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

RTAS Paper # 24, 11 pages

***Abstract*—Ensuring timely and predictable data propagation is essential in real-time robotic systems, yet a growing trend is to 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, utilizing its client-server architecture to provide a reliable task-cancellation mechanism and feedback channel. This solution requires no modification to the existing ROS 2 framework *(most notably, no change to the underlying ROS 2 codebase)* and remains compatible with existing ROS 2 end-to-end latency analyses. The effectiveness is demonstrated by evaluating system architectures and practical applications, showing that timely results can be achieved via flexible cancellations. Extensions to multi-threaded executors and accelerators and the impact on future research, like limited preemptive scheduling in ROS 2, are also discussed.

I. Introduction

Many modern robotic systems, such as autonomous driving platforms [41], must ensure timely and predictable data propagation to maintain safe operation. These systems incorporate computationally expensive tasks like sampling-based planning and vision-based sensing to ensure safe and efficient operation. 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 the integration of multiprocessor architectures with external accelerators like GPUs to provide the necessary computational power. This integration must be handled carefully to ensure that the system can leverage the computational power of these devices (non-preemptively) while maintaining 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 [21], [41]. A popular choice is the Robot Operating System 2 (ROS 2) [35], which provides the corresponding tools and libraries.

The ROS 2 executor is a user-space scheduler that handles the task management and implements 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 managed by the same executor. 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 *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, this solution introduces additional complexity and potential issues, as 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. [15] 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 with iterative algorithms for planning or perception, we can go one step further.

Specifically, we can transform the long-running task into an *anytime algorithm* [16], 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 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.

Specifically, we execute anytime algorithms using a dedicated component called *ROS 2 actions*, which utilize a client-server architecture for long-running tasks. ROS 2 actions are widely-used for handling goal-oriented tasks 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 process 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 [16].

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 changing 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 and rational, such that our solution requires only minimal effort and **no modification to the existing ROS 2 framework**. 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, and a generic solution for integrating most anytime algorithms, *requiring no change to the underlying ROS 2 codebase*. We plan to release our source code.
- **Tradeoffs.** Design tradeoffs, 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 a Monte Carlo Pi approximation and an Anytime YOLO 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.

II. Anytime Algorithms

This section provides an overview of anytime algorithms' fundamental benefits and implications for real-time systems. Optimization and iterative approximation algorithms, such as robot path planning via Monte Carlo Tree Search [12] and trajectory planning for autonomous racing cars [38], are often among the most computationally demanding tasks within autonomous driving software stacks [5].

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 system. Simple solutions are *time-budgeted* (or contract) algorithms [39], 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 [13], [53], 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 and use quality measures to track the result's quality during computation. Formulating these tasks as anytime algorithms allows 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, 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 controlling the algorithm's execution and 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 return the result 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 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 a time-interruptible code [16].

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.

III. ROS 2 Fundamentals

This section provides a high-level overview of the ROS 2 system model. Here, we focus on the information needed to elaborate on the additional challenges when considering ROS 2 as a platform to deploy anytime algorithms, 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 [8], [10], [46], [48]. We use the LTS release ROS 2 Humble [35] 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. The function of a task, denoted as *callback*, is executed when the task is scheduled.

The underlying *Data Distribution Service* (DDS) middleware facilitates communication between event-triggered tasks via three *publish-subscribe 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. Due to 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, the execution flow of the task must actively be managed over multiple subtasks. Thus, we introduce a task activation mechanism that handles the scheduling of segments, cancellations, the calculation of intermediate results, and returning the best result so far.

3. Anytime Interfaces. Key aspects of ROS 2 are its modularity and the ease of integrating new components. We discuss and choose an appropriate communication interface for anytime algorithms in ROS 2, providing a generic and reusable solution for various applications.

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 efficient 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.

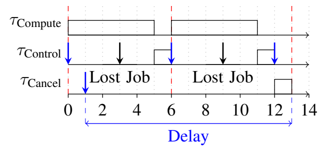
A. Enabling Anytime Computation

Our aim is to enable anytime algorithms in ROS 2 by providing an effective cancellation mechanism to interrupt long-running tasks and return intermediate results. In the following, 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.

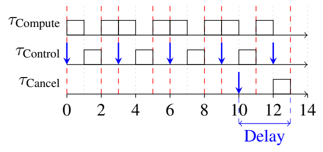
Executors determine the set of eligible task instances (jobs), the so-called *wait set*, and decide which job to execute next. The wait set is managed in two distinct, alternately recurring phases: *polling points* and *processing windows*.

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 event if multiple are present.

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.



(a) Scheduling delays and lost job behavior.



(b) Computation split into multiple segments.

Fig. 1: Scheduling delays and lost jobs in the single-threaded executor and the impact of splitting the computation.

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

Example 1 (Figure 1a). *Consider three tasks $\tau_{Compute} > \tau_{Control} > \tau_{Cancel}$, where τ_{Cancel} is processing 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 processed 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.[1] Thus, it is important to consider the execution times of tasks that are added to the wait set as well as their priorities, to minimize the scheduling delays for the single-threaded executor.

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. Furthermore, for the τ_{Cancel} task, which is activated at time 10 (indicated by the blue arrow), it is sampled and added to the wait set at the next polling point at time 11, and processed in the next processing window at time 12. After finishing at time 13, the total delay is 3 time units, a significant reduction over the previous case.*[2]

The example shows two key benefits of this approach: (1) reduced delays for other tasks, and (2) potentially less lost jobs. Furthermore, our approach introduces preemption points, which allow the integration of anytime algorithms. Specifically, other tasks can be scheduled between the computation in order to suspend, resume, or cancel the computation, as well as compute and return intermediate results. One limitation is the requirement that the algorithm can be split into multiple segments, which may not be feasible for all algorithms.

B. Anytime Task Management

The solution in Section V-A enables running any iterative algorithm as an anytime algorithm by splitting its computation into multiple jobs. For example, an anytime neural network can now be 'split' layer by layer. However, in comparison to the previous long-running task, which only had one job and was 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 the result computation with minimal overhead. This is especially important for algorithms that require a large amount of data or have many segments with short execution times.

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 specific conditions. Compared to timers, guard conditions do not have fixed periods, allowing for more dynamic and responsive task management.

Regarding Requirement 2, since guard conditions are intended to be used within a single ROS 2 node, they can directly access its shared data structures. Thus, the current state of the anytime algorithm can be stored in the node's shared data structures, allowing each segment to access and update the state directly without additional data transfer.

Implementation Variations: Managing anytime algorithms with guard conditions can be implemented in multiple ways. The solution we propose is shown in Figure 2. 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. Furthermore, it can also return status messages as *feedback*, and return intermediate results and *finish*. In addition to the management task, there are three main computation functions:

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

[2]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.

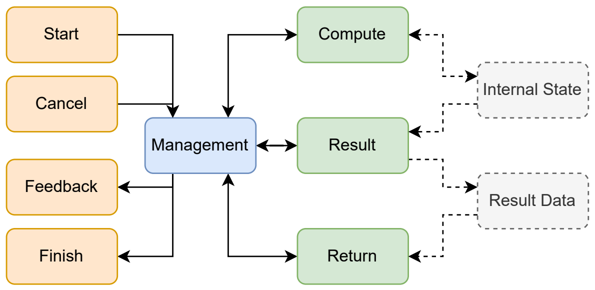


Fig. 2: Management Architecture of Anytime Algorithms

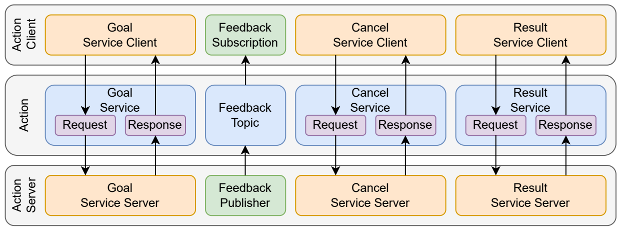


Fig. 3: Overview of ROS 2 Actions

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 specific algorithms, the concrete implementation of this architecture can vary. For example, the management task can be implemented as a guard condition that activates itself after each execution of the compute, result, and return functions. Alternatively, all functions (blue and green blocks in Figure 2) can be implemented as separate guard conditions that activate each other based on specific conditions. In that case, special care must be taken to ensure that the correct execution flow is maintained, and no race conditions occur.

In summary, guard conditions provide a flexible mechanism to manage the different tasks of anytime algorithms, allowing the concrete solution to be tailored to the specific requirements of the anytime algorithm and the system in which it is executed. 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*, and *finish* the anytime algorithm and *query* its state, allowing for fine-grained control. Moreover, this interface should provide a generic solution that only requires minimal changes to integrate different anytime algorithms into ROS 2.

ROS 2 provides three communication interfaces to choose from: *topics*, *services*, and *actions* [7]. 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 since one-way communication complicates fine-grained control. Specifically, their lack of a direct response mechanism makes it difficult to maintain a coordinated execution flow.

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 to process. 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.

Our goal is 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 affects existing scheduling analysis techniques for ROS 2, allowing us to provide timing guarantees for anytime algorithms integrated into ROS 2, as well as the benefits that our transformation provides for other tasks in the system.

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 subproblem (4) and subproblem (5), defined in Section IV, respectively.

A. Synchronization in Multi-Threaded Executors

For the single-threaded executor, all tasks are processed sequentially by the same thread. Hence, no synchronization between the tasks is needed. The ROS 2 multi-threaded executor enables parallel execution of tasks, allowing for an increase in system performance by running multiple tasks concurrently. However, this comes at the cost of increased complexity, as concurrent tasks may access shared data structures, requiring synchronization mechanisms to ensure data integrity.

In our case, this entails 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 of the *management*, such as *start*, *cancel*, *return*, and *finish*, need to be synchronized to ensure that the state of the action server 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 which tasks should be part of which callback group and how shared data structures can be synchronized. We keep this description general, as the exact implementation depends on the concrete task structure that is used in the algorithm.

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 take a look at two different approaches with a slightly different focus to group the *computation tasks* and how to synchronize the shared data structures.

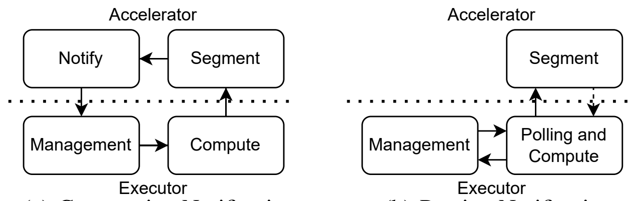
1) Each task is assigned to a separate callback group. Hence, the tasks can run in parallel, which is especially useful if the *segments* and *result computation* are computationally intensive. However, this approach requires a separate synchronization mechanism, external to the ones provided by ROS 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 to have it readily available whenever the algorithm finishes. Synchronization can easily 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



(a) Cooperative Notification (b) Passive Notification

Fig. 4: Cooperative and Passive Notification Mechanisms.

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 the accelerator completes its computations. This simplifies synchronization but may idle the CPU.

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 the result, while waiting for the accelerator to complete its computations. This approach requires synchronization between the executor and the accelerator. Specifically, there needs to be a mechanism to detect when a segment on the accelerator completes in order to avoid large 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 which the executor can respond to. Specifically, it is possible to use guard conditions by passing a reference to the guard notification and its trigger mechanism as an input to the accelerator, allowing it to directly notify once its computation is complete.
- *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, it is less responsive as the result computation is delayed until the computation is canceled.

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 computation 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. Contrarily, 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.

Scheduling Interference. In ROS 2 systems, it is common to have multiple executors, each executing multiple tasks. 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 tradeoffs between achieving responsiveness and maintaining efficient CPU utilization. Our design space mapping provides a foundation for analyzing these tradeoffs, 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 on 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.

We focus on a single long-running task τ_i, which is part of at least one cause-effect chain. Our goal is to transform τ_i into an anytime algorithm to improve 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.[3]
- $\tau_{\text{as-result}}$: 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 $\tau^{(k)}_{\text{as-comp}}$ for $k \in [1, N]$ has negligible execution time and overhead, as they primarily handle communication and coordination.

This transformation has a significant impact on the timing behavior of the entire system. 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. [46].

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. [46] 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.

[3]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.

In most cases, the interference for chains not involving τ_i will decrease. However, in the analysis by Teper et al. [46], 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. The original 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-cancelled 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, instead of scheduling one event for τ_i, the new chain requires multiple, separate scheduling events, each incurring its own scheduling latency and communication overhead. Second, each of these new, smaller tasks can be delayed by interference from other tasks in the executor. Consequently, the sum of these new overheads and interferences results 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 one specific chain, 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: $E_{\text{cancel}} = (\tau_{\text{trigger}}, \tau_{\text{ac-send-cancel}}, \tau_{\text{as-recv-cancel}}, \tau_{\text{ac-req-result}}, \tau_{\text{as-send-result}}, \tau_{\text{ac-recv-result}}, \ldots)$, with potentially additional tasks following. The total cancellation latency is 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. This chain still includes many tasks compared to the original monolithic task. However, depending on the system's configuration, specifically when assigning the anytime algorithm to a dedicated executor, this cancellation latency can be low enough to meet the system's responsiveness requirements.

IX. Implementations and Evaluations

We evaluated two anytime algorithms integrated into ROS 2 using the proposed framework: (1) CPU-bound Monte Carlo π approximation in Section IX-A and IX-B, and (2) GPU-bound AnytimeYOLO object detection in Section IX-C. Our experiments aim to answer the following 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 *proactive* and *reactive* architectures (Section VII) as well as single-threaded and multi-threaded ROS 2 executors (Section VI-A) to show their impact on system performance. Regarding accelerator integration (Section VI-B), we evaluated both synchronous and cooperative asynchronous execution of the GPU-bound AnytimeYOLO algorithm.

We implemented our architectures using modular template-based C++ classes in ROS 2, where the architecture can be selected at compile time. Our implementation includes the task-activation mechanism through guard conditions and the configuration of callback groups for the multi-threaded executor. It is based solely on the interfaces and task types provided directly by ROS 2, and a generic solution to integrate most anytime algorithms, *requiring no change to the underlying ROS 2 codebase.* Furthermore, we added custom tracepoints using ROS 2's tracing framework [4] to measure the timing behavior of our architectures with minimal overhead. We plan to release our source code.

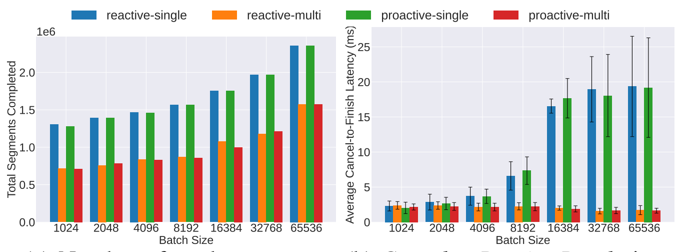
We implemented our experiments using ROS 2 Humble Hawksbill with CycloneDDS as the DDS implementation. The experiments are conducted on an Nvidia Orin NX with 16 GB of shared memory, using Ubuntu 22.04 (Jetpack 6.2).

A. Monte Carlo Pi Approximation

Monte Carlo Pi Approximation is a classic anytime algorithm. It repeatedly generates random points in a unit square and counts how many fall within the unit circle inscribed within the square. The ratio of points inside the circle to the total number of points, multiplied by 4, approximates π. The algorithm can be interrupted at any time, returning the current approximation based on the points sampled so far.

We evaluated the following parameters:

- **Block Size**: 2^{10} (1024) to 2^{16} (65536) segments per job.



(a) Number of total segments. (b) *Cancel* to *Receive Result* time.

Fig. 5: Monte Carlo π approximation results for different executor and architecture configurations.

- **Architecture**: Either proactive or reactive.
- **Executor-Threading**: Either single-threaded or multi-threaded (two threads)[4].

Each configuration ran for one minute. The action client sent a goal request to the action server every 500 ms. After 200 ms, the action client sent a cancellation request to the action server. We evaluated the following two metrics:

- **Segment Count**: The number of segments computed before the cancellation request is received. Higher segment counts indicate that more work is done, suggesting greater efficiency and less overhead.
- **Cancellation Delay**: The time between the cancellation request of the action client and receiving the final result. The lower the value is, the higher the responsiveness.

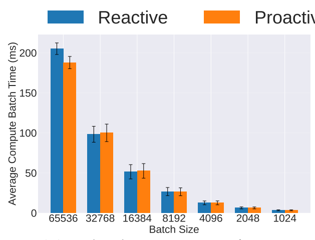
The results are shown in Figure 5. For the *segment count* in Figure 5a, we see a linear increase in the number of segments with increasing block size, showing that the scheduling overhead becomes less significant for larger blocks. Regarding the *cancellation delay* in Figure 5b, larger blocks lead to higher cancellation delays for the single-threaded executor, as the algorithm takes longer to finish the current block before it can return the result. Meanwhile, the multi-threaded executor is always able to maintain a low cancellation delay due to its ability to return the result in parallel with a running computation. However, the added overhead of managing multiple threads leads to a lower segment count compared to the single-threaded executor. This shows the trade-off for task granularity discussed in Section VII, as well as the impact of executor configuration on the performance.

B. Interference on Other Tasks

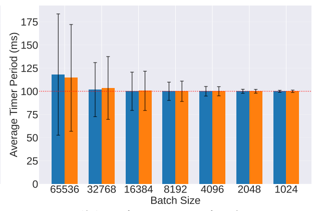
We evaluated the interference of the Monte Carlo Pi Approximation on other tasks in ROS 2, by varying its block size. We added a timer with period 100 ms and compared the time between two consecutive executions with the expected 100 ms. We evaluated proactive and reactive architectures, but only for the single-threaded executor to cause interference between the anytime algorithm and the timer.

The results are shown in Figure 6 and Table I. With increasing block size, the computation time of one block increases linearly, as shown in Figure 6a. The timer periods

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



(a) Block compute time.



(b) Timer period.

Fig. 6: Impact of Monte Carlo Pi Approximation block size on (a) computation time and (b) periodic task's timer latency.

TABLE I: Percentage of missed timer periods for different block sizes and architectures.

Configuration	1024	2048	4096	16384	32768	65536
Proactive	0.0%	0.0%	0.0%	4.60%	9.41%	23.38%
Reactive	0.0%	0.0%	0.0%	3.41%	6.98%	16.44%

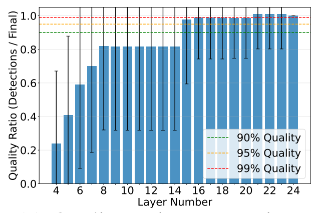
are increasingly affected by the longer computation times of the anytime algorithm, as shown in Figure 6b, leading to latencies of up to 190 ms between two consecutive timer callbacks for the largest block size. Furthermore, for the block sizes of 16384, 32768, and 65536, the periodic task misses timer periods due to the interference caused by the anytime algorithm, as shown in Table I, with the proactive architecture causing more interference than the reactive one.

In total, our experiments based on the Monte Carlo Pi Approximation algorithm demonstrate that algorithms using the ROS 2 executor need to balance task granularity and system responsiveness to have a well-functioning system.

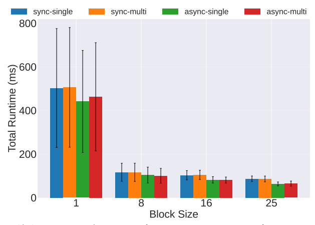
C. Anytime YOLO

AnytimeYOLO [23] is an anytime variant of the popular YOLO (You Only Look Once) object detection algorithm [37]. For our evaluation, we consider the medium variant with a body of 22 computational blocks, which, combined with the exit, post-processing module, and a final completion step, results in 25 distinct steps. The goal of our experiment is to find an AnytimeYOLO configuration that achieves similar detection performance to the full model while only running a fraction of the network, and to demonstrate its capability to dynamically adapt to intermediate results. We adopted the following workflow:

- We used the MS COCO validation dataset [30], specifically the 191 images that contain *traffic lights*.
- We determined the execution time of each layer and exit to assess its computational cost.
- The quality ratio progression per layer for *traffic lights* (compared to the full network's result) identifies how many layers are needed to reach a target quality. Figure 7a shows that the quality ratio is stagnant from layers 8-14 and reaches near-full quality after 16 layers.
- We determined suitable block sizes (preemption points) that balance responsiveness and efficiency, evaluating block sizes of 1, 8, 16, and 25 layers in our experiments.



(a) Quality ratio progression.



(b) Total runtime comparison.

Fig. 7: AnytimeYOLO evaluation showing (a) quality ratio progression for traffic light detection per layer and (b) total runtime comparison for different block sizes.

- We ran AnytimeYOLO with these configurations to evaluate the total runtime and assess performance.

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

To realize the dynamic cancellation, we triggered it based on concrete detection results. Specifically, we canceled the action if *traffic lights* are detected with an accuracy of 70% or more. We achieved this by adding a *feedback* message to the action server, which contained the detected object IDs and an accuracy estimate. In our proactive setup, the server sent this feedback to the client after each block is processed. The client then checked the feedback against the cancellation condition and issued a cancellation request if the 70% accuracy threshold is met, after which the server returned the current result.

We evaluated synchronous (*sync*) and asynchronous (*async*) execution. As the computation runs on the GPU, the executor must be notified when the GPU is ready for the next layer (see Section VI-B). For the non-blocking *async* case, we implemented a *cooperative* waiting mechanism: we enqueue a host callback function into the CUDA stream, which triggers the action server's guard condition once the GPU finishes its task.[5] We did not evaluate the *passive* polling mechanism, as it performs worse than this cooperative approach.

Results. As shown in Figure 7b, configurations running only 8 or 16 layers per block achieve total runtimes similar to running the full YOLO model. While this specific experiment does not reduce the total computation time (as the goal was quality-based cancellation), it successfully demonstrates the system's ability to dynamically adapt the computation based on the quality of intermediate results.

X. Discussions and Future Research

In this section, we highlight research directions opened up by our contribution and discuss how they relate to existing challenges and potential issues in ROS 2 development.

Limited Preemption in ROS 2. Most prior work on ROS 2 scheduling [8], [10], [11], [47], [49] is constrained by the non-preemptive nature of the executor, where individual callbacks

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

run to completion once started. While preemption in real-time systems broadly refers to interruptibility and the ability to cancel or pause a task, ROS 2 does not natively support preemption at the callback level.

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 (WCET), future work can focus on cancellation delays, enabling more adaptive and robust scheduling guarantees. These capabilities are particularly relevant for dynamic-priority scheduling and overload control scenarios.

Execution and Communication Synchronization. Another important topic 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 [40]. Our approach can be extended to synchronize the execution of concurrent workloads by parallelizing multiple computations that are run side-by-side using the proposed anytime task management. Given our dynamic task-activation mechanisms, synchronizing the activation of tasks across different nodes is possible. This is especially relevant in the context of timing misalignment [22], which can affect system performance and predictability.

Another important topic in ROS and ROS 2 is message synchronization [26], [27], [29], [44], [51], [52]. 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 actively 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 the accelerator must be considered. To this end, previous work on accelerator task scheduling management in ROS 2 [14], [28] 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 alternative to compute-intensive tasks and enable adaptive use of computational resources. In contrast, time-budgeted approaches like contract algorithms [39] provide latency guarantees for multi-stage processing pipelines by assuming a predetermined budget for each task [17], [38]. Recently, there has been growing interest in anytime machine-learning algorithms for real-time systems [3], [6], [18]–[20], [25], [31], [32], [42], [43]. Specifically, such algorithms are applied for perception tasks such as object detection and classification. The anytime property can be achieved by early exits for inference in neural networks, e.g., [9], [24], [25], [42], [43], [45]. Other possibilities involve inspecting a smaller input and altering the processing order of the data, e.g., [3], [19], [20], [31], [32], [42], [43]. Additionally, widely-used optimization [17] and iterative approximation algorithms, like route planning [12], [34], [38], also demand significant computational resources.

ROS 2. The Robot Operating System [36] was released in 2007 and has become a widely-used framework for creating robot systems. It was originally not designed to be real-time capable. Its successor ROS 2 [33], introduced in 2018, has been designed to be a more flexible and modular framework. Several real-time studies consider analyses for ROS 2. Blass et al. [8] established response-time bounds for ROS 2 tasks that propagate data through the system. Teper et al. [46], [47] conducted a formal analysis of end-to-end latencies in ROS 2, which is essential for determining the time required for data propagation in a ROS 2 system. Furthermore, there have been extensions to ROS 2 to support preemptive or dynamic scheduling without relying on the native ROS 2 executor [1], [2], [50] Moreover, in a recent result, Teper et al. [48] 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 to ensure timely task completion. We identified and solved key challenges caused by the non-preemptive nature of the ROS 2 executor, which typically causes high interference and prevents early task termination. Our solution is to split the long-running computation into a series of smaller, manageable segments. This transformation enables timely task completion, reduces interference with other tasks, and naturally facilitates the implementation of *anytime algorithms* by creating preemption points where tasks can be canceled.

We leverage 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 is demonstrated by evaluating system architectures and practical applications, showing that timely results can be achieved via flexible cancellations, which requires careful consideration of the design choices for optimal performance. Crucially, our solution requires **no modification to the existing ROS 2 framework** *(most notably, no change to the underlying 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.

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 *59th ACM/IEEE Design Automation Conference*, DAC '22, 2022.

[3] S. Bateni and C. Liu. Apnet: Approximation-aware real-time neural network. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, 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] T. Betz, M. Schmeller, A. Korb, and J. Betz. Latency measurement for autonomous driving software using data flow extraction. In *2023 IEEE Intelligent Vehicles Symposium (IV)*, pages 1–8, 2023.

[6] 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.

[7] G. Biggs, J. Perron, and S. Loretz. Ros 2 actions, Mar. 2019. https://design.ros2.org/articles/actions.html.

[8] T. Blass, D. Casini, S. Bozhko, and B. B. Brandenburg. A ros 2 response-time analysis exploiting starvation freedom and execution-time variance. In *Proceedings of the 42nd Real-Time Systems Symposium (RTSS)*, 2021.

[9] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han. Once-for-all: Train one network and specialize it for efficient deployment. *arXiv preprint arXiv:1908.09791*, 2019.

[10] D. Casini, T. Blass, I. Lütkebohle, and B. B. Brandenburg. Response-time analysis of ros 2 processing chains under reservation-based scheduling. In *Proceedings of the 31st Euromicro Conference on Real-Time Systems (ECRTS)*, 2019.

[11] H. Choi, Y. Xiang, and H. Kim. Picas: New design of priority-driven chain-aware scheduling for ROS2. In *27th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2021, Nashville, TN, USA, May 18-21, 2021*, pages 251–263. IEEE, 2021.

[12] T. Dam, G. Chalvatzaki, J. Peters, and J. Pajarinen. Monte-carlo robot path planning, 2022.

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

[14] D. Enright, Y. Xiang, H. Choi, and H. Kim. Paam: A framework for coordinated and priority-driven accelerator management in ROS 2. In *2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 81–94. IEEE, 2024.

[15] 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.

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

[17] N. J. A. Harvey, C. Liaw, E. Perkins, and S. Randhawa. Optimal anytime regret with two experts, 2021.

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

[19] Y. Hu, I. Gokarn, S. Liu, A. Misra, and T. Abdelzaher. Algorithms for canvas-based attention scheduling with resizing. In *IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2024.

[20] 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 *28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2022.

[21] 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 *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, April 2018.

[22] 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 *30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2024.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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)*, 2022.

[28] 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 *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 93–105, 2023.

[29] 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 *2023 IEEE Real-Time Systems Symposium (RTSS)*, pages 185–197, 2023.

[30] 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 *Computer vision–ECCV 2014: 13th European conference, zurich, Switzerland, September 6-12, 2014, proceedings, part v 13*, pages 740–755. Springer, 2014.

[31] 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 *28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2022.

[32] 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)*, 2020.

[33] 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.

[34] R. Mangharam and A. A. Saba. Anytime algorithms for gpu architectures. In *32nd Real-Time Systems Symposium (RTSS)*, 2011.

[35] Open Robotics. Ros 2: Humble, 2024. https://docs.ros.org/en/humble.

[36] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng. Ros: an open-source robot operating system. volume 3, 01 2009.

[37] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection, 2016.

[38] 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.

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

[40] 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.

[41] SOAFEE. SOAFEE: Scalable open architecture for embedded edge, 2022.

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

[43] 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.

[44] 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 *2023 IEEE Real-Time Systems Symposium (RTSS)*, pages 172–184, 2023.

[45] S. Teerapittayanon, B. McDanel, and H.-T. Kung. Branchynet: Fast inference via early exiting from deep neural networks. In *2016 23rd international conference on pattern recognition (ICPR)*, pages 2464–2469. IEEE, 2016.

[46] 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 *30th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2024.

[47] H. Teper, M. Günzel, N. Ueter, G. von der Brüggen, and J. Chen. End-to-end timing analysis in ros2. In *Proceedings of the 43rd Real-Time Systems Symposium (RTSS)*, 2022.

[48] 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.

[49] 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.

[50] 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 *2025 IEEE 31st Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2025.

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

[52] 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.

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