ConfigBot: Adaptive Resource Allocation for Robot Applications in Dynamic Environments

Rohit Dwivedula, Sadanand Modak, Aditya Akella, Joydeep Biswas,
Daehyeok Kim, and Christopher J. Rossbach

Abstract— **The growing use of service robots in dynamic environments requires flexible management of on-board compute resources to optimize the performance of diverse tasks such as navigation, localization, and perception. Current robot deployments often rely on static OS configurations and system over-provisioning. However, they are suboptimal because they ignore variations in resource usage, leading to system-wide issues like robot instability or inefficient resource utilization. This paper presents ConfigBot, a novel system designed to adaptively reconfigure robot applications to meet a predefined performance specification by leveraging *runtime profiling* and *automated configuration tuning*. Through experiments on multiple real robots, each running a different stack with diverse performance requirements, which could be *context*-dependent, we illustrate ConfigBot's efficacy in maintaining system stability and optimizing resource allocation. Our findings highlight the promise of automatic system configuration tuning for robot deployments, including adaptation to dynamic changes. Code available at: https://github.com/ldos-project/configbot**

I. Introduction

Service robots are increasingly being used to assist humans in everyday tasks, with commercial examples including Amazon Astro, EEVE Willow, and Misty the Robot. Currently, these robots rely on vendor-programmed functions, but we believe that future extensibility through third-party applications will unlock diverse capabilities - much like app stores (*e.g.* Google Play Store) did for smartphones. Furthermore, the set of actively running apps can change dynamically—whether through user-driven actions like starting or stopping functionality, installation/removal of apps, or robot personalization per user [1], [2], thereby significantly expanding the range of workloads supported on a given robot.

Real-time responsiveness is crucial for robot apps, as delays or outdated data can hamper effective functionality in dynamic environments (§IV-A). Ensuring that numerous concurrently running and dynamically changing sets of apps can effectively share the computational resources on robots while meeting their real-time goals is thus an important resource allocation problem. Most consumer-grade robots are inherently resource-constrained, meaning they cannot simply scale resources on demand—unlike cloud servers—to meet apps' performance needs. This makes the resource allocation problem in robots particularly challenging.

Robot apps' distinct attributes further add to the challenge (§III-B). First, robot apps have highly diverse resource requirements, performance targets, and service level objectives (SLOs). Second, robot apps operate in various environments—such as homes, workplaces, and outdoor settings—where resource demands vary significantly due to environmental variations. Third, robots frequently adapt their algorithms based on the environment [3], adding to resource demand volatility.

Modern operating systems (OSes)–particularly the Linux variants widely used in robotics–fail to account for these *context-specific* app resource needs due to reliance on simple resource allocation heuristics. A potential workaround is to over-provision resources and manually set static allocations for each app such that it performs adequately across *all* contexts; however, this is impractical on resource-constrained systems.

To address these challenges, we introduce ConfigBot, a novel resource allocation framework for service robots that *automatically tailors* app resource allocations to dynamic contexts by building on three unique *opportunities* that derive from properties of robot apps, as described next.

First, robot apps involve persistent, long-running tasks like object tracking and sensor processing [4]. Thus, ConfigBot optimizes resource allocation infrequently targeting stable environments with consistent workloads, and relies on lightweight runtime monitoring to detect significant environmental or workload changes. Further, ConfigBot maintains a library of proven configurations linked to specific environments and quickly reapplies them in familiar contexts, ensuring efficiency with minimal overhead.

Second, robot functionality can be divided into *core services* (*e.g.* localization, navigation) and *non-core apps*. Building on this, ConfigBot formulates resource allocation as an optimization problem, treating core service performance as *constraints* and non-core app performance as *objectives*. This reduces decision variables by limiting essential service tuning, and simplifying optimization.

Third, many robot apps operate in a data-driven manner, performing computations only when new data arrives. Coarse resource limits (*e.g.* setting max CPU usage limits) can trigger adverse effects (§IV-C) that degrade the performance of the throttled process needlessly. ConfigBot controls data flows – particularly for non-core apps – via the novel *adaptor* abstraction (§V), gracefully reducing computational demand under resource constraints without disrupting critical functions.

ConfigBot uses Bayesian Optimization to tune the values of OS-based system-level (`cgroup`) settings, as well as the thresholds for the proposed ROS-layer adaptors to meet a developer-defined *specification* of desired robot behavior.

Department of Computer Science, The University of Texas at Austin. This work has been supported by U.S. National Science Foundation (NSF) grants CISE-2326576 and NSF CAREER-2046955, with additional support from gifts by Meta and Cisco. Any opinions, findings, and conclusions expressed in this material do not necessarily reflect the views of the sponsors.

TABLE I: Common robot apps: typical resource needs, execution frequency and performance metrics

Application	Frequency	Resources Used	Performance Metrics	# of topics	
				published to	subscribed to
LiDAR processing	40 - 60 Hz	I/O, CPU	Latency, Density	12	2
Navigation	20 - 40 Hz	CPU, GPU	No collisions, smooth movements	25	18
Localization	30 - 40 Hz	CPU	Accuracy, Recovery	4	5
Web dashboard	10 Hz	Network	Latency/delay, throughput	3	12
Object detection	1-5 Hz	CPU, GPU	Frame rate, detection accuracy	2	2

Once deployed, ConfigBot *learns* a configuration for the current workload + context that meets the specification and applies this identified configuration to the system – without requiring code changes to the robot apps/services. We also discuss how ConfigBot monitors for context shifts that could invalidate prior configurations and re-optimizes for new workloads and environments automatically.

We evaluate ConfigBot on state-of-the-art robot platforms, and illustrate how configurations our system identified allowed a navigation and obstacle-avoidance system on the Boston Dynamics `Spot` to run on 4× fewer CPUs while still meeting developer-specified performance requirements (§VIII-A). We then show that our solution generalizes to multiple sets of robot apps (§VIII-B), varying environments (§VIII-D) and other robots (§VIII-E).

II. Related Work

Automated config search. Automated config search has been successful at making web servers [5], [6], databases [7], and cloud apps [8] more efficient. They use bayesian optimization [5], bandit algorithms [8], causal inference [6] or combinations of these techniques [9] to *tune* config knobs. Cherrypick [5], for instance, uses bayesian optimization to efficiently identify the optimal cloud configuration (*i.e.*, VM type, number of CPUs, etc) for big data analytics jobs. To the best of our knowledge, robot OS tuning via automated config search has not been attempted so far.

Context-dependent navigation. Most work in this area primarily focuses on getting a set of "good" parameters under different environments. APPLD [3] uses labeled contexts (*i.e.*, environments) to train a supervised classifier that selects a context-conditioned behavior cloning policy. Another work [10] tries to dynamically find trajectory optimization weights for a Dynamic Window Approach (DWA) planner for straight-line and U-turn contexts. However, none of the aforementioned works detect workload changes automatically as they do not profile the system, and so they need a way to classify the contexts a priori.

III. Background

In this section, we describe modern robot hardware (§III-A) and key attributes of robot apps (§III-B), setting the stage for the resource management challenges discussed subsequently.

A. Robot Hardware & Compute

Service robots vary in form (e.g., wheeled, quadruped, humanoid) and feature diverse hardware configurations suited to their environments. Most rely on resource-constrained embedded systems, making efficient resource allocation critical for performance across workloads. Typically, robots are equipped with (1) LiDAR or depth cameras for 3D perception and (2) an inertial measurement unit for odometry and orientation, both essential for navigation and obstacle avoidance. Some also have monocular or stereo RGB cameras. Additional hardware may be used depending on the application, such as tactile sensors for terrain-aware navigation [11] or network cards for server communication and remote processing [4], [12].

Our framework's design and evaluation target multiple robot platforms, further detailed in §VII.

B. Characteristics of Robot Applications

Robot apps are typically built using ROS, in which functionality is implemented in multiple *nodes*, each of which is a separate Linux process. Nodes communicate via *topics*, a message-passing system that enables publishing and subscribing to named queues. Robots run a "*stack*"—a cohesive set of ROS apps—continuously to perform their tasks.

Fig. 1 illustrates the structure of a **Basic-NAV** stack, which we use as a running example. This stack consists of five apps, communicating through publisher-consumer dependencies. While simplified in the figure, each node is highly complex. For example, the "navigation algorithm" node—one part of the navigation app—publishes to 21 topics, subscribes to 16, and employs a worker thread pool for concurrent computation.

Beyond this complexity, robot apps have unique characteristics that complicate resource allocation:

- **Diverse resource demands and execution needs:** As shown in Table I, robot apps vary widely in resource usage, execution rates, and performance requirements. Some rely mainly on CPUs (*e.g.*, localization), others on GPUs (*e.g.*, object detection), and some on both (*e.g.*, navigation). Additional resources like networking (*e.g.*, web dashboards) or I/O (*e.g.*, LiDAR) may also be critical. Execution rates differ significantly: navigation runs at 20–40Hz, while object detection operates 4–40× slower. *Unlike traditional workloads* (e.g., *databases, cloud computing), robot applications exhibit extreme resource heterogeneity, often coexisting on the same*

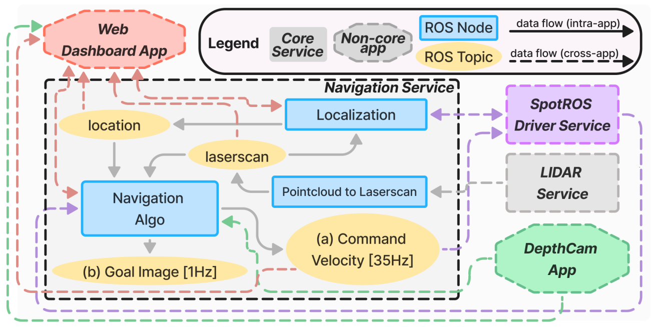


Fig. 1: **Basic-NAV** stack. A *stack* (*i.e.*, set of robot apps) w/ sensor processing, navigation, and telemetry [12]. Navigation is a *core* service and produces two outputs: (a) **command velocity** for actuators and (b) a **goal image** showing the projected path for monitoring/debugging. The minimum safe update rate required for (a) is 35Hz (developer-specified). All communication (arrows) is via ROS topics; topic names omitted for cross-app flows (dashed arrows) for brevity.

TABLE II: Resource usage of NAV algorithms (§VII).

Algorithm	Compute	Memory
Basic-NAV (Fig 1)	1.1 CPUs	526 MB
Intermed-NAV	1.5 CPUs	490 MB
Terrain-NAV	6.9 CPUs, GPU	1267 MB

machine. Cloud schedulers typically consolidate homogeneous workloads [13], [14] onto the same machine.

- **Implicit environment dependencies:** A robot's physical environment greatly influences resource consumption. Tasks like pose estimation [15], [16] and object tracking [17] involve multi-stage pipelines—first detecting objects, then estimating poses—leading to higher compute costs in dynamic or crowded settings. Similarly, telemetry systems [18] compress static scenes efficiently but generate larger, compute-intensive streams in dynamic environments [19], [20]. Thus, *static resource allocations are ineffective; efficient management requires continuous adaptation to environmental changes.*
- **Environment-adaptive algorithm selection:** Robot developers often choose algorithms based on environmental context, further amplifying resource variability. For instance, social navigation [21] may be necessary in crowded indoor spaces, while terrain-aware algorithms [11] are better suited for outdoor environments. Algorithm selection can also occur dynamically at runtime [22]. Table II highlights how resource usage for the "same" navigation task varies based on algorithm. *Resource allocation must not only be workload-specific but also dynamically adapt to environmental shifts.*

IV. Resource allocation in modern robots

Allocating a robot's compute resources to applications is handled by the underlying *operating system* (*i.e.*, Linux in ROS-based robots), which has multiple *subsystems* (CPU, I/O, network, etc.) that coordinate resource-sharing via heuristics (*e.g.*, the "completely fair scheduler" [23] for CPU scheduling). Unfortunately, these default resource schedulers often lead to degraded robot performance (§IV-A) since they prioritize broad objectives (*e.g.*, fairness), which conflict with the complex, interdependent and environment-dependent nature of robot stacks and their execution (§III). Linux allows configurability of these schedulers through `cgroups`, which enable fine-grained CPU allocation through resource limits and priorities at a per-process level. While `cgroups` can improve performance by tailoring resource allocation (§IV-B), we find that they alone cannot fully address all performance issues in robot apps (§IV-C). We present and argue for the abstraction of configurable *adaptors* (§V), which modulate data flows between ROS nodes serving as important knobs for performance and stability.

TABLE III: Performance issues in resource-constrained robots

Robot	Observed degradation in functionality
① `Spot`	Stumbling, inability to stabilize at destination, intermittent pauses during navigation.
② `Jackal`	Collisions with obstacles
③ `Cobot`	Delays, jerkiness in manipulating objects.

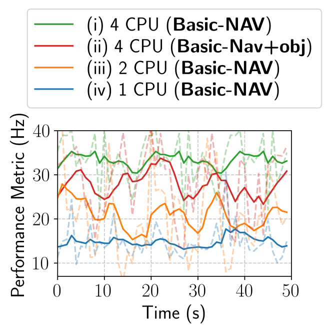
A. *Default OS configurations frequently lead to performance degradation in resource-constrained robots*

We study the performance offered by the default OS configuration when running (i) **Basic-NAV** (Fig 1) and (ii) **Basic-NAV** along with an object detection (obj) application [24] on the Boston Dynamics `Spot` robot.

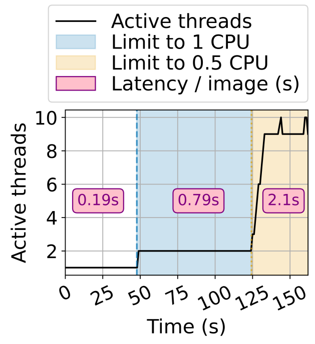
Figure 2a shows *how frequently* new navigation decisions are sent to the actuators in settings with varying resource constraints. In a resource-rich system, without any memory or compute limits, **Basic-NAV** was designed to update these *command velocities* at 35-40 Hz, a condition that is even close to being satisfied only when > 4 CPUs are available, as shown in Fig 2a-i. When we constrain the system to fewer CPUs (Figs 2a-iii, 2a-iv) or introduce an additional app (Fig 2a-ii), we observe that the default OS configurations cause the frequency of updates to fall by up to a factor of $3\times$. In particular, the reduced frequency of execution of the core app leads to degraded robot performance, resulting in end-to-end failures summarized in Table III. Here, the default OS configurations do a poor job of allocating *resources that are available* across running jobs, inducing destructive inter-app contention which we show below is avoidable.

B. *`cgroup` settings can help core app performance*

Control groups (`cgroups`) are a core feature of Linux enabling fine-grained resource management. They allow administrators to define CPU, memory, and I/O usage limits and priorities for processes, which the kernel's scheduling mechanisms enforce at runtime. The `cpu.max` controller, in particular, allows setting the maximum fraction of CPU



(a) Update rate of command velocity in resource-constrained deployments of `Spot`. None of these deployments (ranging from 1–4 CPUs) satisfy the requirement of > 35Hz consistently; as a result, `Spot` experiences performance degradation (Table III).



(b) **Basic-Nav+obj** stack on `Spot`. Number of active threads in the object detection [24] app under increasingly tight CPU allocation. At 0.5CPUs, latency exceeds 2s as threads compete for a drastically reduced CPU budget.

Fig. 2: Impact of resource constraints

time that can be allotted to a process over a time window. `cgroups` can form a useful mechanism to mitigate application contention in the previous section.

For instance, the performance degradation seen in Fig 2a-(ii) could trivially be solved by starving the object detection app. In practice, robot developers today manually identify an appropriate `cpu.max` limit for the new app, ensuring it does not overuse resources, through a combination of trial and error and intuition. As workloads and environments change, these static allocations require frequent reconfiguration, which is challenging once the robot has been deployed to run in the real-world. We will see in §VIII-B that ConfigBot can automatically set `cgroup` settings to improve performance only up to a certain extent.

C. `cgroups` on their own are not sufficient

Unfortunately, as we show next, relying solely on CPU throttling via `cgroups` leads to suboptimal system behavior.

Internal contention within apps. To illustrate the limitations of `cgroups`, consider the **Basic-Nav** stack (Fig 1). Some parts of the navigation app, which process LiDAR data and compute the navigation command to send down to actuators (**Task 1**), are clearly *core* parts of the stack. However, the same app also overlays the planned path on an image from the camera ("Goal Image" in Fig. 1) for monitoring, logging and debugging (**Task 2**) - useful but non-core functionality. In resource-constrained environments, **Task 2** can degrade **Task 1** by competing for the same compute resources.

Shared worker pools. In the above example, we would ideally deprioritize threads handling **Task 2** to ensure **Task 1** remains unaffected. However, this is challenging because both tasks share a single thread pool: the same thread may serve either task, depending on when data arrives. This scheduling behavior is typical in C++ ROS applications using `ros::AsyncSpinner` (or `rclcpp::executors::MultiThreadedExecutor` in ROS2), which dispatches callbacks across a set of threads. A similar issue arises in Python nodes relying on `rospy.spin()`, commonly used to implement multithreaded callbacks. Both APIs serve as the *de facto* approach for implementing the perpetual event loop in ROS, making them ubiquitous in robot apps.

Oversubscription under throttling. A `cgroup`-based controller for the **Basic-Nav+obj** stack might reduce CPU shares for the object detection (a non-core) app whenever a core app is at risk. Object detection is a multithreaded app that uses a thread-pool of workers to process images; restricting the CPU usage of a multithreaded app can trigger a surge in parallel tasks, causing severe performance drops. As shown in Figure 2b, lowering the object detection app's CPUs from 4 to 1 and then 0.5 drastically increases the number of active threads (capped at 10), raising image-processing latency beyond two seconds per frame. This occurs because each incoming image spawns a new thread. Under tighter CPU constraints, multiple threads remain active simultaneously, contending for an ever-diminishing resource pool and compounding the performance penalty.

In both the examples in this section, direct control over data flows could be used to diminish these performance penalties. In the shared worker pool case, for instance, we could throttle the number of images being sent to **Task 2** – the lesser images received by **Task 2**, the lesser its' compute footprint. Similarly, in the oversubscription example, throttling the number of incoming images could help prevent the compounding penalties. *Adaptors*, described in the next section, are an abstraction we built to modulate data flows such as these in a *transparent* way (*i.e.* not requiring application code changes).

V. Throttling data flows to control resource usage

Adaptors (Fig. 3) offer a finer-grained resource control mechanism. In the above example scenario, adaptors can "intercept" camera data on a per-subscriber basis, throttling (*i.e.* dropping data) only the problematic paths without penalizing everyone else. Adaptors thus enable ROS to manage data flows within and across apps and achieve more nuanced resource control than `cgroups` alone can provide. Adaptors are inserted on each subscription edge and they control the frequency and volume of messages passed to each consumer. Both attributes can be dynamically reconfigured at runtime to shape the resource usage pattern of the consumer. We integrate adaptors into the `ros_comm` library, a core ROS component responsible for communication, by modifying the subscriber API to enable message filtering. Adaptors are automatically setup when an app running on our modified ROS build calls `rospy.Subscriber` (Python) or `nh.subscribe` (C++). Each adaptor also has a dedicated ROS service associated with it, allowing dynamic adjustment of its filtering frequency at runtime. Our implementation required only ~ 150 lines of code across the C++ and Python implementations of `ros_comm`.

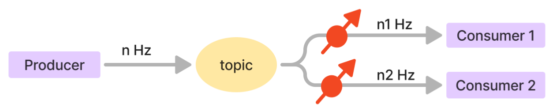


Fig. 3: In traditional ROS message passing, every consumer receives all messages published to the topic (*i.e.* $n = n_1 = n_2$). Our custom ROS build places *adaptors* (🧲) on each subscription edge, enabling independent rate control per-consumer.

The problems we described in §IV-C can be fixed by changing the application code itself; for example, a developer could create separate worker pools for each independent task in the node, or split out the node into multiple nodes. However, these approaches are intrusive, forcing developers to re-architect their apps. Adaptors, by contrast, offer an application-transparent, drop-in solution. We will show in §VIII-A and §VIII-B that the addition of adaptors allows ConfigBot to unlock higher performance over a ConfigBot-like approach that uses only `cgroups`.

VI. ConfigBot Design

The previous section demonstrated how resource constraints degrade robot stacks (§IV-A), and how carefully tuning CPU limits (§IV-B) or inserting adaptors (§V) can restore performance. However, deciding precisely which `cgroup` parameters or adaptor rates to apply remains challenging, often requiring expert insight into application dependencies and resource demands and painstaking and error-prone manual tuning. Even after identifying the best configuration, we will show (§VIII-C) that almost any change in the workload renders the existing configuration suboptimal for the new scenario. ConfigBot, which can automatically pinpoint "good" configurations for complex app mixtures running on a robot at any given time and environment is proposed as a solution to this challenge.

A. Problem Definition

Consider a robot stack (*i.e.*, the set of apps a_1, a_2, ..., a_n) operating under specified conditions, which include both system resources and the physical environment. For each app, we assume the developer provides a performance metric ($\mathbf{J}_i$) to evaluate its behavior (*e.g.*, frame rate, message publish frequency) and a *desirable target* value J_i^o. Achieving this target ensures the robot operates effectively. This assumption is reasonable, as such metrics are often easy to define.

As mentioned earlier, we categorize robot apps into two classes:

- **Core services (A_c):** Critical for the robot's functionality (*e.g.*, localization, navigation, obstacle avoidance) and must always perform reliably.
- **Non-core apps (A_n):** Non-essential and can be executed opportunistically when resources permit.

Armed with this categorization, we can specify the task of identifying an optimal configuration as a constrained black-box multi-objective optimization problem:

$$\begin{aligned} &\underset{c}{\operatorname{argmax}} \min(\mathbf{J}_i(c), J_i^o) \quad \forall a_i \in A_n \\ &\qquad \text{s.t.} \;\; \mathbf{J}_i(c) \geq J_i^o \;\; \forall a_i \in A_c \\ &\qquad\qquad c \in \mathbf{C} \;\; \textit{i.e.}\text{, config knobs space} \end{aligned}$$

where c is one specific *configuration* of the operating system (*i.e.* values for `cgroups`) and for adaptors from the entire possible configuration space ($\mathbf{C}$).

The above defines an optimization problem over $\mathbf{C}$ with $|A_n|$ objectives and $|A_c|$ constraints, with the full solution existing on a Pareto front. The min operator enforces an upper bound on performance for each application, ensuring that once an app reaches its target, additional resources are not wasted on it. For apps without a strict target, this bound is effectively set to ∞. When no configuration that satisfies the constraints for the core apps (A_c), the optimization process outputs UNSAT, indicating none of the explored configurations are feasible, meaning that the robot developer has to either relax their constraints or provision more resource.

B. The ConfigBot Approach

Figure 4 outlines our framework with details provided next.

Specification and configuration space. The robot developer begins by defining performance metrics ($\mathbf{J}(c)$) for their apps, along with desirable target values (J^o) for each. For example, the developer might say that the navigation should run at a frequency (*i.e.*, $\mathbf{J}(c)$) of more than 35 Hz (*i.e.*, J^o). These specifications form the basis of the optimization problem. Our system then populates a list of available config knobs – we place each ROS node in a separate `cgroup`, allowing our system to control the `cpu.max` allocations for nodes independently. Additionally, we place adaptors on topics that (A) have high volumes of messages (*i.e.* > 5 Hz), **and** (B) that involve a non-core app either as a publisher or subscriber. We do this for for practical reasons (*i.e.* cut down on the number of config knobs to tune); without these conditions (A) and (B) in place even the **Basic-NAV** stack (Fig 1) would have 50+ adaptors making optimization unwieldy. This design choice is justified, as the primary goal of adaptors is to limit excessive resource usage by large data flows involving non-core apps.

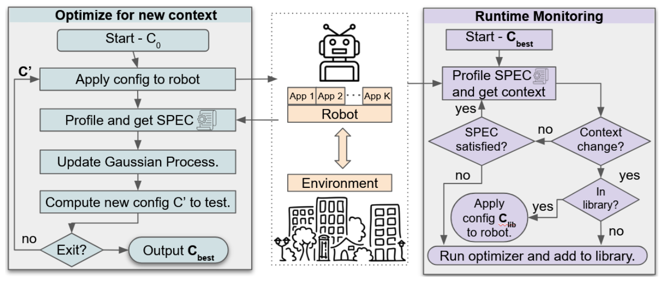


Fig. 4: Flowchart: How ConfigBot learns configs (given a specification – SPEC) for new contexts and performs runtime monitoring to identify when relearning is needed.

Initial learning. Using the provided specifications, ConfigBot models the requirements of core services as constraints and those of non-core apps as optimization objectives. As shown in the left half of Fig 4, ConfigBot leverages Bayesian optimization, well-suited for efficiently exploring configurations in high-dimensional, black-box scenarios. The optimization algorithm samples the config space ($\mathbf{C}$) to identify a config (C) that is applied to the system; we then profile the performance of our objectives and constraints under C for a brief amount of time (5s), after which we pass these observations to the optimizer; the optimizer uses this information to suggest a new point C'. This iterative process outputs the best config identified C_{best} (or UNSAT if no solutions are found).

Online monitoring and relearning. The right half of Fig. 4 illustrates the online monitoring process after a suitable configuration is identified and applied. ConfigBot uses a three-layer monitoring system including: (a) an eBPF [25] monitor that runs continuously, (b) a `rosmaster`-based monitor that wakes up every 30 seconds, and (c) a lightweight constraint monitor that continuously checks if the developer defined-spec is being met. More details are discussed through an illustration in the results section VIII-D. The eBPF and `rosmaster`-based monitors detect new processes and ROS nodes respectively, and raise warning flags before constraint violations are detected by the constraint monitor.

Optimizations. To minimize downtime during retraining, ConfigBot maintains a configuration library, mapping operational contexts (*e.g.* current location coordinates, list of apps) to optimal configurations, enabling rapid reuse of known good configs; a design pattern that other robot systems use as well [3]. When immediate retraining is not feasible (*e.g.* the robot is executing a critical task and cannot pause), we defer retraining by logging a trace of the current inputs (capturing sensor inputs via `rosbag`) and applying quick remediation by terminating non-core apps and reallocating their CPU shares to core services.

VII. Implementation

Robots. We implement ConfigBot on three different robots: (a) Spot: Boston Dynamics Spot w/ NVIDIA AGX Orin, (b) Jackal: Clearpath Jackal w/ Intel i7 (8 cores), and (c) Cobot: a mobile navigation base w/ Kinova arm, Intel NUC 10 (no GPU). We primarily use Spot for evaluating ConfigBot, though we also show that it generalizes effectively to other robots.

Stacks. We evaluate ConfigBot on six different sets of stacks across the three robots. For Spot, we have (i) **Basic**, which uses standard DWA based local planner for obstacle avoidance, and a carrot-based high level planner, (ii) **Intermed**, which adds an additional layer of discretized A-star grid-based planning in between the two layers described in **Basic**, and (iii) **Terrain**, that uses the terrain-aware navigation stack [11]. For Cobot, we have **CoNav** which is similar to **Basic** but adapted for Cobot, and **CoMap**, a manipulation stack based on top of Kinova Kortex API. For Jackal, we use **Phoenix** which is Army Research Laboratory's navigation stack [26]. In addition, we use these non-core apps: web dashboard (web) [12], object detection (obj) [24], segmentation (segment) and pose estimation (pose) [27].

Fig. 5: Robots we use: Spot, Cobot, and Jackal (left to right).

VIII. Evaluation

In this section, we show:

- A walkthrough of ConfigBot's learning process on **Basic**, illustrating how it discovers configurations that outperform default configurations, even when the default OS is equipped with 4× more resources (§VIII-A)
- Results on `Spot` demonstrating ConfigBot identifies good configurations for multiple robot stacks (§VIII-B)
- Demonstration of how the optimal configuration for one set of apps doesn't work for others (§VIII-C)
- Illustration of how ConfigBot handles runtime monitoring and retraining (§VIII-D)
- ConfigBot's generality to other robots (§VIII-E)

Metrics. We measure performance using two metrics: for core services, the ***constraint satisfaction rate*** measures the proportion of time in which all constraints are satisfied – ideally, we would want this to be 100%; for non-core apps, we use the developer-specified ***performance of non-core app*** to compare configs. For both these metrics, higher values are better, although a high non-core performance at the cost of a lower-satisfaction rate is undesirable.

Comparisons. We compare the performance of the ConfigBot-tuned configs with (a) the default config and (b) ConfigBot-cg, *i.e.* a ConfigBot without adaptors (tunes only `cgroup` settings), to quantify the relative improvements caused by adaptors.

A. ConfigBot: learning a config for ***Basic*** *on* `Spot`

Table IV below summarizes the configurations identified by ConfigBot, the satisfaction rate of those configs, and the non-core performance. ConfigBot performs nearly 3× better than the default config, while maintaining a near-perfect satisfaction rate. In fact, the default config fails to outperform the ConfigBot-tuned config on constraint satisfaction rate, *even when we provide the default with 4× more CPUs.*

Fig 6 plots the optimization progress of ConfigBot as it identifies the config in Table IV. In Fig 6, each point corresponds to one configuration tested by the optimizer; a red cross indicates that config fails to satsify the core service constraint. The optimizer chooses the highest performing

TABLE IV: ConfigBot on **Basic** with web

	Core (%CS)	Non-core (Hz)
ConfigBot	**99.42%**	9.90
ConfigBot-cg	93.76%	1.97
Default	0.00%	3.22
Default (4× CPUs)	84.94%	29.53

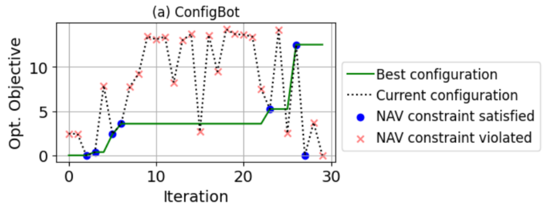


Fig. 6: Optimizing **Basic** with ConfigBot on `Spot`

blue-dot (*i.e.* constraints satsified) as the best config; the *green* solid line tracks the best config identified *so far*.

The constraint satisfaction rate is a binary metric - which tells us if a config satisfies the constraint, or doesn't; it doesn't inform us on how *close* it was to satisfying the constraint threshold. Fig 7 plots a histogram of the core services' performance and shows that the default config (10 – 15Hz) is far from the developer-defined performance constraint (35Hz).

B. Benchmarking ConfigBot performance on `Spot`

Table V summarizes the performance of ConfigBot in optimizing three different navigation stacks on `Spot`, while an additional resource-hungry non-core app (`obj`) is running. The number in (parentheses) represents the performance of the non-core app (`obj`) and %ages are satisfaction rates.

Table V illustrates how ConfigBot's config has a near 100% satisfaction rate for all three stacks while being able to perform between 49% (for **Intermed**) to 193% (for **Terrain**) better on the non-core tasks, when compared against default. Additionally, the addition of adaptors allows ConfigBot to discover configs that have as much as 92% (**Terrain**) better performance on non-core apps when compared against ConfigBot-cg. In addition to the usual comparisions, Table V also includes three randomly chosen configurations sampled uniformly at random from the config space – none of these configs preserved high constraint satisfaction while maximizing non-core performance to the extent ConfigBot-tuned configs were able to, illustrating the sparsity of good configs within the config space.

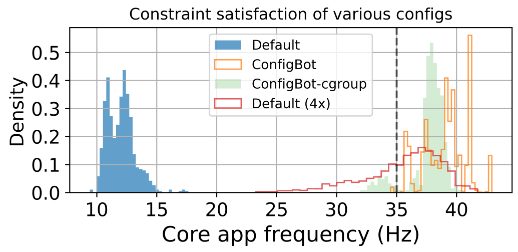


Fig. 7: Histogram: performance of core-services

TABLE V: Performance of **NAV** stack with **obj**.

	Basic	**Intermed**	**Terrain**
ConfigBot	**100% (2.12)**	**100% (1.79)**	96.0% (**1.35**)
ConfigBot-cg	**100%** (1.78)	**100%** (1.71)	**100%** (0.70)
Default	22.4% (1.45)	0.0% (1.20)	0.4% (0.46)
Random-1	99.3% (0.62)	0.0% (0.62)	99.4% (0.56)
Random-2	35.0% (1.04)	0.0% (0.95)	24.9% (1.04)
Random-3	86.9% (1.31)	60.2% (1.61)	39.2% (0.80)

C. Absence of a globally optimal config

Throughout this paper, we have been alluding to the benefits of constantly learning and relearning configurations for each new context. To illustrate the importance of relearning, we attempt to see if there exists an optimal config that works across all three of our stacks. Intuitively, a config we learn for a *more-resource intensive* core service (*e.g.* **Terrain**) should perhaps work well on simpler apps (*e.g.* **Basic**, **Intermed**).

Table VI evaluates the best config we identified for each app (*e.g.* **Terrain**$_c$) on the other two apps (*e.g.* **Basic**). We find that there is a clear one-one correspondence between the highest performing config on each stack. Specifically, the optimal config learned from **Basic** performs better on **Basic** than on **Terrain**, reinforcing the need to relearn configurations for each context, as no single config generalizes well.

TABLE VI: Cross-evaluation of optimal configs on stacks.

Config	Navigation Stack (running on robot)		
	Basic	**Intermed**	**Terrain**
Basic$_c$	**100% (2.12)**	85.7% (1.63)	69.2% (1.51)
Intermed$_c$	**100%** (1.86)	**100% (1.79)**	45.31 (**1.52**)
Terrain$_c$	99.4% (1.61)	4.4% (1.38)	**96.0%** (1.35)

D. Context-dependent relearning

Figure 8 demonstrates a scenario where `Spot` is initially navigating an indoor environment with **Basic**; then, it steps outdoors and switches to **Terrain** at time $t \approx 50s$, which is what it has been programmed to do. ConfigBot is able to detect this almost instantly through the use of an eBPF (Extended Berkeley Packet Filter) monitor which enables low-overhead, in-kernel monitoring of system activity and can detect new processes even before they initialize ROS (*e.g.* before `rospy.init()`). Our eBPF monitor dispatches a `wakeup` message to our second monitor, a `ROS`-Python node that uses the `rosmaster` API, which would otherwise check for new nodes every ∼ 30 seconds. Due to the `wakeup`-call, our `rosmaster`-based profiling identifies the new ROS nodes less than 2s after they are initialized; at this stage, we could either start the reoptimization process (or reuse a config from the library, if this context has been seen before); however, for illustrative purposes, we don't do so, and wait for constraint violations to accumulate over a period

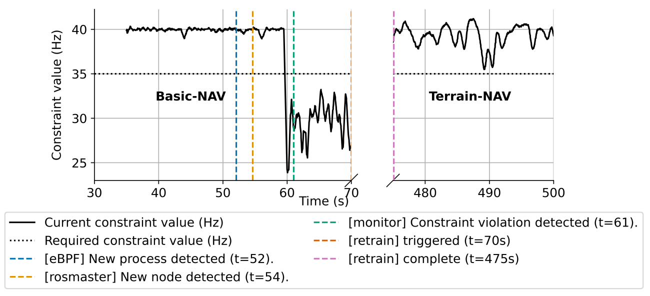


Fig. 8: ConfigBot runtime monitoring + relearning

of time before our third trigger (the constraint monitor) starts the retraining.

E. Generalization to other robots

Table VII summarizes the results of running a completely different set of stacks on Jackal and Cobot. ConfigBot clearly outperforms the Default config, which validates its general and widespread applicability.

TABLE VII: Performance on Cobot and Jackal.

Robot	Core (non-core)	ConfigBot	Default
Cobot	**CoNAV** (pose)	**100%** (**0.78**)	47.1% (0.45)
Cobot	**CoMap** (pose)	**100%** (**2.47**)	54.3% (1.67)
Jackal	**Phoenix** (web)	**100%** (2.32)	7.2% (**6.71**)
Jackal	**Phoenix** (segment)	**98.2%** (**0.76**)	0.0% (0.42)

IX. Conclusion and Future Work

We presented ConfigBot, an automated configuration tuning system designed to dynamically reconfigure service robots to meet predefined performance specs, and demonstrated it's ability to maintain system stability across a range of challenging scenarios. Looking ahead, we identify two key directions for future work. First, automating the process of defining performance specifications remains an open challenge. Developing methods to infer these specifications directly from high-level task descriptions could significantly reduce the burden on developers. Second, while ConfigBot currently reacts to changes in the environment or application demands, incorporating reinforcement learning (RL) could enable proactive management of resources. By predicting potential performance bottlenecks or resource conflicts before they occur, an RL-driven controller could anticipate problems and adapt configurations preemptively.

References

[1] D. Han, T. McInroe, A. Jelley, S. V. Albrecht, P. Bell, and A. Storkey, "Llm-personalize: Aligning llm planners with human preferences via reinforced self-training for housekeeping robots," *arXiv preprint arXiv:2404.14285*, 2024.
[2] S. Modak, N. Patton, I. Dillig, and J. Biswas, "Synapse: Learning preferential concepts from visual demonstrations," *arXiv preprint arXiv:2403.16689*, 2024.
[3] X. Xiao, B. Liu, G. Warnell, J. Fink, and P. Stone, "Appld: Adaptive planner parameter learning from demonstration," *IEEE Robotics and Automation Letters*, vol. 5, no. 3, pp. 4541–4547, 2020.
[4] Z. Hu, *et al.*, "Deploying and evaluating llms to program service mobile robots," *IEEE Robotics and Automation Letters*, 2024.
[5] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "{CherryPick}: Adaptively unearthing the best cloud configurations for big data analytics," in *14th USENIX Symposium on Networked Systems Design and Implementation*, 2017, pp. 469–482.
[6] M. Bin-Yahya, *et al.*, "{Config-Snob}: Tuning for the best configurations of networking protocol stack," in *USENIX Annual Technical Conference*, 2024.
[7] L. Zhang and M. A. Babar, "Automatic configuration tuning on cloud database: A survey," *arXiv preprint arXiv:2404.06043*, 2024.
[8] G. Somashekar, *et al.*, "OPPerTune:Post-Deployment Configuration Tuning of Services Made Easy," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024, pp. 1101–1120.
[9] A. Karthikeyan, *et al.*, "{SelfTune}: Tuning cluster managers," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 1097–1114.
[10] A. Binch, G. P. Das, J. P. Fentanes, and M. Hanheide, "Context dependant iterative parameter optimisation for robust robot navigation," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2020, pp. 3937–3943.
[11] H. Karnan, E. Yang, D. Farkash, G. Warnell, J. Biswas, and P. Stone, "Self-supervised terrain representation learning from unconstrained robot experience," in *ICRA Workshop on Pretraining for Robotics*, 2023.
[12] K. S. Sikand, L. Zartman, S. Rabiee, and J. Biswas, "Robofleet: Open source communication and management for fleets of autonomous robots," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2021, pp. 406–412.
[13] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 455–466, 2014.
[14] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 153–167.
[15] Y. Cai, *et al.*, "Learning delicate local representations for multi-person pose estimation," in *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part III 16*. Springer, 2020, pp. 455–472.
[16] S. Huang, M. Gong, and D. Tao, "A coarse-fine network for keypoint localization," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 3028–3037.
[17] L. Zheng, M. Tang, Y. Chen, G. Zhu, J. Wang, and H. Lu, "Improving multiple object tracking with single object tracking," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021.
[18] F. Nenci, L. Spinello, and C. Stachniss, "Effective compression of range data streams for remote robot operations using H.264," in *IEEE/RSJ International Conf. on Intelligent Robots and Systems*, 2014.
[19] L. Liu, H. Li, and M. Gruteser, "Edge assisted real-time object detection for mobile augmented reality," in *The 25th annual international conference on mobile computing and networking*, 2019, pp. 1–16.
[20] J. Ichnowski, *et al.*, "Fogros2: An adaptive platform for cloud and fog robotics using ros 2," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2023, pp. 5493–5500.
[21] A. Francis, *et al.*, "Principles and guidelines for evaluating social robot navigation algorithms," *arXiv preprint arXiv:2306.16740*, 2023.
[22] W. Tang, L. Zheng, M. Jiang, and Z. Ding, "Dynamic algorithm selection for mobile robots motion planning," in *6th International Conference on Dependable Systems and Their Applications*, 2019.
[23] C. S. Pabla, "Completely fair scheduler," *Linux Journal*, vol. 2009, no. 184, p. 4, 2009.
[24] T. Ren, *et al.*, "Grounded sam: Assembling open-world models for diverse visual tasks," *arXiv preprint arXiv:2401.14159*, 2024.
[25] eBPF Community, "eBPF: Extended Berkeley Packet Filter," https://ebpf.io/, 2025, accessed: 2025-03-01.
[26] "Phoenix stack army research lab," 2025. [Online]. Available: https://arl.devcom.army.mil/cras/sara-cra/sara-overview/
[27] R. Khanam and M. Hussain, "Yolov11: An overview of the key architectural enhancements," *arXiv preprint arXiv:2410.17725*, 2024.