result* tasks are assigned to one callback group, while the *return* task is part of a separate callback group. Thus, the potentially complex data structures of the *internal state* can be safely accessed without additional synchronization mechanisms. Meanwhile, the *result data* can still be synchronized using common techniques, as described in the first approach.

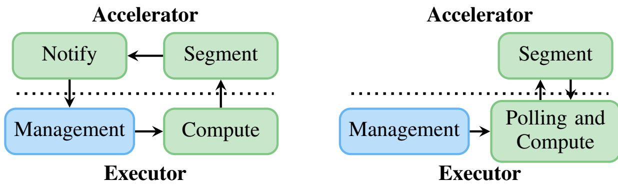
Depending on the algorithm in use, a case-by-case evaluation is required to determine which approach is more suitable and provides the better performance.

Summary: We use callback groups to ensure the state of the action server is consistent and shared data structures are safe from concurrent accesses. Furthermore, we propose how to split the tasks into separate callback groups and how to synchronize shared data structures efficiently.

B. Synchronization with Accelerators

Integrating anytime algorithms that utilize accelerator devices (such as GPUs, FPGAs, or TPUs, a common setting for machine-learning algorithms) in ROS 2 requires consideration of the interaction mechanisms that are available between the CPU and the accelerator. We examine suitable task activation and notification mechanisms and their implications for the integration of anytime algorithms with accelerator devices. Specifically, we discuss suitable task activation and notification mechanisms for blocking calls as well as for non-blocking calls with cooperative and passive notification mechanisms.

Blocking calls: The CPU's executor thread is blocked for each segment on the accelerator and released once it completes, simplifying synchronization but potentially idling the CPU.



(a) Cooperative: the accelerator notifies the executor.

(b) Passive: the executor periodically polls the accelerator status.

Fig. 4: Notification mechanisms for accelerator integration.

Non-blocking calls: The executor thread on the CPU does not wait for the accelerator to complete its computations. Thus, the executor thread can handle other tasks, including cancellations and returning results, in the meantime. However, this approach requires a mechanism to detect when a segment on the accelerator completes, to avoid large synchronization delays. We discuss two notification mechanisms for non-blocking calls: cooperative and passive notification, as illustrated in Figure 4.

- *Cooperative*: The accelerator actively signals the completion of a segment that the executor can respond to. Specifically, guard conditions can be used by passing a reference of the guard notification and its trigger mechanism as an input to the accelerator, allowing it to directly notify the executor upon completion.
- *Passive*: A timer periodically polls the status of the accelerator and retrieves the result when the computation is complete. While simpler, this approach may lead to large delays or inefficiencies if the polling interval is not well aligned with the computation time of the accelerator.

Summary: We discussed the synchronization mechanisms between the CPU and accelerator devices. We differentiated between blocking and non-blocking calls and discussed the implications of using non-blocking calls. Furthermore, we considered cooperative and passive notification mechanisms for non-blocking calls and discussed their implications for the integration of anytime algorithms with accelerator devices.

VII. System Design Trade-Offs

Multiple design choices impact the system's efficiency and responsiveness when integrating anytime algorithms in ROS 2. We discuss the trade-offs, focusing on the *result computation strategy*, *task granularity*, and *scheduling interference*.

Result Computation Strategy. We differentiate between *proactive* and *reactive* strategies when computing the results.

Proactive strategies compute the results after a fixed number of segments (e.g., every 5th segment). This precomputed result can be returned almost immediately after cancellation, improving the responsiveness of the system. Importantly, a proactive strategy is strictly necessary if the calculation should be canceled based on the quality of the result, e.g., when a certain object is detected. One drawback of proactive strategies is that they may lead to a decrease in efficiency, as some results are potentially never used and discarded. Furthermore, additional memory is needed to store the precomputed result.

Reactive strategies compute the result on demand (i.e., after cancellation or finish), which is more efficient and requires less memory. However, reactive strategies are less responsive, as the result is only computed after cancellation.

Task Granularity. The task granularity depends on the length of a single segment and the number of segments per job. Two types of latencies have to be considered: (1) the scheduling latency of the ROS 2 *executor*, and (2) the computation latency of a single job in the anytime algorithm.

Finer granularity improves responsiveness, as the anytime algorithm can react faster to cancellation requests. Yet, the scheduling overhead increases compared to the execution time, as the executor must switch between tasks more frequently, which may introduce additional delays. Conversely, coarser granularity reduces the scheduling overhead but also the responsiveness, as the processing windows are longer. Moreover, this affects not only the cancellation delay, but also the frequency at which other system tasks are executed. Notably, our implementation exposes the granularity (i.e., the block size) as a ROS 2 parameter, allowing it to be adjusted at runtime without recompilation. This enables developers to inject custom adaptation logic, e.g., to dynamically balance responsiveness and overhead based on system load or application requirements. In this paper, we focus on static block sizes to isolate and evaluate the core trade-offs of our approach.

Scheduling Interference. In ROS 2 systems, tasks are typically distributed across multiple executors, each managing a subset of the system's workload. Specifically, as the number of tasks assigned to an executor increases, the length of the processing window increases as well. As a result, (1) the system's responsiveness is reduced, as the cancellation request delays may increase, and (2) the interference between tasks increases due to the processing window, as discussed in Section V-A. Thus, we suggest assigning the anytime algorithm to a dedicated executor to simplify the design and provide more predictable latencies. However, in cases where the system has limited resources, sharing an executor may be necessary.

Ultimately, executor-level design involves trade-offs between responsiveness and efficient CPU utilization. Our design space mapping provides a foundation for analyzing these trade-offs, which we evaluate in Section IX.

VIII. Latency Guarantees

In this section, we provide formal guarantees on the timing behavior of ROS 2 systems that integrate anytime algorithms using our proposed design. We start with the system model, detail the transformation of a long-running task, and analyze the timing implications for the entire system.

We consider a ROS 2 system composed of a set of tasks $\tau = \{\tau_1, \ldots, \tau_n\}$. Each task τ_i has a worst-case execution time C_i and is statically assigned to a ROS 2 node and its corresponding executor. Tasks communicate via ROS 2 topics or services (inter-node), or via intra-node mechanisms like shared memory or callbacks triggered by guard conditions. A set of cause-effect chains $E = \{E_1, \ldots, E_m\}$ describes the system's functionality, where each E_j is a sequence of tasks.

Preprint Version. Citation Info: XX.XXXX/XXXXXXX

Preprint Version. Citation Info: XX.XXXX/XXXXXXX

We focus on a single long-running task τ_i, which is part of at least one cause-effect chain, and transform it into an anytime algorithm to improve the overall system responsiveness. The original task τ_i is monolithic: it receives an input, performs a lengthy computation, and then produces an output. We replace this single task with a set of smaller, coordinated tasks within the same node, orchestrated by a ROS 2 action. This splits the monolithic computation into a manageable, iterative process. The new set of tasks, replacing τ_i, includes:

- $\tau_{\text{ac-recv}}$: An action client callback that receives the initial input and sends a goal to the action server.
- $\tau_{\text{as-goal}}$: An action server callback that receives the goal and initiates the computation.
- $\tau^{(k)}_{\text{as-comp}}$: A series of tasks, where $\tau^{(k)}_{\text{as-comp}}$ represents the k-th segment of the computation. The number of segments N can be either fixed or dynamic.[4]
- $\tau_{\text{as-result}}$: An action server callback that returns the final result upon completion.
- $\tau_{\text{ac-result}}$: An action client callback that receives the final result and sends it to the next task in the original chain.

This transformation refactors the single execution of τ_i into a sequence of smaller, interconnected task executions. Importantly, every task except the computation segments $\tau^{(k)}_{\text{as-comp}}$ for $k \in [1, N]$ has negligible execution time and scheduling overhead, as they primarily handle internal communication and coordination between the action client and action server.

This transformation has a significant impact on the timing behavior of the entire system, affecting both the transformed chain and all other chains sharing the same executor. We analyze its effects on chains that either include or do not include the transformed task, based on the end-to-end timing analysis for multi-executor ROS 2 systems by Teper et al. [53].

Impact on Unrelated Task Chains. For any task τ_j that is not part of a chain involving the original τ_i, our transformation can reduce the interference it suffers from τ_i and hence the worst-case end-to-end latency of chains including τ_j. Teper et al. [53] bound the interference a task experiences by two terms: (1) ub^{pre}_j, upper-bounding the time between the release and the start of the task's execution, and (2) ub^{exe}_j, upper-bounding the execution time of the task itself once it starts executing. While the latter term is unaffected by our transformation, the former term can be significantly reduced, since it depends on the length of the processing windows in the executor.

Specifically, by splitting the long-running task τ_i (with execution time C_i) into multiple smaller tasks (e.g., $\tau^{(k)}_{\text{as-comp}}$ with execution time $C^{(k)}_{\text{as-comp}}$), the execution time of any single new task is smaller than C_i. Assuming the execution times of the other new tasks are negligible, our transformation thus reduces the maximum length of the processing windows, thereby lowering the potential interference for all other tasks in the system. Consequently, in most cases, the end-to-end latencies for task chains not involving τ_i will decrease.

[4]For other task designs for the compute and result functions (reactive versus proactive from Section VII), this design can be adapted accordingly.

However, a caveat to this improvement is the task prioritization, which may affect the interference in the analysis. For our transformation, we introduce new tasks, including service servers and clients, as well as the task $\tau^{(k)}_{\text{as-comp}}$ that is triggered by a guard condition, which, by the default design of the ROS 2 executor, have lower priorities than subscription tasks (see Section V-A). Consequently, for any task τ_j, these new, lower-priority tasks are now introduced as a source of interference that was not present in the original task set, potentially increasing the interference bound ub^{pre}_j.

Concretely, in the analysis by Teper et al. [53], these prioritization changes can increase the pessimistic interference bound. Their analysis uses bounds (e.g., ub^{pre}_j) that sum the execution times of all higher-priority ($C^{\tau}_{j,hp}$) and relevant lower-priority ($C^{\tau}_{j-1,lp}$) tasks. Because our transformation adds new, lower-priority tasks, these are newly included in the $C^{\tau}_{j-1,lp}$ term, increasing the calculated worst-case bound.

Impact on Anytime Task Chain. Conversely, for any cause-effect chain that originally included τ_i, its end-to-end latency will increase. Any such chain, say $(\ldots, \tau_{\text{prev}}, \tau_i, \tau_{\text{next}}, \ldots)$, is replaced by a much longer sequence of tasks. A simplified representation of the new chain for a full, non-canceled execution is: $E_{\text{new}} = (\ldots, \tau_{\text{prev}}, \tau_{\text{ac-recv}}, \tau_{\text{as-goal}}, \tau^{(1)}_{\text{as-comp}}, \ldots, \tau^{(N)}_{\text{as-comp}}, \tau_{\text{as-result}}, \tau_{\text{ac-result}}, \tau_{\text{next}}, \ldots)$. This expansion introduces overhead in two ways. First, each of the new scheduling events incurs its own scheduling latency and communication overhead. Second, each smaller task can be delayed by interference, resulting in a higher worst-case end-to-end latency bound for the transformed chain. This is a deliberate trade-off: we accept a higher worst-case end-to-end latency bound for chains including τ_i, but enable significantly improved responsiveness for all other chains in the system that are no longer blocked by the long-running task τ_i.

A key benefit of our design is the ability to trade execution time for result quality. We can formally define the quality of the result $Q(t)$ as a function of the computation time t. Given that the quality improves after each computation segment, we can model $Q(t)$ as an increasing step function. If $L(\tau)$ denotes the worst-case end-to-end latency of a task τ, then the time to achieve quality Q_k (i.e., after completing the k-th segment) is upper-bounded by $\sum_{j=1}^{k} L(\tau^{(j)}_{\text{as-comp}})$ relative to the start of the computation. This provides a formal mapping from the time invested to the minimum quality of the result.

In the event of a cancellation, a different task chain is executed. This is initiated by a trigger task τ_{trigger} and proceeds as follows: $E_{\text{cancel}} = (\ldots, \tau_{\text{trigger}}, \tau_{\text{ac-send-cancel}}, \tau_{\text{as-recv-cancel}}, \tau_{\text{ac-req-result}}, \tau_{\text{as-send-result}}, \tau_{\text{ac-recv-result}}, \ldots)$, with potentially additional tasks following. The total cancellation latency is then the sum of the worst-case response times of the tasks in this chain. This provides an upper bound on the time from when a cancellation is initiated to when the latest available intermediate result is processed. While this cancellation chain involves many tasks, assigning the anytime algorithm to a dedicated executor can keep the cancellation latency low enough to meet responsiveness requirements.

IX. Implementations and Evaluations

We evaluated the proposed framework by integrating two anytime algorithms into ROS 2: (1) CPU-bound RRT* path planning (Section IX-A) and its impact on the responsiveness of other tasks on the same executor (Section IX-B), and (2) GPU-bound AnytimeYOLO object detection (Section IX-C). Our experiments aim to answer the questions:

- How fast can ROS 2 anytime algorithms return intermediate results after a cancellation?
- How efficient are ROS 2 anytime algorithms in utilizing computational resources?
- How can we implement flexible cancellation conditions based on non-linear quality metrics?
- How does the configuration of our anytime algorithm affect the responsiveness of the overall system?

We evaluated single- and multi-threaded ROS 2 executors (Section VI-A), as well as synchronous and cooperative asynchronous execution for GPU-bound accelerator integration (Section VI-B). We also evaluated the impact of *proactive* and *reactive* architectures (Section VII) on system performance.

We implemented our architectures using modular template-based C++ classes in ROS 2, including guard-condition-based task activation and callback group configuration for the multi-threaded executor, selectable at compile time. It is based solely on the interfaces and task types provided directly by ROS 2, and is a generic solution to integrate most anytime algorithms, *requiring no change to the underlying ROS 2 codebase.* We added custom tracepoints using ROS 2's tracing framework [4] to measure the timing behavior of our architectures with minimal overhead. Across all experiments, the communication and infrastructure overhead of our anytime framework is less than 1% of the total computation time. Our source code is publicly available[5], including all experiment configurations, evaluation scripts, and detailed documentation.

We implemented our experiments using ROS 2 Humble [41] with CycloneDDS [16] as the DDS implementation. The experiments were conducted on an Nvidia Orin NX with 16 GB of shared memory, using Ubuntu 22.04 (Jetpack 6.2).

A. RRT* Path Planning

RRT* [24] (Rapidly-exploring Random Tree Star) is a sampling-based path motion planning algorithm that incrementally builds a space-filling tree while asymptotically converging to the globally optimal path. We implement the anytime variant proposed by Karaman et al. [25], which extends RRT* with branch-and-bound pruning to actively focus tree growth on the most promising regions. Each iteration samples a random point in the configuration space, steers toward it from the nearest tree node, and, if the resulting edge is collision-free, inserts the new node by choosing the parent that minimizes the overall path cost, then rewires neighboring nodes to reduce path costs. The algorithm is thus inherently anytime: it can be interrupted at any point in time to return the best path found so far, while the solution quality improves as the tree is refined.

[5]https://github.com/tu-dortmund-ls12-rt/Anytime-Development

Integration with the Anytime Framework. We integrated RRT* following the management architecture in Figure 2. Each block of RRT* iterations maps to one *segment*, and the block size determines how many iterations are processed per job. The mapping to the framework's functions is as follows:

- **Compute:** Executes a block of RRT* iterations (one job), extending the tree by sampling, steering, collision checking, and rewiring nodes.
- **Result:** Extracts the current best path and its associated cost from the search tree.
- **Return:** Sends the best path cost, total iterations, block size, and per-block computation time back via the action's result or feedback interface.
- **Internal State:** The full RRT* search tree, including all nodes, edges, and the current best solution, reset upon receiving a new goal request.
- **Result Data:** The current best path cost and the total number of completed iterations.

We evaluated RRT* on two occupancy grid maps from Navigation 2 (Nav2) [39][6], a widely-used navigation framework for ROS 2: a *depot* ($30.2 \times 15.4\,\mathrm{m}$) and a *warehouse* ($30.2 \times 50.2\,\mathrm{m}$), with the start and destination placed at two static positions at opposing ends of each map. The algorithm uses branch-and-bound pruning every 1000 iterations, 5% goal bias, and a step size of $0.5\,\mathrm{m}$, as described in [25], with collision checking via ray-tracing on the occupancy grid.

We evaluated the following parameters:

- **Block Size**: 1, 16, 64, 256, 1024, 4096 iterations per job.
- **Architecture**: Either proactive or reactive.
- **Executor-Threading**: Either single-threaded or multi-threaded (two threads)[7].

Each configuration ran for $10\,\mathrm{s}$ over 5 runs each. The action client sent a goal request to the action server every $500\,\mathrm{ms}$. After $\underline{200\,\mathrm{ms}}$, the action client sent a cancellation request to the action server. We evaluated the following two metrics:

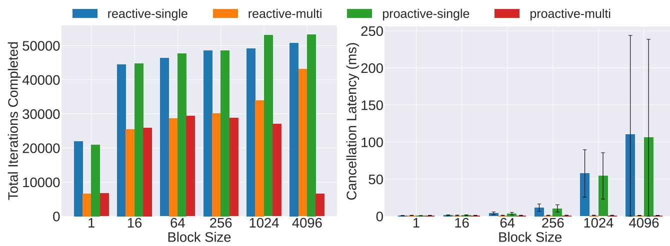
- **Iteration Count**: The number of RRT* iterations computed before the cancellation request is received. Higher iteration counts indicate that more work is done, suggesting greater efficiency and less framework overhead.
- **Cancellation Delay**: The time between sending the cancellation request and receiving the final result. Lower values indicate higher responsiveness.

Results. The results are shown in Figure 5. For the *iteration count* in Figure 5a, the number of completed iterations increases with block size, confirming that larger blocks reduce the relative impact of the framework's scheduling overhead by processing more iterations per job. For small block sizes, the framework overhead dominates the total cycle time, leaving little time for actual RRT* computation. The multi-threaded executor suffers from a lower iteration count than the single-threaded executor due to higher per-block overhead from managing multiple threads and callback groups.

[6]https://github.com/ros-navigation/navigation2

[7]For single-threaded, action server and client are assigned to separate executors, while for multi-threaded, they are assigned to the same executor.

Preprint Version. Citation Info: XX.XXXX/XXXXXXX



(a) Total number of computed iterations before cancellation.

(b) Latency from cancel request to result reception.

Fig. 5: RRT* path planning results across executor (single-/multi-threaded) and architecture (proactive/reactive) configurations. (a) The single-threaded executor is more efficient due to lower executor overhead. (b) The multi-threaded executor maintains near-constant low cancellation delay while the single-threaded executor's delay increases with block size.

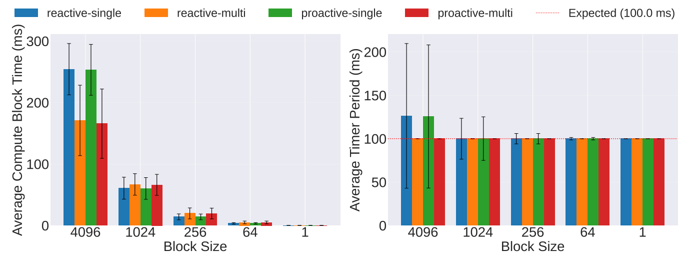
Regarding the *cancellation delay* in Figure 5b, larger block sizes lead to higher cancellation delays for the single-threaded executor, as the executor thread cannot process the cancellation request until the current block completes. Meanwhile, the multi-threaded executor maintains a near-constant low cancellation delay, as the cancellation request can be processed on a separate thread in parallel with the computation.

However, for very large block sizes, the cancellation may arrive mid-block. In the reactive architecture, the result includes the iterations of the partial block that was still running when the cancellation arrived. In the proactive architecture, results are computed only after each complete block; if no block has completed before the cancellation, the algorithm returns no path. For example, in the proactive multi-threaded configuration with block size 4096, the first block sometimes takes more than 200 ms to complete and is therefore canceled, as shown by the lower iteration count in Figure 5a. This illustrates the task granularity trade-off (Section VII): the block size must balance efficiency against responsiveness.

B. Interference on Other Tasks

To evaluate how the block size affects the responsiveness of other tasks on the same executor, we added a ROS 2 wall timer with a period of 100 ms and a simulated workload of 10 ms per callback to the same executor as the RRT* action server. We used the same architectures and executor configurations as in Section IX-A, but ran only the depot map with block sizes 1, 64, 256, 1024, and 4096, for 10 runs of 30 s each.

As discussed in Section V-A, long-running tasks can cause permanent loss of timer jobs, as the executor dispatches only a single job after a long block. This experiment quantifies this effect and shows how our task-splitting approach mitigates it.
Results. The results are shown in Figure 6 and Table I. For the multi-threaded executor, a second thread is almost always available to process the timer callback, since the remaining framework tasks have negligible workload, yielding 0% skipped timer jobs across all block sizes (Table I).



(a) Computation time per block.

(b) Observed timer period.

Fig. 6: Impact of block size on task interference using RRT*. (a) Per-block computation time increases with block size. (b) For the single-threaded executor, larger block sizes cause the observed timer period to deviate from the expected 100 ms, while the multi-threaded executor remains unaffected.

TABLE I: Percentage of skipped timer jobs for different block sizes, architectures, and executor configurations. The multi-threaded executor never misses timer jobs, as a second thread is available. The single-threaded executor loses timer jobs when the block's computation time exceeds the timer period.

Configuration	4096	1024	256	64	1
Proactive (multi)	0.0%	0.0%	0.0%	0.0%	0.0%
Proactive (single)	24.97%	4.07%	0.0%	0.0%	0.0%
Reactive (multi)	0.0%	0.0%	0.0%	0.0%	0.0%
Reactive (single)	25.11%	3.65%	0.0%	0.0%	0.0%

For the single-threaded executor, the timer and anytime algorithm have to share the same executor thread. With large block sizes, the per-block computation time exceeds the timer period (Figure 6a), causing the timer to deviate from its expected 100 ms period (Figure 6b) and permanently lose timer jobs. At block size 4096, approximately 25% of timer jobs are skipped, decreasing to 4% at block size 1024 (Table I). As we reduce the block size, the interference decreases: for block sizes up to 256, the per-block computation time stays below the timer period and no timer jobs are skipped.

In summary, our task-splitting approach is effective at reducing interference and keeping other tasks on the same executor responsive. The block size must be chosen such that the per-block computation time does not exceed the periods of other tasks sharing the same executor.

C. AnytimeYOLO

AnytimeYOLO [28] is an anytime variant of the popular YOLO (You Only Look Once) object detection algorithm [44]. YOLO processes an input image through a sequence of neural network layers on the GPU, where each layer refines the feature representation. AnytimeYOLO exploits this sequential structure by attaching exit modules at intermediate layers, enabling early termination with a valid detection result before the full network has been evaluated. For our evaluation, we use the medium variant with 22 computational layers. The goal of our experiment is to demonstrate how the anytime framework integrates a GPU-bound algorithm into ROS 2 and enables dynamic cancellation based on the intermediate result quality.

Preprint Version. Citation Info: XX.XXXX/XXXXXXX

Preprint Version. Citation Info: XX.XXXX/XXXXXXX

Integration with the Anytime Framework. We integrate AnytimeYOLO following the management architecture in Figure 2. Each layer corresponds to one iteration, and a configurable block of layers maps to one *segment*. The block size parameter determines how many layers are processed per job. The mapping to the framework's functions is as follows:

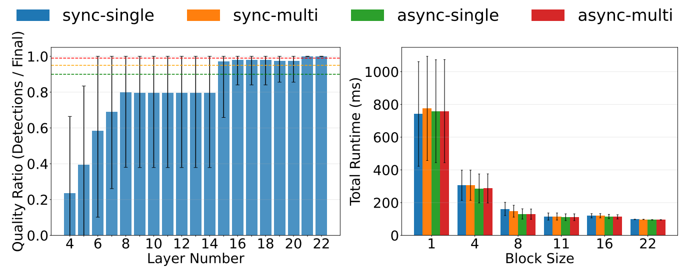
- **Compute:** Executes a block of YOLO layers (one job), advancing the network's inference state.
- **Result:** Runs the exit and post-processing modules on the current layer activations to produce object detections with bounding boxes and confidence scores.
- **Return:** Sends the current detections back to the client via the action's result or feedback interface.
- **Internal State:** The inference state and GPU tensors holding layer activations, reset and initialized with the preprocessed input image upon receiving a new goal.
- **Result Data:** The post-processed detections with bounding boxes, class IDs, and confidence scores.

We adopt the following workflow:

- We use the MS COCO validation dataset [35], specifically the 191 images that contain *traffic lights*.
- We determine the execution time of each layer and each exit module to assess its computational cost.
- The quality ratio progression per layer for *traffic lights* (compared to the full network's final result) identifies how many layers are needed to reach a given target quality. Figure 7a shows that the quality ratio plateaus at approximately 0.80 from layers 8–14, then jumps to approximately 0.97 at layer 15, with reduced variance by layer 16, crossing both the 90% and 95% quality thresholds. Near-full quality (99%) is reached by layer 21.
- Based on these observations, we evaluate block sizes of 1, 4, 8, 11, 16, and 22 layers: block sizes 4 and 8 capture the early quality boundaries, block size 11 covers the pre-jump plateau region, block size 16 benefits from the quality jump and its low variance, and block size 22 runs the full model as a reference baseline.
- We ran AnytimeYOLO with these configurations to evaluate the total runtime and assess performance. Each configuration was evaluated over 5 runs using the 191 traffic light images from the MS COCO validation dataset.

ROS 2 Integration and Cancellation. We integrate AnytimeYOLO into ROS 2 using the architectures in Section VII. To dynamically adapt computation, we must assess intermediate result quality, which requires the **proactive architecture**.

To realize the dynamic cancellation, we trigger it based on concrete detection results. Specifically, we cancel the action if *traffic lights* are detected with an estimated accuracy of 80% or more. We include the detected object IDs and confidence scores in the action's *feedback* message, which the server sends back to the client after each block is processed. The client checks the feedback against the cancellation condition and issues a cancellation request once the threshold is met, after which the server returns the current best result.



(a) Detection quality ratio per processed layer. (b) Total runtime for different block sizes.

Fig. 7: AnytimeYOLO evaluation on traffic light detection. (a) Detection quality ratio per layer, plateauing at 0.80 for layers 8–14 and jumping to 0.97 at layer 15. (b) Total runtime comparison across block sizes, showing that block sizes of 8 and above achieve runtimes close to the full 22-layer model.

We evaluate synchronous (*sync*) and asynchronous (*async*) execution modes (see Section VI-B). In the synchronous case, the executor thread blocks until the GPU finishes the current block computation. In the asynchronous case, a notification mechanism informs the executor when the GPU completes its work. We use a *cooperative* approach: we enqueue a host callback function into the CUDA stream, which triggers the action server's guard condition once the GPU completes the current segment.[8] We do not evaluate the *passive* polling mechanism, as it performs worse than this cooperative approach.

Results. Figure 7b shows that configurations with block sizes of 8 layers and above achieve total runtimes comparable to the full 22-layer model, while block size 1 incurs significant overhead due to per-layer exit computation and framework scheduling. The asynchronous mode is particularly affected at block size 1, as each layer requires a cooperative notification round-trip (CUDA host callback, guard condition activation, executor dispatch); for the multi-threaded async configuration, additional thread synchronization overhead further increases the per-layer cost. While quality-based cancellation does not reduce the total computation time compared to the full 22-layer model, it demonstrates the system's ability to dynamically adapt the computation based on the intermediate result quality.

X. Discussions and Future Research

In this section, we highlight several research directions enabled by our contribution and discuss their relation to existing challenges in ROS 2 development.

Limited Preemption in ROS 2. Most prior work on ROS 2 scheduling [9], [11], [12], [54], [55] is constrained by the non-preemptive nature of the executor, where individual callbacks run to completion once started. While preemption in real-time systems broadly refers to the ability to interrupt, pause, or cancel a running task, ROS 2 does not natively support this functionality at the callback level.

[8]This host callback is processed by a separate, unmanaged CUDA thread, as there is no technical possibility to directly notify the ROS 2 executor from within the CUDA stream, due to CUDA's design.

Preprint Version. Citation Info: XX.XXXX/XXXXXXX

Our work introduces limited preemption semantics within ROS 2 by transforming long-running tasks into sequences of short, interruptible jobs and managing their execution via guard conditions and ROS 2 actions. This allows tasks to be suspended or canceled at predefined preemption points, providing practical preemption behavior while staying within the constraints of the ROS 2 execution model.

This mechanism opens new avenues for scheduling analysis in ROS 2. Instead of reasoning about monolithic worst-case execution times, future work can focus on cancellation delays, enabling tighter bounds by reasoning about segment-level rather than task-level execution times. These capabilities are particularly relevant for dynamic-priority scheduling and overload control scenarios. Furthermore, the reduced blocking times per processing window can mitigate priority inversion effects inherent to the non-preemptive executor, as the maximum blocking time decreases with finer task granularity.

Importantly, our approach operates entirely at the application level without modifying the underlying ROS 2 infrastructure. It is therefore complementary to alternative ROS 2 executors, such as PICAS [12] and the rclc-executor [5], which also rely on non-preemptive callback execution. These executors could similarly benefit from our proposed task-splitting approach to reduce callback-level blocking times.

Execution and Communication Synchronization. A related and important direction is the synchronization of concurrent tasks in ROS 2. It has been shown that ROS nodes can be synchronized to run at different frequencies to reduce the timing differences between nodes [47]. Our approach can be extended to synchronize the execution of concurrent workloads by parallelizing multiple computations using the proposed anytime task management. Given our dynamic task-activation mechanisms, synchronizing the activation of tasks across different nodes is readily possible. This is especially relevant in the context of timing misalignment [27], which can affect overall system performance and predictability.

A closely related challenge in ROS and ROS 2 is message synchronization [31], [32], [34], [51], [57], [58]. Our work enables message synchronization by actively controlling the flow of messages during the computation. While traditional ROS 2 message synchronization focuses on aligning messages through buffering mechanisms, anytime algorithms introduce a dynamic aspect that can be used to control the rate of message processing and publication in an online manner. By adjusting the computational time through the anytime property, the output timing of messages can be controlled to adapt to the message flow of other nodes. This dynamic control over computation time provides a new dimension for message synchronization that complements existing approaches.

Accelerator Latency Guarantees. When leveraging accelerators, such as GPUs or FPGAs, the timing behavior of these devices must be considered. To this end, previous work on accelerator scheduling in ROS 2 [17], [33] can be used to both determine latency bounds on task chains involving GPUs and apply prioritization mechanisms to ensure that the anytime algorithm is executed in a timely manner.

XI. Related Work

Anytime algorithms offer a flexible approach to handling compute-intensive tasks, enabling adaptive use of computational resources. In contrast, time-budgeted approaches like contract algorithms [46] provide latency guarantees for multi-stage processing pipelines by assuming a predetermined budget for each task [20], [45]. Recently, there has been growing interest in anytime machine-learning algorithms for real-time systems [3], [7], [21]–[23], [30], [36], [37], [49], [50]. Specifically, such algorithms are applied to perception tasks such as object detection and classification. The anytime property can be achieved by early exits for inference in neural networks, e.g., [10], [29], [30], [49], [50], [52]. Other possibilities involve inspecting a smaller input and altering the processing order of the data, e.g., [3], [22], [23], [36], [37], [49], [50]. Additionally, widely-used optimization [20] and iterative approximation algorithms, like route planning [13], [40], [45], also demand significant computational resources.

ROS 2. The original Robot Operating System [42], released in 2007, was not designed for real-time use. Its successor ROS 2 [38], introduced in 2018, provides a more flexible and modular framework. Several studies have addressed the real-time analysis of ROS 2. Blass et al. [9] established response-time bounds for ROS 2 tasks that propagate data through the system. Teper et al. [53], [54] conducted a formal analysis of end-to-end latencies in ROS 2, which is essential for data propagation analysis in ROS 2. Furthermore, there have been extensions to ROS 2 to support preemptive or dynamic scheduling without relying on the native ROS 2 executor [1], [2], [56]. Moreover, in a recent result, Teper et al. [55] showed the standard ROS 2 executor is prone to starvation.

XII. Conclusion

In this paper, we proposed a structured approach for integrating computationally expensive, long-running tasks into ROS 2 by splitting them into smaller, manageable segments. This transformation addresses the high interference and lack of early termination caused by the non-preemptive ROS 2 executor, and naturally facilitates *anytime algorithms* by creating preemption points where tasks can be canceled.

We leveraged native ROS 2 mechanisms, using guard conditions for task management and ROS 2 actions to provide an effective interface for starting, canceling, and receiving feedback. We explored the design trade-offs involved, such as task granularity and proactive versus reactive result computation. The effectiveness of our approach is demonstrated by evaluating system architectures and practical applications, showing that timely results can be achieved via flexible cancellations when design choices such as block size and architecture are carefully tuned. Crucially, our solution requires **no modification to the existing ROS 2 codebase** and remains compatible with existing ROS 2 end-to-end latency analyses. Furthermore, our work enables several research directions, including limited preemptive scheduling in ROS 2, mitigation of timing misalignments, and any-reason cancellation mechanisms.

Preprint Version. Citation Info: XX.XXXX/XXXXXXX

Acknowledgements

This work has received funding by the German Federal Ministry of Education and Research (BMBF) in the course of the 6GEM research hub under grant number 16KISK038.

This result is part of a project (PropRT) that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 865170).

This work is partially supported by NSF grant CPS 2521121, and the Humboldt Fellowship.

References

[1] A. Al Arafat, K. Wilson, K. Yang, and Z. Guo. Dynamic priority scheduling of multithreaded ros 2 executor with shared resources. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(11):3732–3743, 2024.

[2] A. A. Arafat, S. Vaidhun, K. M. Wilson, J. Sun, and Z. Guo. Response time analysis for dynamic priority scheduling in ROS2. In *ACM/IEEE Design Automation Conference (DAC)*, 2022.

[3] S. Bateni and C. Liu. Apnet: Approximation-aware real-time neural network. In *Real-Time Systems Symposium (RTSS)*. IEEE, 2018.

[4] C. Bédard, I. Lütkebohle, and M. Dagenais. ros2_tracing: Multipurpose low-overhead framework for real-time tracing of ROS 2. *IEEE Robotics and Automation Letters*, 7(3):6511–6518, 2022.

[5] K. Belsare, A. C. Rodriguez, P. G. Sánchez, J. Hierro, T. Kołcon, R. Lange, I. Lütkebohle, A. Malki, J. M. Losa, F. Melendez, M. M. Rodriguez, A. Nordmann, J. Staschulat, and J. von Mendel. *Micro-ROS*, pages 3–55. Springer International Publishing, Cham, 2023.

[6] T. Betz, M. Schmeller, A. Korb, and J. Betz. Latency measurement for autonomous driving software using data flow extraction. In *Intelligent Vehicles Symposium (IV)*, pages 1–8. IEEE, 2023.

[7] J. Bian, A. A. Arafat, H. Xiong, J. Li, L. Li, H. Chen, J. Wang, D. Dou, and Z. Guo. Machine learning in real-time internet of things (iot) systems: A survey. *IEEE Internet of Things Journal*, 9(11):8364–8386, 2022.

[8] G. Biggs, J. Perron, and S. Loretz. ROS 2 actions, 2019. https://design.ros2.org/articles/actions.html, accessed 2026.

[9] T. Blass, D. Casini, S. Bozhko, and B. B. Brandenburg. A ROS 2 response-time analysis exploiting starvation freedom and execution-time variance. In *Real-Time Systems Symposium (RTSS)*. IEEE, 2021.

[10] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han. Once-for-all: Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations (ICLR)*, 2020.

[11] D. Casini, T. Blass, I. Lütkebohle, and B. B. Brandenburg. Response-time analysis of ros 2 processing chains under reservation-based scheduling. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2019.

[12] H. Choi, Y. Xiang, and H. Kim. Picas: New design of priority-driven chain-aware scheduling for ROS2. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 251–263. IEEE, 2021.

[13] T. Dam, G. Chalvatzaki, J. Peters, and J. Pajarinen. Monte-carlo robot path planning. *arXiv preprint arXiv:2208.02673*, 2022.

[14] T. L. Dean and M. S. Boddy. An analysis of time-dependent planning. In *AAAI Conference on Artificial Intelligence*, 1988.

[15] F. Dellaert, D. Fox, W. Burgard, and S. Thrun. Monte carlo localization for mobile robots. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, volume 2, pages 1322–1328, 1999.

[16] Eclipse Foundation. Eclipse cyclone dds, 2022. https://cyclonedds.io/, accessed 2026.

[17] D. Enright, Y. Xiang, H. Choi, and H. Kim. Paam: A framework for coordinated and priority-driven accelerator management in ROS 2. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 81–94. IEEE, 2024.

[18] C. Fan, L. Nie, J. Zhang, K. Dai, S. Gao, and J. Li. Uncovering underexplored runtime behaviors in ROS2-based autonomous systems. *ACM Trans. Internet Things*, Oct. 2025.

[19] J. W. Grass and S. Zilberstein. Anytime algorithm development tools. *SIGART Bull.*, 7:20–27, 1996.

[20] N. J. A. Harvey, C. Liaw, E. Perkins, and S. Randhawa. Optimal anytime regret with two experts, 2021. arXiv preprint arXiv:2002.08994.

[21] S. Heo, S. Cho, Y. Kim, and H. Kim. Real-time object detection system with multi-path neural networks. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2020.

[22] Y. Hu, I. Gokarn, S. Liu, A. Misra, and T. Abdelzaher. Algorithms for canvas-based attention scheduling with resizing. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2024.

[23] W. Kang, S. Chung, J. Y. Kim, Y. Lee, K. Lee, J. Lee, K. G. Shin, and H. S. Chwa. Dnn-sam: Split-and-merge dnn execution for real-time object detection. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2022.

[24] S. Karaman and E. Frazzoli. Sampling-based algorithms for optimal motion planning. *The International Journal of Robotics Research*, 30(7):846–894, 2011.

[25] S. Karaman, M. R. Walter, A. Perez, E. Frazzoli, and S. Teller. Anytime motion planning using the RRT*. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 1478–1483, 2011.

[26] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi. Autoware on board: Enabling autonomous vehicles with embedded systems. In *ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, April 2018.

[27] D. Kuhse, N. Hölscher, M. Günzel, H. Teper, G. von der Brüggen, J.-J. Chen, and C.-C. Lin. Sync or sink? the robustness of sensor fusion against temporal misalignment. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2024.

[28] D. Kuhse, H. Teper, S. Buschjäger, C.-Y. Wang, and J.-J. Chen. You only look once at anytime (AnytimeYOLO): Analysis and optimization of early-exits for object-detection. *arXiv preprint arXiv:2503.17497*, 2025.

[29] S. Laskaridis, A. Kouris, and N. D. Lane. Adaptive inference through early-exit networks: Design, challenges and directions. In *Proceedings of the 5th International Workshop on Embedded and Mobile Deep Learning*, 2021.

[30] H. Lee and J. Shin. Anytime neural prediction via slicing networks vertically. *arXiv preprint arXiv:1807.02609*, 2018.

[31] R. Li, Z. Dong, J.-M. Wu, C. J. Xue, and N. Guan. Modeling and property analysis of the message synchronization policy in ros. In *International Conference on Mobility, Operations, Services and Technologies (MOST)*, 2023.

[32] R. Li, N. Guan, X. Jiang, Z. Guo, Z. Dong, and M. Lv. Worst-case time disparity analysis of message synchronization in ros. In *Real-Time Systems Symposium (RTSS)*. IEEE, 2022.

[33] R. Li, T. Hu, X. Jiang, L. Li, W. Xing, Q. Deng, and N. Guan. ROSGM: A real-time GPU management framework with plug-in policies for ROS 2. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 93–105. IEEE, 2023.

[34] R. Li, X. Jiang, Z. Dong, J.-M. Wu, C. J. Xue, and N. Guan. Worst-case latency analysis of message synchronization in ros. In *Real-Time Systems Symposium (RTSS)*, pages 185–197. IEEE, 2023.

[35] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft coco: Common objects in context. In *European Conference on Computer Vision (ECCV)*, pages 740–755. Springer, 2014.

[36] S. Liu, X. Fu, M. Wigness, P. David, S. Yao, L. Sha, and T. Abdelzaher. Self-cueing real-time attention scheduling in criticality-aware visual machine perception. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2022.

[37] S. Liu, S. Yao, X. Fu, R. Tabish, S. Yu, A. Bansal, H. Yun, L. Sha, and T. Abdelzaher. On removing algorithmic priority inversion from mission-critical machine inference pipelines. In *Real-Time Systems Symposium (RTSS)*. IEEE, 2020.

[38] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022.

[39] S. Macenski, F. Martín, R. White, and J. G. Clavero. The marathon 2: A navigation system. *arXiv preprint arXiv:2003.00368*, 2020.

[40] R. Mangharam and A. A. Saba. Anytime algorithms for gpu architectures. In *Real-Time Systems Symposium (RTSS)*. IEEE, 2011.

[41] Open Robotics. ROS 2: Humble, 2024. https://docs.ros.org/en/humble, accessed 2026.

[42] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng. ROS: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, volume 3, 2009.

[43] J. B. Rawlings, D. Q. Mayne, and M. Diehl. *Model Predictive Control: Theory, Computation, and Design*. Nob Hill Publishing, 2nd edition, 2022.

[44] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788. IEEE, 2016.

[45] M. Rowold, L. Ögretmen, T. Kerbl, and B. Lohmann. Efficient spatiotemporal graph search for local trajectory planning on oval race tracks. *Actuators*, 11(11), 2022.

[46] S. J. Russell and S. Zilberstein. Composing real-time systems. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, 1991.

[47] Y. Saito, T. Azumi, S. Kato, and N. Nishio. Priority and synchronization support for ros. In *4th International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, 2016.

[48] SOAFEE. SOAFEE: Scalable open architecture for embedded edge, 2022. https://www.soafee.io, accessed 2026.

[49] A. Soyyigit, S. Yao, and H. Yun. Anytime-Lidar: Deadline-aware 3D Object Detection . In *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, Aug. 2022.

[50] A. Soyyigit, S. Yao, and H. Yun. Valo: A versatile anytime framework for lidar-based object detection deep neural networks. *arXiv preprint arXiv:2409.11542*, 2024.

[51] J. Sun, T. Wang, Y. Li, N. Guan, Z. Guo, and G. Tan. Seam: An optimal message synchronizer in ros with well-bounded time disparity. In *Real-Time Systems Symposium (RTSS)*, pages 172–184. IEEE, 2023.

[52] S. Teerapittayanon, B. McDanel, and H.-T. Kung. Branchynet: Fast inference via early exiting from deep neural networks. In *International Conference on Pattern Recognition (ICPR)*, pages 2464–2469. IEEE, 2016.

[53] H. Teper, T. Betz, M. Günzel, D. Ebner, G. von der Brüggen, J. Betz, and J. Chen. End-to-end timing analysis and optimization of multi-executor ROS 2 systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2024.

[54] H. Teper, M. Günzel, N. Ueter, G. von der Brüggen, and J. Chen. End-to-end timing analysis in ROS2. In *Real-Time Systems Symposium (RTSS)*. IEEE, 2022.

[55] H. Teper, D. Kuhse, M. Günzel, G. v. d. Brüggen, F. Howar, and J.-J. Chen. Thread carefully: Preventing starvation in the ROS 2 multithreaded executor. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(11):3588–3599, 2024.

[56] K. Wilson, A. Arafat, J. Baugh, R. Yu, and Z. Guo. Physics-informed mixed-criticality scheduling for f1tenth cars with preemptable ROS 2 executors. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2025.

[57] C. Wu, R. Li, N. Zhan, and N. Guan. Improving the reaction latency analysis of message synchronization in ros. In *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 43–48. IEEE, 2024.

[58] C. Wu, R. Li, N. Zhan, and N. Guan. Modeling and analysis of the latesttime message synchronization policy in ros. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(11):3576–3587, 2024.

[59] S. Zilberstein. Using anytime algorithms in intelligent systems. *AI Mag.*, 17, 1996.

Preprint Version. Citation Info: XX.XXXX/XXXXXXX