tinyKube: A Middleware for Dynamic Resource Management in Cloud-Edge Platforms for Large-Scale Cloud Robotics

Chanh Nguyen*, Eunil Seo*, Muhammad Zahid†, Oliver Larsson*, Florian T. Pokorny†, Erik Elmroth*
*Department of Computing Science, Umeå University, 90187 Umeå, Sweden
Email: {chanh, eunil.seo, olars, elmroth}@cs.umu.se
†School of Electrical Engineering and Computer Science, KTH Royal Institute of Technology, 10044 Stockholm, Sweden
Email: {mzmi, fpokorny}@kth.se

***Abstract*—With advancements in ubiquitous networking infrastructure and the increasing availability of data centers, ranging from local edge servers to remote cloud data centers, integrating robots with edge-cloud computing resources holds significant potential. However, introducing additional layers of computing brings significant challenges, particularly in ensuring adequate resource allocation and efficiently scheduling resources across a distributed infrastructure to meet the performance demands of robotics applications.**

In this paper, we present tinyKube, a middleware tailored for dynamic resource management across the cloud-edge platform for large-scale cloud robotics deployments. tinyKube leverages the advantage of Kubernetes for resource orchestration and integrates Prometheus for resource monitoring, facilitating seamless coordination across multiple distributed data centers, enabling unified monitoring, task scheduling, and resource provisioning across both edge and cloud infrastructures.

We conduct experiments using a sample robotic manipulation compute workload requiring both object detection and motion planning and using the Cloud Robotics testbed CloudGripper and a real-world cloud-edge computing infrastructure. Our experimental results indicate that the proposed middleware is capable of automatically managing task dispatching and resource allocation in response to varying quality of service requirements and workload fluctuations from large-scale robotic systems. tinyKube aims to simplify resource management in the robotic application development process, accelerating testing and deployment for large-scale cloud robotics and facilitating more efficient real-world implementation of robotic applications.

***Index Terms*—Cloud Robotics, Cloud-Edge Infrastructure, Resource Orchestration, Performance Monitoring, Middleware.**

I. Introduction

A. Cloud Robotics - Leveraging Cloud Resources for Advanced Robotic Systems

In recent years, the utilization of mobile robots has experienced a significant upsurge, representing a critical milestone in their widespread adoption [1]. However, many of these robots still rely predominantly on classical control mechanisms or are controlled remotely by humans. This limited intelligence is in part due to the high costs associated with onboard computation and storage required for state of the art machine learning-enabled applications, impacting both the affordability of robots and their mobility and operational capacity due to increased size and weight requirements.

Advancements in wireless technology and the expanding availability of data centers – from local edge nodes to remote cloud facilities – have unlocked substantial potential for integrating robots with edge-cloud computing resources. This integration brings several key benefits, including but not limited to [2], [3], [4]: 1) *offloading heavy computational tasks* to edge and cloud servers, which increases application performance, extends battery life, and allows for more compact and lightweight robot designs by minimizing additional hardware requirements; 2) *resource elasticity*, enabling dynamic adjustment of computing resources based on demand, optimizing both efficiency and scalability; and 3) *controllable privacy enhancement*, which allows for the deployment of application services while offering flexible control over where and how sensitive sensor data is transmitted.

In fact, the term 'Cloud Robotics' was introduced over a decade ago [5], [6], embodying the vision of robots harnessing the power of cloud computing infrastructure. Since then, it has garnered significant attention: in industry, platforms such as Amazon RoboMaker [7] and NVIDIA Omniverse [8] provide environments for simulating and managing robots that operate in cloud-connected systems, while in academia, projects like RoboEarth [9], BRASS [10], Dex-Net [11], Rapyuta [12], and more recently FogROS and FogROS2 [13], [14], which facilitate cloud and fog robotics compatible with the Robot Operating System (ROS), have further advanced the field. Researchers have explored utilizing cloud resources to improve robot performance, citing benefits such as reduced SLAM latency [14], [15] and faster motion planning [16], [17] and privacy-preserving cloud-enabled robotic manipulation methods have been developed [18]. Simultaneously, early research on the economic benefits of using cloud resources for robotic applications [19] suggests the potential to not only reduce significant upfront capital expenditure (CapEx), but also to extend the robot's service life and optimize resource utilization.

B. Challenges in Resource Management for Large-Scale Cloud Robotics

The integration of additional computing layers, particularly within large-scale, distributed cloud and edge comput-

ing infrastructures, presents significant challenges for robotic ecosystems. In cloud computing environments, where resources are virtually unlimited but operate under a pay-as-you-use model, the traditional robotics approach of utilizing maximum available computing power to enhance robot performance becomes both resource-inefficient and cost prohibitive [20], [21]. The critical challenge lies in balancing performance improvements with the financial costs associated with leveraging additional computational resources. While existing research has largely focused on the initial configuration of cloud resources for robotics deployments [22], [23], it often neglects the dynamic nature of cloud and edge environments, fluctuating workloads, and the mobility of robots [24].

Consider a fleet of robots operating across various workspaces (e.g., factories, hospitals, airports) that rely on compute-intensive services (e.g., object detection, motion planning) deployed on a distributed platform comprising edge servers and remote cloud data centers. For optimal resource utilization, resource allocation to these services should dynamically adjust based on demand – scaling up during periods of high request volume and scaling down when demand decreases. Furthermore, robots in certain sensitive environments, such as hospitals, may require enhanced privacy measures, ensuring that data is transmitted only to application services deployed on servers with high-privacy guarantees.

Current robot applications built with ROS (Robot Operating System) rely on the ROS Master (in ROS 1) [25] or the Data Distribution Service (DDS)[1] in the latest ROS 2 [26] for node (i.e., abstractions of computational units) discovery and managing connections between nodes. However, this architecture presents drawback when distributing ROS-based robotic application nodes across distributed edge-cloud platforms: 1) the use of the *pub/sub protocol*, where publishers (e.g., camera nodes) continuously transmit data without awareness of whether subscribers (e.g., object detection nodes) are available to process it, can lead to unnecessary network bandwidth usage, increased congestion, and potential failures [23], [27]; 2) while DDS provides various Quality of Service (QoS) policies – such as *latency budget* (ensuring timely data delivery, e.g., 50 ms), *deadline* (ensuring the subscriber processes data within a specific time frame, e.g., 100 ms), *reliability* (ensuring all data reaches the subscriber without loss), etc. – it is not inherently designed to interact with cloud orchestration layers. As a result, although DDS can raise alerts or flag QoS violations, it lacks the capability to dynamically provision or scale cloud resources to meet the specified QoS requirements [28], [29].

In the context of cloud-native computing (CNCF)[2], the *Kubernetes*[3] platform has gained widespread adoption for automating the deployment, scaling, and management of containerized applications. Concurrently, *Prometheus*[4], an open-source monitoring and alerting toolkit, has become the de facto standard for monitoring cloud-native applications, particularly those deployed within Kubernetes clusters. However, Kubernetes lacks native support for managing multi-cluster environments, such as those spanning from edge servers to cloud data centers. To integrate resources across multiple clusters, extensions like KubeFed [30] and OpenShift [31] have been developed. However, these solutions introduce significant complexity in setup and configuration, as well as operational overhead for maintenance, which can slow down testing and development of cloud robotics applications.

To address these challenges, we propose *tinyKube*, a middleware that selectively utilizes the neccessary capabilities of Kubernetes and Prometheus to provide a lightweight yet comprehensive platform for monitoring, task dispatching, and resource provisioning across distributed edge and cloud data centers for large scale cloud robotic deployments. tinyKube considers not only traditional QoS requirements (such as response time, latency) but also privacy concerns when making task dispatching decisions, aiming to simultaneously optimize QoS adherence and resource utilization. The primary goal is to provide a toolkit that reduces the complexity of testing and deploying robotic applications at scale, while also streamlining the implementation and evaluation of resource provisioning strategies for these applications.

We evaluate tinyKube using a robotic application developed for the CloudGripper system [32][5], a globally accessible Cloud Robotics Testbed consisting of 32 small robotic arms and parallel jaw grippers. In experiments we test our approach by deploying compute-intensive services, including object detection and motion planning, on a real cloud-edge infrastructure. The experimental results indicate that tinyKube dynamically scales resources across both edge and cloud clusters in response to varying robot request arrival rates, adhering to the logic defined in the scaling mechanism. Additionally, it efficiently balances and dispatches robot requests to ensure compliance with Quality of Service (QoS) requirements – particularly with respect to response time and privacy – while minimizing the rejection rate.

In summary, this paper presents the following key contributions:

- tinyKube – a lightweight middleware designed to automate task dispatching and resource allocation for large-scale robotics applications deployed across cloud-edge computing platforms (Section III).
- Comprehensive experimental evaluations of tinyKube's performance, conducted using a large-scale robotic gripper system on a cloud-edge platform, demonstrating its efficiency in dynamic resource management (Section IV and V).

II. Related Work

Cloud robotics, first introduced over a decade ago, has rapidly evolved with the advancements in cloud computing and wireless network technologies [33], [34], [35], [36]. Today, propelled by the fourth industrial revolution (Industry 4.0),

[1]https://www.omg.org/omg-dds-portal/
[2]https://www.cncf.io/
[3]https://kubernetes.io/
[4]https://prometheus.io/
[5]https://cloudgripper.org

large-scale robot deployments (also known as fleet robotics or multi-agent robotic systems) are beginning to find widespread application across various domains: in smart factories, where robots and humans collaborate throughout industrial value chains [37], [38]; in healthcare systems supporting elderly care [39]; in agriculture smart farming systems [40]; and in disaster management, where unmanned search and rescue operations are conducted in hostile environments [41]. A common aspect of these deployments is the ability of robots to offload computational tasks – such as SLAM [14], [15], motion planning [16], [17], and object recognition [42] – to the cloud, enabling collaborative efforts to achieve shared objectives.

To support the offloading of computation-intensive robotic tasks across edge and cloud infrastructures, several toolkits and platforms have been proposed. Rapyuta [12], developed by the Swiss Federal Institute of Technology Zurich, is one of the first open-source Platform-as-a-Service (PaaS) frameworks specifically designed to help robotic applications offload heavy, non-real-time computations to secure and customizable cloud computing environments. Rapyuta's core tasks include: 1) the controller task, which manages the command data structure, oversees robot connections, handles configuration requests, and monitors network status; 2) the robot task, responsible for forwarding configuration requests to the master, converting data messages, and facilitating communication between robots and endpoints; and 3) the environment task, which handles communication with ROS nodes and other endpoints, and manages the launching and stopping of ROS nodes.

FogROS1 [13] and FogROS2 [14], developed at UC Berkeley, have gained popularity for supporting ROS-based and ROS2-based applications, respectively. These adaptive cloud robotics platforms, as described in [14], are capable of provisioning and launching cloud resources, configuring and securing network communications, installing robot code and dependencies, and initiating robot and cloud robotics operations. Extensions like FogROS2-config [22] and FogROS2-LS [23] further enhance these capabilities by helping robotics systems select efficient hardware configurations that balance cost and latency, and by transitioning to optimal service deployments that meet latency requirements. Nonetheless, the manual setup and explicit component placement required by FogROS2, which focuses on individual robots, limits its scalability for larger or multi-robot systems [24].

Researchers from Karlsruhe University of Applied Sciences introduced KubeROS [43], an automated platform that leverages Kubernetes as its underlying orchestration framework for deploying multi-robot applications. KubeROS abstracts onboard computing devices, edge, and cloud resources into a unified infrastructure, offering developers a seamless platform for managing robotic systems. It also features monitoring and a resource scheduler that supports dynamic rescheduling based on the system's real-time state. The KubeROS API Server further supports REST services to handle and process deployment requests. One disadvantage of KubeROS is its complex hardware and network setup, which requires intervention from a system administrator for management.

In this paper, we introduce tinyKube, a lightweight middleware designed for the seamless integration of distributed resources across edge-cloud infrastructures to support large-scale robotics deployments. tinyKube leverages Kubernetes for orchestration and Prometheus for monitoring, providing an integrated platform for efficient system monitoring, task dispatching, and resource allocation. Designed specifically for the rapid testing and deployment of robotic applications, tinyKube simplifies network setup and minimizes configuration requirements, thereby lowering the operational overhead for developers.

III. TINYKUBE DESIGN

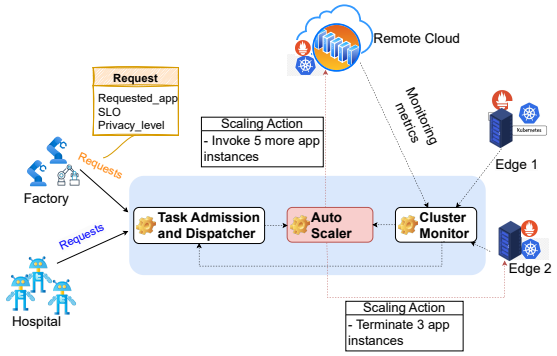

Fig. 1. The tinyKube middleware design.

Figure 1 illustrates the components of the tinyKube middleware and its use case in managing resources across heterogeneous, geographically distributed edge servers and cloud data centers for cloud robotics applications. The middleware design includes the following key components: Cluster Monitor, Task Admission and Dispatcher, and Auto Scaler. We assume that each resource node has Kubernetes and Prometheus deployed and thus refer to it as a 'cluster' (in line with common Kubernetes terminology, where a cluster represents a set of nodes managed as a single unit).

A. Cluster Monitor – Monitoring Across Multiple Clusters

The Prometheus service in each cluster periodically scrapes and stores metrics in a local time-series database (TSDB) [44]. These metrics include system metrics (e.g., CPU, memory, and filesystem usage), application metrics (e.g., error rates, throughput), Kubernetes-specific metrics (e.g., pod and node status), and custom metrics defined by applications.

However, querying individual Prometheus instances via the API presents several challenges in multi-cluster cloud-edge environments: 1) *lack of scalability*, as Prometheus is designed as a single-node system and cannot query multiple instances simultaneously; 2) *lack of a global view*, as Prometheus lacks native support for aggregating metrics across clusters into a unified interface; 3) *lack of long-term storage*, as Prometheus is optimized for short-term retention, making it difficult to store data for extended periods without running into disk space

limitations or performance degradation as the data volume increases.

To address these challenges, we integrated Thanos [45], a Prometheus federation toolkit, as the core of the middleware's cluster monitor component. In essence, Thanos aggregates metrics from multiple Prometheus instances into a single queryable interface, providing a global system view and enabling seamless, scalable monitoring across clusters without the need for reconfiguration as new clusters are added.

Figure 2 depicts the integration of Thanos within the cluster monitor component. In each cluster, the *Thanos Sidecar* container runs within the same pod as the Prometheus instance, collecting metrics and exposing them to the central *Thanos Querier* over the internet using the gRPC (i.e., google Remote Procedure Call) protocol. When adding a new cluster to the platform, the Thanos Querier configuration simply needs to be updated with the Thanos Sidecar endpoint from the new cluster (i.e., `--store = <store-api>:<grpc-port>`), as highlighted in the red rectangle in Figure 2.

Furthermore, Thanos enables long-term metric storage by offloading data to cloud-based object storage (e.g., AWS S3, Google Cloud Storage). These archived historical data are crucial for evaluating system performance and optimizing strategies for application deployment and resource allocation.

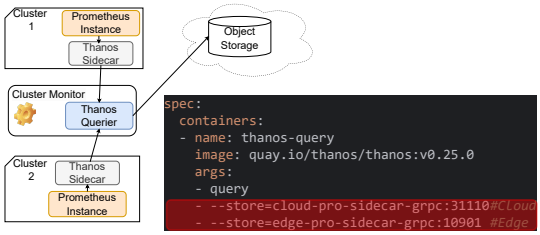

Fig. 2. Cluster monitor component.

Finally, the cluster monitor component queries and aggregates metrics across multiple clusters using the HTTP API exposed by the Thanos Querier, which processes PromQL (Prometheus Query Language) queries and returns data in JSON format. The real-time monitoring data is then fed into the Task Admission and Dispatcher, as well as the Auto Scaler, guiding the middleware's decisions on task dispatching and resource allocation. For example, if the metrics show that a particular cluster is under heavy load and unable to meet its service level objectives (such as response time or throughput), the middleware can automatically scale resources or redistribute tasks to other clusters, ensuring load balancing and maintaining quality of the service.

B. Task Admission and Dispatcher

Figure 3 depicts the task admission and dispatcher component in detail. Robots from various workplaces submit task requests by sending metadata to the middleware via the standard web-based REST API [46] communication protocol. This metadata specifies the requested application service (e.g., motion planning, object detection), defined service level objectives (SLOs) such as response time, privacy preferences (e.g., requests favoring local edge processing for enhanced privacy), and other parameters critical to task dispatching and resource allocation decisions. The request parser processes each request to extract SLO information, which is then forwarded to the estimator and dispatcher for further handling.

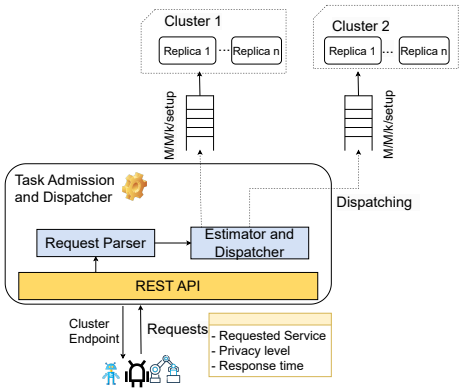

Fig. 3. Task admission and dispatcher component.

We model each cluster member involved in the cloud-edge platform using an *M/M/k/setup* queuing system [47], [48], where request arrivals follow a Poisson process, service times are exponentially distributed, and k represents the number of application instances (i.e., replicas) within the cluster. By incorporating setup times – the time required to start a new replica within the cluster – into the total waiting time estimation, the model supports more proactive scaling to meet rising demand. This approach closely reflects real cloud system behavior and is well-suited for robotic applications, where minimizing processing delay is critical.

We denote by T_{setup} the average setup time for an application instance, λ denotes the request arrival rate, μ is the service rate (number of requests handled by the instance per time unit), L_q denotes the average number of requests in the queue, and k is the number of active application instances. The effective service rate, accounting for the setup time, is given by:

$$\mu_{eff} = \frac{\mu}{1 + T_{setup}\lambda} \tag{1}$$

The queue waiting time for a request is then calculated as:

$$W_q = \frac{L_q}{\lambda(1 - \lambda\mu_{eff}/k)} \tag{2}$$

Finally, the total time from when the request arrives until it is fully serviced and completed is calculated as:

$$W = W_q + \frac{1}{\mu_{eff}} \tag{3}$$

Requests are admitted and dispatched to the cluster to ensure that W remains within an acceptable range, meeting the response time requirements specified in the request's Service Level Objective (SLO) for both privacy-sensitive requests

(which are prioritized for edge processing), and normal requests.

Once the decision is made, the middleware communicates the endpoint information of the assigned cluster (i.e., `<cluster_exposed_IP>:<port>/<service>`) to the robot via a REST API POST request. The robot then connects to the designated cluster and transfers the necessary data (e.g., camera frames for object recognition) to execute the requested application service.

C. Auto Scaler

The Auto Scaler component utilizes real-time metrics from the cluster monitoring system, including resource utilization (CPU, memory), cluster queue lengths, and the Service Level Objectives (SLOs) of incoming requests. Based on this, the Auto Scaler dynamically adjusts resource allocation across clusters within the cloud-edge infrastructure, with the aim of optimizing application performance and resource efficiency.

Through the Kubernetes API, the Kubernetes Horizontal Pod Autoscaler (HPA) can be configured to define a scaling strategy. However, HPA operates on a fixed polling interval, potentially introducing delays in scaling responses as it checks metrics only at specific intervals (e.g., every 15 seconds). To address this, our middleware incorporates the Kubernetes Event-Driven Autoscaler (KEDA)[6], which responds immediately when a metric threshold is crossed, enabling more responsive scaling – particularly critical for robotic applications that require rapid adjustments.

```yaml
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
...
spec:
  scaleTargetRef:
    name: placeholder-deployment
  minReplicaCount: 1
  maxReplicaCount: 10
  cooldownPeriod: 300
  triggers:
    - type: prometheus
      metadata:
        serverAddress: "http://thanos-querier:9090"
        metricName: placeholder-metric
        threshold: placeholder-threshold
        query: "sum(rate(placeholder-metric[1m]))"
```

Listing 1. KEDA ScaledObject YAML Configuration.

Listing 1 provides a customizable template for configuring a KEDA ScaledObject serving as a flexible *placeholder* that enables "robotic service providers" to define scaling policies aligned with their specific objectives and operational requirements.

IV. Experiment Setting

In this section, we present the experimental setup used to evaluate the efficiency of the proposed middleware in managing task dispatching and resource allocation across the cloud-edge infrastructure. First, we describe the cloud robotics testbed utilized in our experiments where compute-intensive services are offloaded to cloud-edge resources to assist a robotic gripper in picking up and moving objects to a designated position. We then introduce the configurations of both the edge and cloud clusters within the experimental platform. This is followed by the experimental scenarios, where we vary the number of robots and request rates. Since these experiments focus on the computational scaling of the pipeline, robots are not physically moved during these experiments, but real-world pre-recorded sensor data from the real system is used as input to the algorithms, allowing us to scale the number of robots in experiments beyond the number of physically available robots on the platform. Finally, we outline the metrics used to evaluate the results.

A. The CloudGripper System

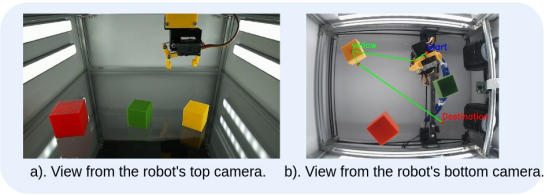

Fig. 4. (a) Top camera view and (b) bottom camera view of the robot. The object detector is trained using the frames captured by the bottom camera. The green line indicating the robot's path to pick and drop the object.

We use the CloudGripper testbed[7] [32], currently comprising 32 remotely accessible small robot arm workcells located at KTH Royal Institute of Technology, Sweden. We designed sample algorithms to test real-world compute loads for this system where the robotic arms are tasked with identifying and picking up objects of interest and transporting them from their current position to a predefined target position in the robot's workspace. To facilitate this process, we implemented an application service that integrates a computer vision model for object detection and a motion planning system to compute suitable collision-free paths for object delivery. A benefit of this pipeline is that it provides a real use case in the sense that it can be executed in real-time on the robots, resulting in the physical task execution and motion of the robotic system but it also allows us to separately focus on studying the computational scaling of our proposed approach by using offline collected input data from the system (i.e. camera images) which we are most interested in investigating in this work. Since the time required for physical motion of the robot is not our primary interest in this study, our experiments employs this offline approach in the following experiments (i.e. robots do not execute the computed motions physically, but real-world camera images are used to evaluate the computational workloads for this sample task).

The computer vision model is trained using EfficientDet [49] to recognize objects of interest and obstacles from the input frames provided by the robot's bottom camera, while the motion planning service utilizes the Rapidly-exploring

[6]https://keda.sh/

[7]https://cloudgripper.org/

Random Tree (RRT) algorithm [50] to compute a collision-free path for object delivery. Due to the limited computational capabilities onboard the robot arm, the application service is offloaded to more powerful computational resources from the cloud-edge infrastructure. To enable this, we created a Docker image of the application service and hosted it in a Docker repository[8], allowing the cluster to instantiate the application on demand by launching a new instance from this image. During each live task execution, the robot arm transmits a frame captured by its bottom camera to the service for processing. Figure 4 shows the top and bottom camera views of the robot during the grasping process. In the offline setting we use to study the compute-load scaling, the camera image is instead pre-recorded and loaded locally from disk.

B. Cloud-edge platform

We configured a cloud-edge platform consisting of a local edge server and a remote cloud server. Both servers are equipped with the Kubernetes platform for orchestrating containerized applications and Prometheus for monitoring and metrics collection, forming what we refer to as clusters.

The edge cluster consists of 32 Raspberry Pi 4B units, each powered by a Quadcore Cortex-A72 processor running at 1.8 GHz [32]. The cluster is hosted at KTH and integrated with the CloudGripper system. The average setup time for an application instance (i.e., a container), from scheduling to being ready for use on this cluster (which uses the ARM64 architecture), is measured as $T_{setup} = 1.8$ seconds.

The remote cloud cluster utilizes resource capacity from the Ericsson Cloud[9], includes 19 CPU cores and 36 GB of memory, which are dedicated specifically to the research project[10]. The average setup time for an application instance, from scheduling to being ready for use on this cluster (which uses the AMD64 architecture), is measured as $T_{setup} = 4.7$ seconds.

Additionally, we assume that the edge cluster provides higher privacy guarantees (i.e., 1), as it is on the same network as the robot arms, while the remote cloud cluster offers lower privacy guarantees (i.e., 0). To register these clusters within the integrated platform, the process simply involves declaring each cluster's endpoint, as shown in Listing 2.

```
clusters:
  edge_clusters:
    - name: "edge"
      endpoint: "http://localhost:8080/detect"
      max_capacity: 17        # Max number of pods
      privacy_level: 1        # High privacy guarantee

  cloud_clusters:
    - name: "cloud"
      endpoint: "http://xxx.xxx.xxx.xxx:yyyyy/detect"
      max_capacity: 19
      privacy_level: 0        # Low privacy guarantee
```

Listing 2. Cluster declaration.

[8]https://hub.docker.com/

[9]https://xerces.ericsson.net/

[10]https://wasp-sweden.org/nest-project-cloud-robotics/

Since the robots and the edge server are on the same network, the network delay between them is negligible. The external network connection from KTH has a bandwidth of 10 Gbps and an estimated round-trip time of 15.5 ms to the Ericsson cloud.

We conducted a benchmarking analysis of the application service deployed on both the edge cluster and the remote cloud cluster to profile its response time for a single request. Based on the 90th percentile of the measured response times [51], we set the Quality of Service (QoS) thresholds: 0.35 seconds for requests processed by the edge deployment and 0.50 seconds for those handled by the remote cloud. Table I summarizes the capacity of the two clusters.

TABLE I
CONFIGURATION OF THE TWO CLUSTERS.

Cluster	**Capacity** #max pods	**Privacy**	**Response Time (s)**	**T_setup (s)**
Edge	17	1	0.35	1.8
Cloud	19	0	0.5	4.7

C. Design of a Straightforward Auto-Scaler

For experimental purposes, we developed a straightforward auto-scaling strategy to manage resource provisioning across the cloud-edge platform in response to workload variations. The resource scaling strategy operates as follows:

We have set the threshold for average CPU utilization across the application instances at 80%. When the average CPU usage exceeds this threshold, the system triggers a scale-up operation. The configuration allows for quick scaling, which increases the number of application instances by up to 100% of the current number of instances within a 30-second window. Furthermore, a stabilization window of 60 seconds ensures that the system waits for sustained CPU load increases before adding more replicas, thus avoiding rapid fluctuations caused by temporary spikes.

Conversely, if the CPU utilization falls below the threshold, the resource scaler reduces the number of replicas to optimize resource usage. The scale-down policy permits a reduction in the number of resources by up to 50% of the current replicas in a 60-second period. To avoid overreacting to transient drops in demand, a stabilization window of 5 minutes is applied, ensuring that the reduction of replicas occurs only when there is a consistent decline in CPU usage, thus promoting system stability.

In the experiment, each application instance is a single container with specified resource allocations: a minimum of 500m CPU (0.5 cores) and 512Mi (∼537 MB) of memory.

At the beginning of the experiment, we initialized both clusters with 5 replicas each.

D. Scenarios

We designed scenarios in which task requests are randomly generated. Each request sent to the middleware includes two key parameters: a privacy level (1 or 0) and an expected

response time between 0.35 and 0.5 seconds, guiding task scheduling and dispatching decisions on where to process the request within the cloud-edge platform. Specifically, the number of simulated concurrent robot instances is randomly selected between 5 and 50. The experiment evaluates how the middleware enhances system performance, focusing on the response time of backend tasks deployed on cloud-edge resources under varying workloads. A pre-recorded dataset of camera images from the CloudGripper system is utilized as input for this experiment without physical motion of the resulting computed motion patterns of the robot arms, allowing us to scale the number of simulated robot instances beyond the available 32 physical robot arms in the CloudGripper system. The experiment includes dynamic resource provisioning and request dispatching with privacy considerations. To stress test the system, we scale up to 50 concurrent simulated robots and utilize a program that emulates real-time requests, while request arrival rates range from 20 to 100 requests per second. For each scenario, the total number of requests is capped at 5,000. Once this limit is reached, the system transitions to a new scenario with a new combination of concurrent robots and request arrival rate.

E. Evaluation metrics

We present the following metrics to evaluate the results:

- The average response time of accepted requests: This metric is based on the response time collected for each request. It allows us to evaluate whether the accepted requests, once dispatched for processing in the cloud-edge platform, violate the Quality of Service (QoS) thresholds in terms of response time.
- The resource allocation behavior throughout the experiment, which involves varying numbers of robots, request arrival rates, and random request patterns: This metric helps us evaluate how the system responds in terms of scaling resources within the cloud-edge platform, considering the fluctuations in workloads.
- The task dispatching outcomes: This metric evaluates how the middleware dispatches requests while respecting their QoS requirements for both privacy level and response time. It includes measuring the percentage of requests accepted and where they are processed, as well as the percentage of requests rejected when the system cannot meet their requirements.

V. Experimental Results and Discussion

We conducted experiments and collected results to analyze how robot requests are dispatched across the cloud-edge infrastructure and to examine the system's behavior in terms of resource allocation, following the logic of the designed auto-scaler. The results presented below were collected over a 2-hour experimental window.

A. The resource allocation behavior throughout the experiment

Figure 5 shows the average CPU utilization (blue line) and the total number of active replicas (red line) in the cloud cluster at each time slot throughout the experiment. Similarly, Figure 6 presents these metrics for the edge cluster over the same time period.

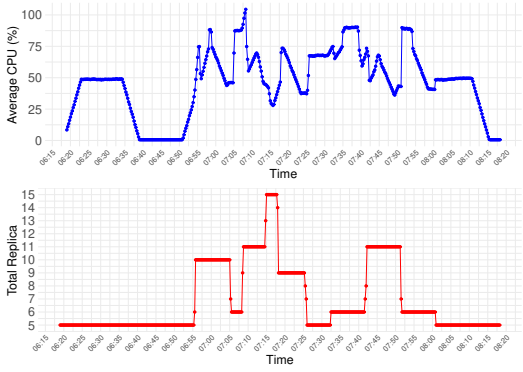

Fig. 5. Number of replicas and average CPU utilization (percentage) on the Cloud cluster throughout the experiment.

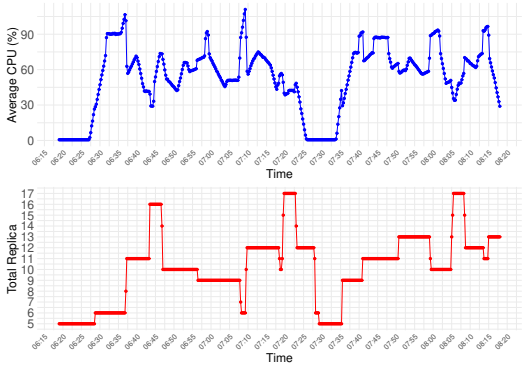

Fig. 6. Number of replicas and average CPU utilization (percentage) on the Edge cluster throughout the experiment.

Initially, the total number of replicas in each cluster was set to 5. As the request arrival rate and the number of concurrent robots increased, the higher influx of requests led to a rise in CPU utilization across the active replicas. Once utilization exceeded the predefined threshold, the auto-scaler triggered a scaling action, adding more instances to the clusters.

Gradually, as resource utilization stabilized, the system began to trigger a scaling-down action, in accordance with the auto-scaler strategy outlined earlier. Both graphs also demonstrate faster scale-up behavior compared to scale-down, which aligns with the 30-second window for scaling up and the 60-second window for scaling down as specified in the settings of the auto scaler.

Furthermore, as observed in two Figure 5 and 6, the edge cluster appears to be more active than the cloud cluster. This can be attributed to two factors: first, task dispatching always directs privacy-sensitive requests to the edge cluster, as it ensures compliance with privacy requirements. Second, the task dispatching mechanism prioritizes available resources in the edge cluster to process other normal requests, aiming to minimize response time.

B. The response time of accepted requests

We examine the response time for robot requests that have been accepted and ditpached by the middleware for processing. These requests are accepted only if the middleware identifies an application instance capable of meeting the required response time or privacy constraints; otherwise, the system rejects them.

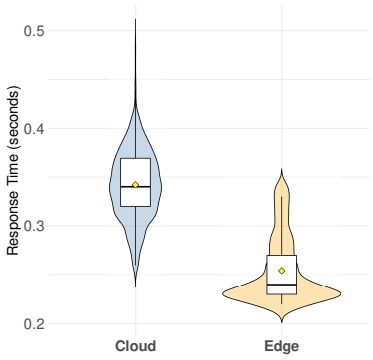

Fig. 7. Distribution of response times for accepted requests. Yellow: requests dispatched to the edge cluster; Blue: requests dispatched to the cloud cluster. The overlaid boxplot illustrates the quartiles, median, and interquartile range.

Figure 7 indicates that for requests routed to the Edge cluster, the middleware consistently maintains response times below the maximum threshold of 0.35 seconds. Similarly, for requests processed by the Cloud cluster, the average response time remains within the defined threshold of 0.5 seconds.

C. The task dispatching outcomes

We tracked the robot requests to analyze the percentage of requests being scheduled and where they were served – whether on edge or cloud servers. Figure 8 presents a bar chart that segments the total requests into three categories: those served on the Cloud, served on the Edge, and those that failed to be scheduled. As observed, the scheduler prioritizes routing requests with privacy requirements to the edge cluster, as it meets the defined privacy criteria, resulting in 84% of such requests being served on the Edge cluster. However, due to the strict requirement to serve on the Edge and the limited resources available at the edge server, approximately 16% of privacy-constrained requests failed to be scheduled.

In contrast, for normal requests, the absence of location constraints results in a higher success rate compared to privacy-constrained requests. Specifically, 98% of normal requests were successfully scheduled, with 7% served on the Edge and 91% served on the Cloud.

In the experiment, the reactive auto-scaling strategy based solely on CPU utilization led to some rejected requests when resources were insufficient at the time of arrival to meet their SLOs. Using the tinyKube middleware, robotic service providers can test deployments and adjust resource provisioning strategies to reduce early failures or rejections resulting from limited resource capacity.

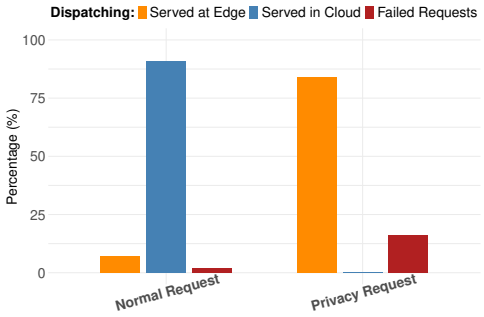

Fig. 8. Dispatching results for two types of robot requests: Privacy requests (with privacy guarantee) and Normal requests (without privacy guarantee).

VI. Conclusion and Future Work

With advancements in ubiquitous networking infrastructure and cloud computing, the realization of cloud robotics has become feasible, allowing robots to offload heavy computations to nearby edge nodes or remote cloud servers. However, to fully maximize these benefits, significant challenges remain in ensuring adequate resource allocation and in efficiently scheduling tasks across the distributed platform to meet the performance demands of robotics applications while optimizing resource utilization.

In this paper, we present tinyKube, a lightweight middleware tailored for the seamless orchestration of resources within distributed cloud-edge infrastructures for large-scale cloud robotics deployments. By leveraging the resource orchestration capabilities of the Kubernetes ecosystem and the Prometheus monitoring system, the middleware provides a comprehensive toolset for continuous system monitoring, request dispatching, and auto-scaling of resources across the cloud-edge platform to meet the demands of robotic applications.

The core contribution of tinyKube is its ability to simplify cloud-native computing and offer an easy-to-use tool for robotic application developers, accelerating the testing and deployment of large-scale robotics. Additionally, it provides a flexible template for quickly evaluating the effectiveness of algorithms and methodologies for managing resource allocation across cloud-edge infrastructures. Experimental evaluations of the middleware using the CloudGripper cloud robotics testbed – where computation-intensive components were deployed across a cloud-edge infrastructure – indicate that tinyKube effectively auto-scales resources according to defined strategies.

Currently, the middleware prototype is based on a centralized architecture. In the future, we plan to evolve beyond this single-entry model by exploring decentralized or distributed architectures to enhance scalability and fault tolerance. Furthermore, we aim to integrate the middleware into a larger ROS-based application deployment platform (e.g., FogROS2) for comprehensive evaluation through more diverse real-world scenarios.

ACKNOWLEDGMENT

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. The computations were in part enabled by the supercomputing resource Berzelius provided by National Supercomputer Centre at Linköping University and the Knut and Alice Wallenberg Foundation, Sweden.

REFERENCES

[1] G. Hu, W. P. Tay, and Y. Wen, "Cloud robotics: architecture, challenges and applications," *IEEE network*, vol. 26, no. 3, pp. 21–28, 2012.

[2] M. Armbrust, "Above the clouds: A berkeley view of cloud computing," 2009.

[3] E. Elmroth, P. Leitner, S. Schulte, and S. Venugopal, "Connecting fog and cloud computing," *IEEE Cloud Computing*, vol. 4, no. 2, pp. 22–25, 2017.

[4] M. Satyanarayanan, G. Klas, M. Silva, and S. Mangiante, "The seminal role of edge-native applications," in *2019 IEEE International Conference on Edge Computing (EDGE)*, pp. 33–40, IEEE, 2019.

[5] J. Kuffner, "Cloud-enabled humanoid robotics." http://www.scribd.com/doc/47896204/James-Kuffner-Humanoids2010, 2010. Accessed: 2023-09-30.

[6] B. Kehoe, S. Patil, P. Abbeel, and K. Goldberg, "A survey of research on cloud robotics and automation," *IEEE Transactions on automation science and engineering*, vol. 12, no. 2, pp. 398–409, 2015.

[7] A. A. RoboMaker. https://aws.amazon.com/robomaker/. Accessed: 2023-09-30.

[8] N. Omniverse. https://developer.nvidia.com/nvidia-omniverse-platform. Accessed: 2023-09-30.

[9] E. P. Roboearth. http://roboearth.ethz.ch/. Accessed: 2023-09-30.

[10] N. Tian, M. Matl, J. Mahler, Y. X. Zhou, S. Staszak, C. Correa, S. Zheng, Q. Li, R. Zhang, and K. Goldberg, "A cloud robot system using the dexterity network and berkeley robotics and automation as a service (brass)," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1615–1622, IEEE, 2017.

[11] J. Mahler, F. T. Pokorny, B. Hou, M. Roderick, M. Laskey, M. Aubry, K. Kohlhoff, T. Kröger, J. Kuffner, and K. Goldberg, "Dex-net 1.0: A cloud-based network of 3d objects for robust grasp planning using a multi-armed bandit model with correlated rewards," in *2016 IEEE international conference on robotics and automation (ICRA)*, pp. 1957–1964, IEEE, 2016.

[12] G. Mohanarajah, D. Hunziker, R. D'Andrea, and M. Waibel, "Rapyuta: A cloud robotics platform," *IEEE Transactions on Automation Science and Engineering*, vol. 12, no. 2, pp. 481–493, 2014.

[13] K. E. Chen, Y. Liang, N. Jha, J. Ichnowski, M. Danielczuk, J. Gonzalez, J. Kubiatowicz, and K. Goldberg, "Fogros: An adaptive framework for automating fog robotics deployment," in *2021 IEEE 17th International Conference on Automation Science and Engineering (CASE)*, pp. 2035–2042, IEEE, 2021.

[14] J. Ichnowski, K. Chen, K. Dharmarajan, S. Adebola, M. Danielczuk, V. Mayoral-Vilches, H. Zhan, D. Xu, R. Ghassemi, J. Kubiatowicz, *et al.*, "Fogros 2: An adaptive and extensible platform for cloud and fog robotics using ros 2," in *Proceedings IEEE International Conference on Robotics and Automation*, 2023.

[15] J. Ahmed Abdulsaheb and D. Jasim Kadhim, "Real-time slam mobile robot and navigation based on cloud-based implementation," *Journal of Robotics*, vol. 2023, no. 1, p. 9967236, 2023.

[16] R. Anand, J. Ichnowski, C. Wu, J. M. Hellerstein, J. E. Gonzalez, and K. Goldberg, "Serverless multi-query motion planning for fog robotics," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 7457–7463, IEEE, 2021.

[17] J. Ichnowski, J. Prins, and R. Alterovitz, "Cloud-based motion plan computation for power-constrained robots," in *Algorithmic Foundations of Robotics XII: Proceedings of the Twelfth Workshop on the Algorithmic Foundations of Robotics*, pp. 96–111, Springer, 2020.

[18] J. Mahler, B. Hou, S. Niyaz, F. T. Pokorny, R. Chandra, and K. Goldberg, "Privacy-preserving grasp planning in the cloud," in *2016 IEEE International Conference on Automation Science and Engineering (CASE)*, pp. 468–475, IEEE, 2016.

[19] J. Ichnowski, J. Prins, and R. Alterovitz, "The economic case for cloud-based computation for robot motion planning," in *Robotics Research: The 18th International Symposium ISRR*, pp. 59–65, Springer, 2020.

[20] Y. Ren, G. Liu, V. Nitu, W. Shao, R. Kennedy, G. Parmer, T. Wood, and A. Tchana, "{Fine-Grained} isolation for scalable, dynamic, multi-tenant edge clouds," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 927–942, 2020.

[21] K. Chen, N. Tian, C. Juette, T. Qiu, L. Ren, J. Kubiatowicz, and K. Goldberg, "Fogros2-plr: Probabilistic latency-reliability for cloud robotics," *arXiv preprint arXiv:2410.05562*, 2024.

[22] K. Chen, K. Hari, R. Khare, C. Le, T. Chung, J. Drake, J. Ichnowski, J. Kubiatowicz, and K. Goldberg, "Fogros2-config: A toolkit for choosing server configurations for cloud robotics," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 12083–12089, IEEE, 2024.

[23] K. Chen, M. Wang, M. Gualtieri, N. Tian, C. Juette, L. Ren, J. Ichnowski, J. Kubiatowicz, and K. Goldberg, "Fogros2-ls: A location-independent fog robotics framework for latency sensitive ros2 applications," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 10581–10587, IEEE, 2024.

[24] G. Toffetti, L. Militano, R. Tharaka, and M. Straub, "Ros-based robotic applications orchestration in the compute continuum: Challenges and approaches," in *Proceedings of the IEEE/ACM 16th International Conference on Utility and Cloud Computing*, pp. 1–6, 2023.

[25] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, *et al.*, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, p. 5, Kobe, Japan, 2009.

[26] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science robotics*, vol. 7, no. 66, p. eabm6074, 2022.

[27] K. Nishimura, T. Ishikawa, H. Sasaki, and S. Kato, "Raplet: Demystifying publish/subscribe latency for ros applications," in *2021 IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 41–50, IEEE, 2021.

[28] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ros2," in *Proceedings of the 13th international conference on embedded software*, pp. 1–10, 2016.

[29] Y. Liu, Y. Guan, X. Li, R. Wang, and J. Zhang, "Formal analysis and verification of dds in ros2," in *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, pp. 1–5, IEEE, 2018.

[30] U. R. Irfan, M. Paul, and T. D. Shashidhara. https://kubernetes.io/blog/2018/12/12/kubernetes-federation-evolution/. Accessed: 2023-09-30.

[31] Red Hat OpenShift. https://www.redhat.com/en/technologies/cloud-computing/openshift. Accessed: 2024-09-30.

[32] M. Zahid and F. T. Pokorny, "Cloudgripper: An open source cloud robotics testbed for robotic manipulation research, benchmarking and data collection at scale," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2024.

[33] O. Saha and P. Dasgupta, "A comprehensive survey of recent trends in cloud robotics architectures and applications," *Robotics*, vol. 7, no. 3, p. 47, 2018.

[34] J. Wan, S. Tang, H. Yan, D. Li, S. Wang, and A. V. Vasilakos, "Cloud robotics: Current status and open issues," *Ieee Access*, vol. 4, pp. 2797–2807, 2016.

[35] J. Zhang, F. Keramat, X. Yu, D. M. Hernández, J. P. Queralta, and T. Westerlund, "Distributed robotic systems in the edge-cloud continuum with ros 2: A review on novel architectures and technology readiness," in *2022 Seventh International Conference on Fog and Mobile Edge Computing (FMEC)*, pp. 1–8, IEEE, 2022.

[36] M. Afrin, J. Jin, A. Rahman, A. Rahman, J. Wan, and E. Hossain, "Resource allocation and service provisioning in multi-agent cloud robotics: A comprehensive survey," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 2, pp. 842–870, 2021.

[37] W. Wang, X. Zhu, L. Wang, Q. Qiu, and Q. Cao, "Ubiquitous robotic technology for smart manufacturing system," *Computational Intelligence and Neuroscience*, vol. 2016, no. 1, p. 6018686, 2016.

[38] M. Aissam, M. Benbrahim, and M. N. Kabbaj, "Cloud robotic: Opening a new road to the industry 4.0," *New Developments and Advances in Robot Control*, pp. 1–20, 2019.

[39] A. Manzi, L. Fiorini, R. Esposito, M. Bonaccorsi, I. Mannari, P. Dario, and F. Cavallo, "Design of a cloud robotic system to support senior citizens: The kubo experience," *Autonomous Robots*, vol. 41, pp. 699–709, 2017.

[40] T. Duckett, S. Pearson, S. Blackmore, B. Grieve, W.-H. Chen, G. Cielniak, J. Cleaversmith, J. Dai, S. Davis, C. Fox, *et al.*, "Agricultural robotics: the future of robotic agriculture," *arXiv preprint arXiv:1806.06762*, 2018.

[41] J. Gregory, J. Fink, E. Stump, J. Twigg, J. Rogers, D. Baran, N. Fung, and S. Young, "Application of multi-robot systems to disaster-relief scenarios with limited communication," in *Field and Service Robotics: Results of the 10th International Conference*, pp. 639–653, Springer, 2016.

[42] J. Lee, J. Wang, D. Crandall, S. Šabanović, and G. Fox, "Real-time, cloud-based object detection for unmanned aerial vehicles," in *2017 First IEEE International Conference on Robotic Computing (IRC)*, pp. 36–43, IEEE, 2017.

[43] Y. Zhang, C. Wurll, and B. Hein, "Kuberos: A unified platform for automated and scalable deployment of ros2-based multi-robot applications," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 9097–9103, IEEE, 2023.

[44] "Prometheus." https://prometheus.io/docs/introduction/overview/, 2024. Accessed: 2024-05-28.

[45] Thanos. https://thanos.io/tip/thanos/getting-started.md/. Accessed: 2024-09-30.

[46] M. Masse, *REST API design rulebook*. " O'Reilly Media, Inc.", 2011.

[47] T. Phung-Duc, "Exact solutions for m/m/c/setup queues," *Telecommunication Systems*, vol. 64, pp. 309–324, 2017.

[48] A. Gandhi, S. Doroudi, M. Harchol-Balter, and A. Scheller-Wolf, "Exact analysis of the m/m/k/setup class of markov chains via recursive renewal reward," in *Proceedings of the ACM SIGMETRICS/international conference on Measurement and modeling of computer systems*, pp. 153–166, 2013.

[49] M. Tan, R. Pang, and Q. V. Le, "Efficientdet: Scalable and efficient object detection," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 10781–10790, 2020.

[50] O. Adiyatov and H. A. Varol, "Rapidly-exploring random tree based memory efficient motion planning," in *2013 IEEE international conference on mechatronics and automation*, pp. 354–359, IEEE, 2013.

[51] M. Welsh and D. Culler, "Adaptive overload control for busy internet servers," in *4th USENIX Symposium on Internet Technologies and Systems (USITS 03)*, 2003.