ABSTRACT

LU, ZIJUN. Microservice Performance Profiling and Uncertainty Quantification: A Case Study on Intelligent Surveillance as a Service. (Under the direction of Dr. Ruozhou Yu).

In recent years, the microservice architecture has become increasingly popular for its scalability, agility, flexibility, and resilience. Administrators often use Docker to containerize microservice applications for simplified deployment and rely on Kubernetes for their management and scaling. However, Kubernetes poses challenges in performance tuning due to its limited observability features, which are important for effective profiling and resource configuration analysis. This necessitates additional tools or strategies to improve performance in Kubernetes environments.

In this thesis, we conducted a comprehensive profiling analysis of an intelligent surveillance system, incorporating a dedicated microservice for object detection. Although existing work has addressed the modeling and prediction of resource-performance trade-offs in video processing tasks, this study aims to answer two questions: (1) Can machine learning models serve as good digital twins for predicting the performance of video processing tasks in microservice environment? (2) How can we make predictions of the twin model to be robust against uncertainty in the measurement data? To answer these questions, we had implemented an end-to-end framework for profiling an intelligent surveillance application. Utilizing Kubernetes and Jaeger distributed tracing tool, we performed extensive profiling of the application under 201 different resource and configuration setups, and collected over 369000 end-to-end microservice execution time data points. Based on the measurement data, we performed feature analysis, and trained and evaluated various machine learning models for predicting the microservice performance given different resource configurations and setups. From our study, we made several key observations: (1) Microservice performance modeling exhibits non-negligible uncertainty that can lead to severe performance violation if not accounted for, especially in congested scenarios when resources are limited. (2) Conformalized Quantile Regression and Probabilistic Neural Networks are effective for prediction and uncertainty quantification given minimal surveillance system features. (3) Although Probabilistic Neural Networks do not require training separate models for different quantiles, they generate more conservative uncertainty estimations compared to Conformalized Quantile Regression. At the conclusion of the thesis, we discussed future directions of research based on case study presented in this thesis.

Microservice Performance Profiling and Uncertainty Quantification: A Case Study on Intelligent Surveillance as a Service

by
Zijun Lu

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina
2024

APPROVED BY:

Dr. Xiaorui Liu	Dr. Yuchen Liu

Dr. Ruozhou Yu
Chair of Advisory Committee

BIOGRAPHY

Zijun Lu was born in Foshan, China, on March 25th, 1993. He moved to Raleigh to attend North Carolina State University in 2012 and earned his B.S. in Electrical and Computer Engineering in 2018. During his college years, he cooped as a Radio Frequency Engineer intern at Longent, LLC (now acquired by Mobile Communications America) for four terms. He was an undergraduate researcher under the supervision of Dr. Ismail Guvenc, working on ray tracing simulations during the summer of 2017. Having received three prestigious scholarships from generous NCSU emeriti and alumni donors, he was eager to give back to the community and has made multiple donations to the NC State College of Engineering in the past years. After his graduation, Zijun joined American Tower Corporation, holding roles such as Network Architecture Engineer and Radio Frequency Engineer.

In 2021, Zijun returned to North Carolina State University to pursue a Master of Science in Computer Science. His graduate life was enriched by two software engineer internships at Credible and ENFOS, two graduate teaching assistantships, and one research assistantship under the supervision of Dr. Ruozhou Yu. Besides his academic achievements, he also holds certifications as a Certified Kubernetes Administrator, Certified AWS Cloud Practitioner, and Cisco Certified Network Associate.

Beyond his professional and academic pursuits, Zijun is passionate about basketball, playing regularly to maintain fitness. He deeply values the time spent with his family and friends, cherishing the moments that bring them together.

Raleigh is not just where Zijun goes to school or works; it has become his second home over the last 11 years. It is where he has achieved milestones, made lasting friendships, and truly grown as an engineer. His love for Raleigh says it all – it is more than just a city to him; it is a huge part of his life and has given him a real sense of belonging.

ACKNOWLEDGEMENTS

I want to express my profound gratitude to my advisor, Dr. Ruozhou Yu. Over the past year, I have learned a lot personally and academically from him. There is no doubt that he is very passionate about his research interest and always has great insights on advising research direction. Moreover, he shows remarkable responsibility as an advisor by dedicating long hours to providing prompt feedback on my thesis paper and presentation slides. I also want to thank my lab mate and mentor, Zhouyu Li, for his constant technical support with Kubernetes and environment setup. I believe he is more than ready to embrace his academic and professional success in the near future.

I would like to thank my esteemed committee members, Dr. Xiaorui Liu and Dr. Yuchen Liu. Thanks for their invaluable and insightful feedback on my thesis paper, which help me improve it significantly.

I am grateful for the close companionship and support of my dear friend, Dr. Yunqing Li, who has been more than I could have asked for during my three-year graduate journey. Last but not least, I am forever indebted to my family, especially my father, Lihua Lu. I regret that at the age of 30, I still require your financial support, but your unconditional love is deeply appreciated and gives me tremendous strength to continue pursuing my American dream.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1

INTRODUCTION

This chapter discusses the background and motivation of the thesis, contributions to the field of surveillance intelligent systems, and thesis organization.

1.1 Background and Motivation

1.1.1 Microservice Archiecture

A microservice architecture is a distributed application framework, comprising lightweight and autonomous modules termed as microservices, each dedicated to a distinct function of the application. These microservices collaborate via messaging to fulfill the application's collective objective (17). This configuration facilitates modular development, thereby streamlining the software development process.

Recently, the microservice architecture has gained popularity in the service-oriented software industry. The industry is projected to expand from $5.49 billion in 2022 to $21.61 billion dollars by 2030, a growth attributed to its scalability, agility, flexibility, and resilience (49). First, microservice architecture permits the selective scaling of modules experiencing high demand, in contrast to monolithic architectures which necessitate replicating the entire application for scaling, thereby optimizing resource utilization (17). Second, microservice architecture

facilitates agile methodologies through incremental development and deployment, permitting updates or replacements of individual services without impacting adjacent components (7). Third, it enables technological heterogeneity for developers since they are not restricted to using specific product stacks or programming languages (12; 54). Finally, microservice architecture is designed for fault tolerance, maintaining service availability through redundant, lightweight microservices, even in the event of individual service failures (17).

1.1.2 Observability of Microservices

Observability is a critical part of managing microservices because of the interconnectivity of numerous microservices. With observability, developers can optimize and debug their application through logs, distributed traces, and metrics data. Logs help one understand information on application events, while distributed traces and metrics assist in understanding the client request pathway and evaluating application performance (19; 42).

In the software industry, distributed tracing is often employed in the microservice architectures as it allows tracking requests across the whole system (34). The main task of Distributed tracing is to record events and their respective timestamps across a program's execution. Every event is instructed to capture a unit of work of the service. This most basic unit of work is often referred to "span" or "operation", and a "trace" can be considered as a collection of "span" connected in a parent/child relationship (19; 52).

The typical tracing and analysis pipeline is described as the following steps (34).

1. Each microservice is paired with a logger, which generates spans for each service call. These spans contain information such as the operation/span customized name, trace ID, span ID, logs, tags, timestamp, and duration, though the exact terminology might vary based on different vendors.

2. A centralized collector gathers span logs from the target microservice instances.

3. The raw trace data needs to be preprocessed before trace analysis, which includes metrics aggregation and log formatting.

4. The preprocessed data needs to be stored for further analysis.

5. Interested parties can analyze trace data to find the root cause and identify performance bottlenecks as examples.

1.1.3 Docker

Docker is a software that uses container virtualization technology to simplify and improve application deployment. Before containerization, virtualization is primarily implemented with virtual machines, which takes up relatively large amount of compute resource. The scaling

with such virtualization results in duplicating operating system running on top of a hypervisor running on top of physical hardware, which your program is then on top of. To address the challenges for speed, performance and development agility, Containerization have a hypervisor placed on top of the operating system thus avoid operation overhead. Docker, one of the most popular containerization software, holds each microservice and its necessary libraries and dependencies, allowing them to run independently in any environment (5). In today's practices, developers commonly use Docker to package their applications locally or upload to Docker Hub, and then use Kubernetes to deploy and mange those containers in a production environment.

1.1.4 Kubernetes and Its Challenges

Kubernetes is an open-source platform developed by Google for orchestrating containerized applications. It provides solutions for automating the deployment, scaling, and management of applications, making it well-suited for microservice architectures (28).

```
controlplane ~ ➜  kubectl get pods -n kube-system
NAME                                   READY   STATUS    RESTARTS      AGE
coredns-69f9c977-swtd2                 1/1     Running   0             25m
coredns-69f9c977-z6c7x                 1/1     Running   0             25m
etcd-controlplane                      1/1     Running   0             25m
kube-apiserver-controlplane            1/1     Running   0             25m
kube-controller-manager-controlplane   1/1     Running   0             25m
kube-proxy-472mp                       1/1     Running   0             24m
kube-proxy-sgsjq                       1/1     Running   0             25m
kube-scheduler-controlplane            1/1     Running   0             25m
metrics-server-98bc7f888-cff59         1/1     Running   0             37s
weave-net-872fw                        2/2     Running   1 (25m ago)   25m
weave-net-qv76c                        2/2     Running   0             24m

controlplane ~ ➜  kubectl top node
NAME           CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
controlplane   124m         0%     834Mi           0%
node01         25m          0%     367Mi           0%
```

Figure 1.1: Example of Kubernetes Metrics API

However, Kubernetes encounters two challenges in performance tuning. First, Kubernetes does not offer detailed monitoring for microservices, especially when it comes to tracking how specific lines of code use resources and how long they take to run (34). It allows you to see overall resource use for groups of containers (pods) or entire machines (nodes) in your Kubernetes setup. However, this information does not give a clear picture of how resources and performance are related, as shown in Figure 1.1. Second, despite Kubernetes not enforcing resource allocation specifications prior to deployment, such allocation is highly recommended.

In Kubernetes, resources can be allocated through command line or usually specified in Yet Another Markup Language (YAML) file. In the absence of these specifications, there is a risk of containers consuming all available CPU and memory resources, potentially resulting in resource contention and adversely affecting the stability and performance of other microservices within the same cluster. Consequently, it is important for those managing Kubernetes to estimate and set aside the resources each job will need to keep everything running smoothly (11).

Optimizing resource allocation and performance within Kubernetes presents considerable challenges, necessitating substantial expertise. Such challenges often lead to suboptimal utilization of computing resources, manifesting as either over-provisioning or under-provisioning. Administrators often struggle to understand how various resource configurations impact performance without proper observability, including identifying potential bottlenecks and estimating uncertainty in workload demands. For instance, without in-depth observability and supportive measurement data, it is difficult to determine whether allocating an additional CPU would lead to a significant improvement in the response times of microservice. Similarly, critical tasks demand reliable services to ensure prompt response times, such as surveillance alarms.

1.1.5 Uncertainty Quantification

Uncertainty refers to the level of doubt or variability in the outcome of the prediction models. It can be caused by different sources, such as limited data, noisy data, and incorrect model assumptions. Further, uncertainty can be categorized into two types: aleatoric uncertainty and epistemic uncertainty. Aleatoric refers to the data's inherent randomness, which can not be reduced by more data or better models. On the other hand, epistemic is a reflection of limited knowledge or confidence of a model, which can be mitigated by more data and better models (22; 44).

Uncertainty quantification is important to be consider for several reasons. First, more and more safety-critical applications demand such trustworthy representation, such as medicine or socio-technical systems (22; 31; 58; 60). Second, it can help ones evaluate the accuracy and confidence of their model predictions. Third, it can assist in identifying areas where the measurement data is lacking and provide guidance on collecting more informative data (1).

Over the past several years, researchers have developed different methods to estimate uncertainties. They include but not limited to quantile regression, data perturbation with different resampling techniques, and Bayesian inference (2). There is also existing work on system profiling and performance prediction, but most of it either fails to address uncertainty or the applications measured are not based on microservices (8; 10; 13; 36).

1.1.6 Case Study: Intelligent Surveillance System

Video Surveillance Systems are designed to monitor and record activities in various environments to boost safety and security, evidence documentation, and streamlined operations (18). In recent years, due to the high demand for security and safety and advancements in computer vision, intelligent surveillance systems have received increasing attention. These systems are designed to deal with real-time monitoring, and have intelligent features to automatically analyze surveillance data, and take proper actions for anomalies. Some of the popular applications for an intelligent video surveillance system are object detection, recognition, tracking, behavioral analysis, and retrieval (23; 57).

Until recently, researchers have proposed an intelligent surveillance system for smart cities which laid a good foundation by integrating microservice architecture in such domain (3). This thesis aims to extend their concepts further to profile the resource-to-performance mapping, using machine learning models to forecast performance, and quantifying uncertainty for the surveillance image processing service based on our measurement data. This endeavor seeks to enhance the understanding and optimization of microservice architectures in a smart surveillance context.

1.2 Contributions

This research makes contributions to the field of surveillance intelligent systems and their applications, which can be summarized as follows:

- We implement an emulated microservice-based surveillance intelligent system with an observability feature to obtain profiling data of the target application under different resource and configuration setups.
- We train machine learning models to learn the resource-performance correlations of the target application, aiming to build a digital twin of its performance model, and integrate uncertainty quantification into the models for robust predictions.
- Through evaluation with the measurement data, we demonstrate the effectiveness of both Conformalized Quantile Regression and Probabilistic Networks accurately predicting the performance and estimating uncertainty with two surveillance features.

1.3 Thesis Organization

The organization of the remaining thesis is as follows: Chapter 2 reviews the related work. Chapter 3 offers a comprehensive overview of the research methodology. Chapter 4 delves into the specifics of the implementation and evaluation. Chapter 5 extends discussion on additional profiling features and resource configuration recommendation. Finally, Chapter 6 concludes the thesis and presents directions for future research.

CHAPTER

2

RELATED WORK

This chapter reviews recent research with a focus on resource management, microservices, and Kubernetes, as well as developments in surveillance systems (3). It analyzes and traces microservices to evaluate performance and talks about the differences between profiling and benchmarking. Additionally described are the workings of REST APIs, the introduction of uncertainty quantification more especially, and the theories underlying Model Agnostic Prediction Interval Estimator (MAPIE) and Probabilistic Neural Networks (PNNs)-aims to capture the essence of the present scholarly conversation around these technologies.

2.1 Advancements in Video Surveillance Systems

The development of video surveillance systems over the last few decades has led to a significant expansion in the field of audio-visual data across various industries (18). At the same time, the world's population increased dramatically, and terrorist attacks at the start of the 21st century also increased. These elements contributed to the widespread deployment of security cameras in cities (43). Millions of cameras are currently used for widespread urban surveillance, which uses a variety of surveillance techniques to support social control and security (16). Collecting, recording, and analyzing surveillance data has become prevalent with the development of the Internet. Using video monitoring safely is becoming more and more in demand. Network

infrastructure, computers, and sensors are essential parts of video surveillance systems. These systems' analytical abilities, powered by complex processing algorithms, are essential to the success of surveillance operations. In this sense, the analytics act as the system's "brain", and the hardware as its "eyes".

2.2 Current Work in Microservices

While studies in software engineering have concentrated on the transition from monolithic architectures to microservices (14) and explored other aspects of software engineering (26), such as software attributes like coupling and cohesion, comprehensive insights into how microservice developers effectively manage resource are absent.

As distributed tracing and analysis in microservice systems attract increasing interest among scholars, questions persist regarding the sufficiency of these approaches in attaining the requisite level of observability. The goal of observability is to obtain data and provide insight to better understand the internal system. Developers can obtain observability by logs, traces, and metrics data. Such data is critical for software development as it facilitates root cause debugging and identifies performance bottlenecks. There are many open-source tools available to use to obtain observability, such as Jaeger and Prometheus (42). Recent work presents a pattern-based method that maps interaction patterns to REST APIs and models them for effective microservices benchmarking (20)

2.3 Resource Management in Kubernetes

Kubernetes is an open-source container orchestration system originally developed by Google. Because of the simple interface and strong capabilities, Kubernetes has become a standard across a wide range of industrial sectors (25; 38). In Kubernetes, the smallest execution unit to encapsulate one or more applications (28). The capacity of Kubernetes to run numerous pods on a single host server makes performance isolation between these pods essential (59). Kubernetes uses a resource management system (50), which enables users to assign particular computer resources, like CPU, memory, and network bandwidth, to each pod. In order to guarantee that the allocated resources are preserved and, in theory, isolated from other pods on the same server, this allocation is carried out through a description file at the time of pod creation.

Kubernetes's resource management technique is based on Linux cgroups, which are fundamental Linux kernel features for managing process resources (21; 48). For instance, a user can

provide a fixed amount of network bandwidth to a pod through tc, as defined in its description file, or limit a pod to use only one CPU core by using the cgroups API. The foundation of this configuration is the belief that Kubernetes can preserve performance isolation between pods in accordance with these established resource distributions. Most existing research has focused on enhancing the efficiency of computing resource utilization within Kubernetes, which primarily concentrates on CPU and GPU performance. A recent work introduces a method to optimize Kubernetes resource management for deep learning tasks (32), which boosts container performance by automating CPU and GPU resource allocation.

There are many existing approaches to address resource allocation strategy without accounting for uncertainty, such as an offline approach to distributed tracing (19). It is understandable that defining uncertainty in resource allocation is difficult, particularly due to unpredictable workload (46). However, researchers recently proposed a performance-aware resource allocation for general objectives via online feedback (11).

2.4 Current Work in Uncertainty Quantification

Uncertainty Quantification (UQ) methods play an important role in reducing the impact of uncertainties in optimization and decision-making processes across a variety of fields. These methods are notable for their contributions to computer vision, autonomous driving, medical image analysis, and natural language processing, highlighting their broad applicability and significance in enhancing accuracy and reliability (4).

In addition, UQ techniques are not limited to classification and detection tasks but are also extensively used in regression problems, highlighting their comprehensive scope for enhancing model reliability and decision-making accuracy in a multitude of disciplines. In order to reduce the original high-dimensional problem to a low-dimensional one, Li et al. offer two inverse regression-based UQ algorithms (IRUQ) (35).

Quantifying uncertainties in machine learning model predictions is essential for reliable AI systems, requiring the need for comprehensive UQ libraries. To assure trustworthiness in AI applications, these libraries need to be open-source, flexible enough to work with a wide range of models and use cases, and able to provide strong theoretical guarantees on uncertainty estimations. In response to these needs, the MAPIE library was developed (51). MAPIE is an open-source Python library built in accordance with the scikit-learn framework and made available through scikit-learn-contrib. It requires only a scikit-learn API for compatibility and supports base estimators that are scikit-learn-compatible. A machine learning framework called conformal prediction (CP) is used to quantify uncertainty. It generates prediction ranges for any underlying point predictor, given that the data are exchangeable. MAPIE stands out for

its implementation of CP methods applicable to both regression and classification tasks. By using these CP techniques, MAPIE is able to provide mathematical assurances for the marginal coverage of uncertainty.

2.5 Problem Statement

Microservice architectures have emerged as a cornerstone of modern software engineering due to its scalability, agility, flexibility, and resilience. Kubernetes, one of the major tool implementing microservices, exhibits significant limitations in its built-in observability capabilities, particularly in profiling resource-performance data and accounting for uncertainty in the execution of critical tasks. Without such observability, system administrators and software developers face difficulty to configure resources to match performance and reliability of critical tasks. Meanwhile, enabling observability via distributed tracing typically involves highly non-trivial overhead on the application, and is not a viable option in performance-critical production environments. To circumvent this issue, a promising way is to build a performance model—a digital twin of the the application's resource-performance correlations—that can be used to predict the expected performance of an application for runtime resource configuration. Ideally, this model should be able to accurately predict the performance of a microservice application given a set of resource configurations. In practice, however, such models commonly suffer from uncertainty inherent in the data (such as workload and input data distributions) and the models themselves. If not accounted for, the uncertainty will hinder the effectiveness of resource management, leading to potential performance degradation or violation of application performance goals.

Given the above problem, we try to address by building a resource-performance model of a microservice application based on measurement data, utilizing machine learning tools, and incorporating uncertainty quantification to make robust performance predictions of the microservice.

CHAPTER

3

METHODOLOGY

In this chapter, we firstly propose a structured framework to profile the mapping of resources to performance under different resource and setup configurations. This is supported by a Microservice Resource-Performance Trace Collector, an Intelligent Surveillance System Model and a Survillance Task Processing Worker Node. Second, we perform Raw Data Extraction and Data Preprocessing to process experimental measurement data as the input data to train and evaluate our models. Lastly, we train our models and integrate Robust Prediction Interval Estimation for uncertainty quantification. We detail each step in the following subsections.

3.1 Intelligent Surveillance System Model

The primary purpose of the Intelligent Surveillance System Model (ISSM) is designed to replicate the operational load of an intelligent surveillance system. It produces workloads that match real-world conditions in data source, frame rate and concurrent device emulation.
Data Source: ISSM utilizes a set of stored images at varying resolutions—HD standard, Quad HD, and 4K UHD—to reproducibly emulate an intelligent surveillance system with heterogeneous cameras (18; 41; 56) as presented in Table 3.2. The emulation is based on the assumption that all client devices capture and transmit identical sets of images. Though this arrangement might not capture the diversity seen in actual surveillance setups, it is designed to normalize the data

for analytical purposes.
Frame Rate: ISSM emulates a video frame rate of 24 frames per second (fps), which is the most common video frame rate. By maintaining this rate across all emulated resolutions, ISSM provides a realistic emulation of the data streaming capabilities of actual surveillance cameras.
Concurrent Device Emulation: ISSM is designed to emulate the concurrent operation up to five clients using Raspberry Pi. The decision to profile the system across a range of 1 to 5 clients is reflecting small to medium-sized setups, such as those found in retail stores and home surveillance systems. By profiling such setups, we aim to gain valuable insights into its performance under various video stream workloads. However, recognizing the potential for larger-scale applications, our work could be extended to explore ISSM's capability to accommodate an increased number of clients in the future.

3.2 Microservice Resource-Performance Trace Collector

Our framework aims to achieve comprehensive observability of the target intelligent video surveillance application. As discussed, Kubernetes is adept at managing containerized applications but does not offer built-in observability tools for monitoring at the code block level. To achieve such observability, we leverage external tools such as Jaeger for distributed tracing, Kafka for real-time streaming pipelines, and Cassandra for data storage (9; 27; 29).
Jaeger: In order to collect tracing data with Jaeger, we follow the following steps to configure the data collection interface (52).

1. We configure a Jaeger agent, which is a network deamon actively collects spans from the target service over UDP on port 6831. It forwards collected tracing data in batch to the collector.

2. We configure a Jaeger collector, which is responsible to receive traces from the Jaeger agent. The Jaeger collector also performances data validations and transformations upon receipt. It then forwards the tracing data to the data backend, Kafka and Cassandra, which we will discuss later in this section.

3. We configure a Jaeger ingestor because it is required to work with Kafa. It is bascially a intermediary service which read from Kafka and write to Cassandra.

4. In order to send tracing data, we also need to instrument our application as shown in sample instrument in Figure 3.1. In this code snippet, we begin with marking a tracing span with a unique name to indicate the start of the modeling and tagging process. Then, we perform image process model inference while measuring time duration of model execution times and collecting system metrics. Finally, we mark another tracing span to indicate the completion of the modeling and tagging process. The purpose of these marking steps is to provide us to

measure the additional tagging time separately from the model inference time.

```
detect_server_span.add_event("Start modeling and tagging for " + uniqueId)
current_process = psutil.Process(os.getpid())
cpu_percent_sys_dummy = psutil.cpu_percent(interval=None)
cpu_percent_ps_dummy = current_process.cpu_percent(interval=None)
start_model = time.time()
results = model(frame)
end_model = time.time()
cpu_percent_ps = current_process.cpu_percent(interval=None)
cpu_percent_sys = psutil.cpu_percent(interval=None)
util_info = get_util_info(current_process)
detect_server_span = note_util_info_in_span(util_info, detect_server_span)
detect_server_span.set_attribute("cpu_percent_ps", cpu_percent_ps)
detect_server_span.set_attribute("cpu_percent_sys", cpu_percent_sys)
model_time = end_model - start_model
detect_server_span.set_attribute("model_time", model_time)
image_size_kb = len(jpg_original) / 1024
detect_server_span.set_attribute("image_size_kb", image_size_kb)
detect_server_span.add_event("End modeling and tagging for " + uniqueId)
```

Figure 3.1: Code Snippet of Jaeger Instrumentation

Kafka: The emulated surveillance system demonstrates significant concurrency, driven by its high frame rate of 24 frames per second and the capacity to handle up to five clients concurrently. The default configuration of Jaeger clients has max queue size of 100, which can only buffer this amount of spans before sending them to the collector. However, our investigated system generates spans at a maximum rate of approximately 190 per second. To address this bottleneck, our architecture incorporates Apache Kafka to enhance the resilience and scalability of our tracing data collection. Kafka, a distributed streaming platform, excels in handling large volumes of data with minimal latency (27). It serves as a reliable and scalable tool for managing the data flow, ensuring no data loss occurs when Jaeger client buffers reach their limits. In a nutshell, Jaeger collector acts as a Kafka producer, which can publish tracing data as messages to a Kafka topic. The Kafka topic is a steam of messages of a specific type. These messages are subsequently saved to a set of configured servers called brokers. And the Jaeger Ingester as the Kafka consumer can retrieve the message from the brokers.

Cassandra: Cassandra is an open-source, distributed, column-oriented database that offers highly available storage service for vast amount of structured data (29). We utilize Cassandra for storing tracing data generated by Jaeger.

3.3 Surveillance Task Processing Worker Node

The Surveillance Task Processing Worker Node is designed to process AI-driven tasks. In our emulation, we choose object detection as a computationlly intensive video analytics task as a representative example, implemented using the mature YOLOv5 algorithm.

YOLOv5: YOLOv5 is a state of the art deep learning model used for object detection tasks. The scope of our application is to send images to the surveillance task processing workload, then the image processing service identifies objects within an image and drawing bounding boxes around them. We use a pre-trained YOLOv5 model from torch.hub.

Within the microservice-based surveillance system, the object detection application is managed by Kubernetes as a worker node within the Kubernetes cluster. For each deployment of the application, we pre-define the resource requirements to specify resource utilization as discussed in Section 3.5. These specifications are instructed within the Kubernetes YAML configuration files, ensuring a consistent and controlled environment for all our experiments.

With the defined resource available to the microservice, it is exposed through an HTTP endpoint within the Kubernetes cluster, serving as the interface for incoming object detection requests. Upon receiving a request, the application initiates the image processing pipeline along with Jaeger tracing, as detailed in the above.

3.4 End-to-end workflow

As discussed and shown in the Figure 3.2, our system architecture consists of the ISSM, the surveillance task processing worker node, and microservice resource-performance trace collector. The ISSM employs five Raspberry Pis as surveillance clients, with each one processing three batches of 200 images. These images are extracted from an 8.4-second clip of BDD100K video and are processed at three resolutions: Standard HD (1280 x 720 pixels), Quad HD (2560 x 1440 pixels), and 4K UHD (3840 x 2160 pixels), all at a frame rate of 24 fps. The process pipeline for each image is uniform: starting with its conversion into a pixel array, proceeding with JPEG encoding, and ending with the transmission of the resulting byte sequence to the worker node via HTTP.

Central to our server-side operations is the object detection service, deployed as a Flask application on the worker node. This service actively listens for image processing requests on the /detectframe endpoint. Upon receiving an incoming request carrying an image byte sequence in its request body, the worker node proceeds to convert the byte sequence into a NumPy array, then decodes it into BGR (Blue, Green, Red) color format for model inference. The application then applies the image processing model to the image frame for detection.

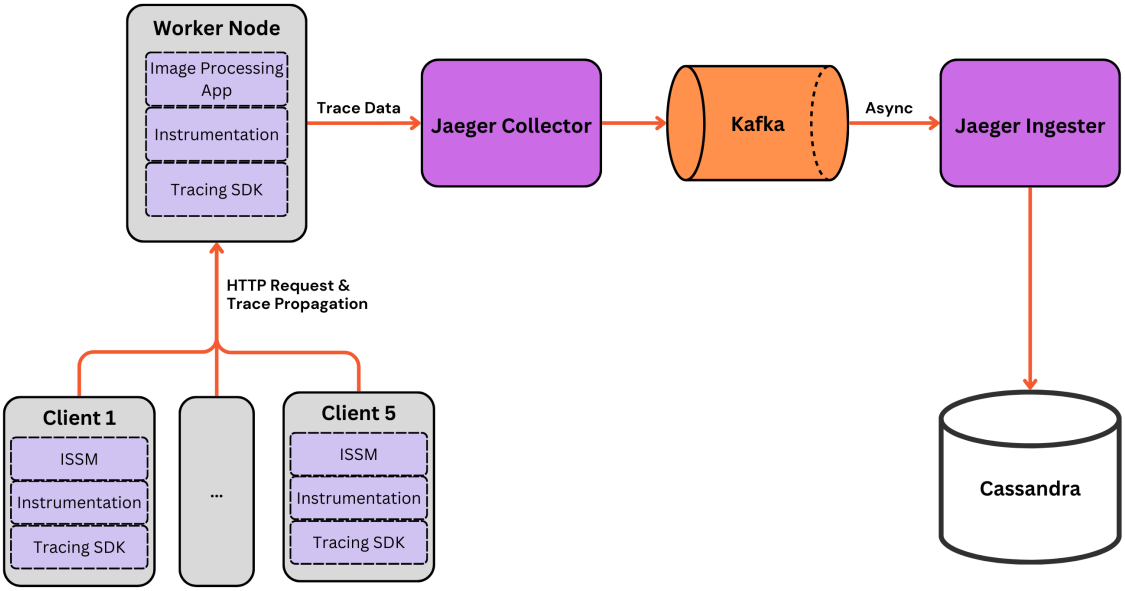

Figure 3.2: Architecture and Flow

Finally, it sends back the object detection results to the respective clients.

Working alongside with image processing workflow, we instrument the observabilty workflow using the microservice resource-performance trace collector. Firstly, We employ Jaeger tracing with essential preparatory tagging activity before the requests are dispatched to the worker node. This preparatory step is crucial as it allows us to stamp each request with specific metadata, enabling the traceability of source ownership throughout the processing stages. Once the trace reaches the worker node, we focus on deriving the execution time of the image processing model inference. We instrument this phase to log the start and end times of the inference task so that we can calculate the total duration of model execution in the raw data extraction procedure.

During the distributed tracing process, the Jaeger agent actively collects tracing spans in batches and forwards them to the Jaeger collector. Then the tracing data is serialized and stored in a Kafka topic as messages. After that, the Jaeger ingester deserializes the messages and writes them persistently to the Cassandra storage backend. The final saved Jaeger trace output data is presented in Table 3.1

Table 3.1: Surveillance Image Processing Task Trace Context Model

Field Name	Type	Purpose
Operation	String	Customized span name
Service	String	Notation for the service name indicating whether a span is captured on the client or server side
Duration	Long	Span duration
Logs	Span Events	Events timestamped on the span to measure code execution duration
Tags	Attributes	Logging process and system metrics
SpanID	Long	Span identifier
TraceID	String	Trace identifier

3.5 Resource and Configuration Setups

The resource allocation for the image processing application is configured as follows. The CPU resources start at 1000 milliCPUs and increase in steps of 500 milliCPUs, reaching a maximum of 7000 milliCPUs. Memory is consistently set at 8 gibibytes. An illustrative example of the resource configuration with 1000 milliCPUs and 8 gibibytes memory is presented in Figure 3.3. In Kubernetes, setting 'requests' ensures a minimum level of resources available for the application, and defining 'limits' prevents the application from using more resources than a specified maximum.

```
spec:
  containers:
    - name: pyserver-detector
      image: zlu5/detector:26.0
      imagePullPolicy: Always
      resources:
        requests:
          cpu: "1000m"
          memory: "8Gi"
        limits:
          cpu: "1000m"
          memory: "8Gi"
```

Figure 3.3: Resource Configuration Specification

In each experiment, surveillance client participants use one of the three stored image sets at three resolutions (standard HD, Quad HD, and 4K UHD), that is totally 200 images to process per client per experiment. The number of surveillance client participants is set from 1 to 5. In each experimental setup, we change one variable at a time—either the number of clients, the image resolution, or the resource allocation (CPU and GPU usage)—and then replicate the same experiment three times under the same conditions. This grid-based iterative approach ensures that each variable is modified independently, aiming to gather 600 resource-mapping data points per client for each tested resource and configuration setup. The data features investigated and label with our experiment, including their corresponding units, are detailed in Table 3.2. As the final result of the data collection, we gather 369,000 data points garnered from 201 different resource and configuration setups.

Table 3.2: Description of Primary Features and Label

Label/Feature	Description	Unit
model_time	Time taken for the model to run	second
image_size	Resolution and typical approximate file size of the image (Standard HD: 300KB, Quad HD: 680KB, 4K UHD: 1200KB)	kilobytes
num_of_client	Number of clients	count
num_of_cpu	Number of CPU cores allocated	milliCPU (m)

3.6 Raw Data Extraction and Data Preprocessing

As discussed above, we need to stamp each request with metadata so that we can identify the source of the request for each tracing span in the step of the raw data extraction process. In this step, we need to group the collected tracing spans from a specific client request by a unique identifier which is a composite of the image file name, operation name and trace ID. After the grouping, we employ regular expression techniques to extract features from Jaeger's structured data format to form tabular data points (53).

Next, to ensure that all features contribute equally to the analysis, a normalization step is employed. We apply the MinMaxScaler to adjust their scale to range from 0 to 1. (45).

Finally, We utilize stratified sampling to partition the dataset from each file into training, validation, and testing sets with 80/10/10 distribution (39). This method ensures that each

subset represents the overall dataset, preserving the distribution of crucial variables.

3.7 Robust Prediction Interval Estimation

To estimate robust prediction intervals in regression analysis, we employ the MAPIE framework with XGBRegressor model (15). MAPIE, grounded in the principles of Conformal Prediction, Quantile Regression, and resampling techniques, provides a systematic approach for handling prediction uncertainties (24; 33; 47; 51). We evaluate different sampling methods used for uncertainty quantification in a regression context, such as cross-validation-plus (CV+) and Jackknife+-after-Bootstrap, which will be discussed in much detail in a later chapter. This exploration allows us to find the trade-offs between interval width and model performance between resampling methods.

As a comparative approach, we extend our uncertainty quantification with PNN (30). This comparative analysis aims to yield the best way to perform resource-performance predictions and uncertainty quantification in our investigation.

CHAPTER

4

IMPLEMENTATION AND EVALUATION

This chapter outlines the implementation to generate the profiling dataset for the evaluation of the YOLOv5 object detection application, as deployed within the intelligent surveillance system. The system's architecture encompasses a Kubernetes cluster, and a Surveillance Workload Emulation Module using Raspberry Pis. Additionally, this section entails the details of uncertainty quantification with MAPIE and PNN.

4.1 Implementation

Kubernetes Cluster Configuration: The Kubernetes cluster is structured with a master node for cluster management, a data node that hosts essential data-related services, and a worker node endowed to provide computational resources. Specifically, the worker node is equipped with a high-performance Intel® Xeon® Gold 5317 CPU, 256GB of memory, and an Nvidia A100 GPU. The worker node runs the YOLOv5 application across 201 resources and configuration setups to profile its performance as discussed in the Methodology section.

Surveillance Workload Emulation Module: This module uses five Raspberry Pis to act as surveillance clients, each processing three sets of 200 images derived from an 8.4-second clip of BDD100K video at resolutions of standard HD (1280 x 720 pixels), Quad HD (2560 x 1440 pixels), and 4K UHD (3840 x 2160 pixels). The surveillance client participants are Flask servers

actively listening for requests on the /detectFrame endpoint. Additionally, we have configured an initiator to concurrently send out a start command to the participants to ensure they all start simultaneously.

Data Node: The data node plays an important role in ensuring tracing data collection. It contains a Cassandra database for persistent data storage, a Jaeger collector for distributed tracing collection, and Kafka for reliable data buffering.

4.2 Uncertainty Quantificaiton with MAPIE

MAPIE supports multiple conformal prediction methods, such as Naive, CV+, Jackknife+-after-Bootstrap, and Conformalized Quantile Regression (CQR). For CQR, a quantile regressor is necessary in MAPIE. As such, we choose to use a LGBMRegressor as our estimator. We will briefly discuss them as follows.

LGBMRegressor: The LGBMRegressor is a part of LightGBM (Light Gradient Boosting Machine), which is a gradient boosting framework based on decision trees to enhance model efficiency and minimize memory consumption.

Naive: The naive method uses the residuals from the training data to estimate the error on a new test data point. The prediction interval is determined by adding and subtracting the quantiles of the conformity scores of the same training set. Therefore, it often leads to over-optimistic and tends to underestimate the width of prediction intervals, resulting in lower coverage than the target. For the Naïve method, it does not have a theoretical coverage guarantee, and its typical coverage is less than $(1-2a)$ (37; 51). This characteristic of the method limits its ability to provide reliable uncertainty estimates for critical applications, but it serves as the baseline in our comparison.

CV+: The cross-validation method involves splitting the training data into K disjoint subsets of equal size, fitting regression functions on the training set without the corresponding k^{th} fold, and computing out-of-fold conformity scores for each data point. This method ensures a coverage level exceeding $(1-2a)$ for a target coverage level of $(1-a)$ without relying on data distribution assumption. It is more conservative and often results in larger prediction intervals (37; 51).

Jackknife+-after-Bootstrap: The Jackknife+-after-Bootstrap involves four key steps: resampling the training set K times to create bootstraps, fitting K regression functions on these bootstraps while excluding the current observation, aggregating predictions with an aggregation function and calculate the conformity scores, and estimate the prediction intervals. Like CV+, it guarantees a $(1-2a)$ coverage with a target coverage level of $(1-a)$ without relying on data distribution assumption (37; 51).

Conformalized Quantile Regression: Unlike the above three methods, the conformalized quantile regression is designed to improve interval estimation for heteroscedastic datasets. It leverages quantile regression at different quantile levels to determine prediction boundaries and the residuals of these methods are used to guarantee a certain coverage probability. The Conformalized Quantile Regression provides at least $(1-a)$ theoretical coverage with similar typical coverage. In summary, this method allows the derivation of local interval widths while maintaining statistical guarantees (37).

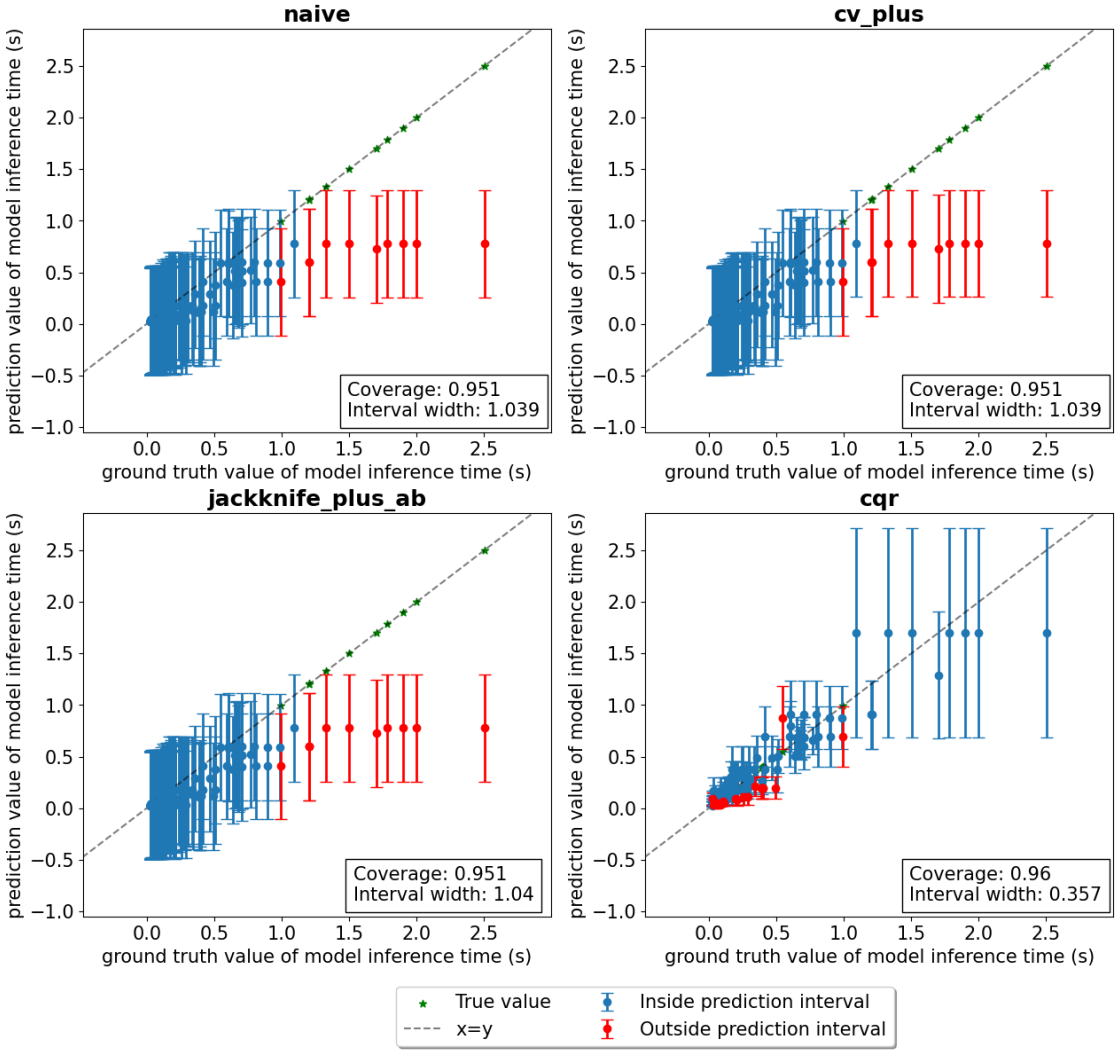

Figure 4.1: Predicted vs. True Values with Prediction Intervals for Different Methods with MAPIE

Surveillance System Feature and Conformal Method Selection: We compare the above con-

formal approaches to determine which method best suits our uncertainty estimation needs in our case. In our uncertainty quantification, we used an a of 5%, aiming for a target coverage probability of 95%. This target coverage implies that the prediction intervals are expected to contain the true value of the dependent variable 95% of the time as shown in Figure 4.1. Additionally, there are a total of three surveillance system features, but we question whether all of them are necessary. Therefore, we trained models with various combinations of feature inputs to determine the minimal necessary surveillance system features. The combinations are the following four: num_of_cpu, num_of_cpu and num_of_client, num_of_cpu and image_size, and num_of_cpu and num_of_client and image_size.

```
Methods = {
    "naive": {"method": "naive"},
    "cv_plus": {"method": "plus", "cv": 10},
    "jackknife_plus_ab": {"method": "plus", "cv": Subsample(n_resamplings=50)},
    "cqr": {"method": "quantile", "cv": "split"},
}
```

Figure 4.2: Method Parameters

For the CQR, we apply a split-conformal approach to partition the dataset into separate training and calibration sets. This was achieved by using the MapieQuantileRegressor with cv="split". In contrast, the other conformal methods were executed via the MapieRegressor, with the cv parameter defining the cross-validation strategy as shown in Figure 4.2. To be more specific, we defined the CV+ method with 10 folds and jackknife+-after-Bootstrap with resampling 50 times to create bootstrap.

Comparison of Conformal Methods

While the other methods produce a fixed interval width, the CQR method shows the width variation of the prediction. This adaptivity allows the intervals to adjust according to the uncertainty inherent in the data at different resource and setup configurations. Additionally, the CQR method has a much narrower overall interval width of 0.357 compared to the other methods, which had interval widths around 1.039 to 1.04. A narrower interval with even better coverage suggests a more precise and desirable estimation of uncertainty.

Considering the above reasoning, we have decided to assess the predictive uncertainty estimation through the Conformal Quantile Regression approach only.

Next, we analyze the model performance with CQR for surveillance feature selection. As shown in the Table 4.2, we observe that while num_of_cpu alone can provide $(1-a)$ target coverage, the combination of num_of_cpu and num_of_client achieve similar coverage while

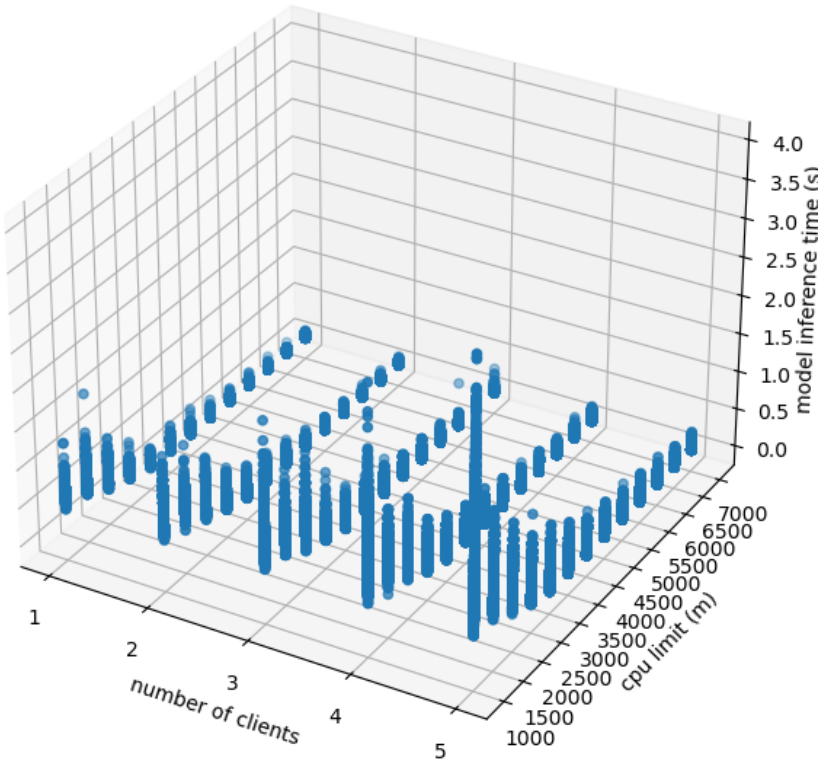

Figure 4.3: Relationship Between Model Inference Time, CPU and Clients

maintaining a narrower interval width. The image_size feature does not have a significant relationship with the predicted outcome and tends to introduce noise into the model, affecting the confidence interval coverage. This results in the actual coverage dropping below the expected level of $(1-2a)$.

A better visualization of these two features' versus model inference time is shown in Figure 4.3. With the zoom in with Figure 4.4, it shows a clear trend of improved performance with increased CPU allocations. The mean execution time decreases significantly from 1.197767 seconds at 1000 milliCPUs to 0.050386 seconds at 7000 milliCPUs as presented in Table 4.1. The most substantial improvements are observed at lower CPU allocations, with a sharp decrease in execution times from 1000 to 4000 milliCPUs. Beyond 4000 milliCPUs, the rate of improvement in execution times diminishes, indicating diminishing returns on performance gains with increased CPU resources. As we shift our attention to the impact of client concurrency on model inference time, Figure 4.5 shows a linear degradation in system performance. This suggests that the system's throughput is adversely affected by client concurrency. Given the above reason, we then conclude that we only have num_of_cpu and num_of_client as feature

Table 4.1: Correlation Between CPU Resources and Model Inference Time

num_of_cpu (m)	Model Inference Time (s)		
	mean	max	min
1000	1.197767	3.933644	0.413501
1500	0.797682	1.800719	0.309294
2000	0.592341	1.497574	0.205622
2500	0.337482	1.277351	0.108062
3000	0.231954	0.904013	0.024618
3500	0.201751	0.699977	0.024266
4000	0.145994	0.605177	0.023122
4500	0.115856	0.493733	0.023395
5000	0.099366	0.497313	0.023413
5500	0.086114	0.405838	0.022732
6000	0.065154	0.597264	0.022735
6500	0.055331	0.303467	0.022908
7000	0.050386	0.353289	0.022674

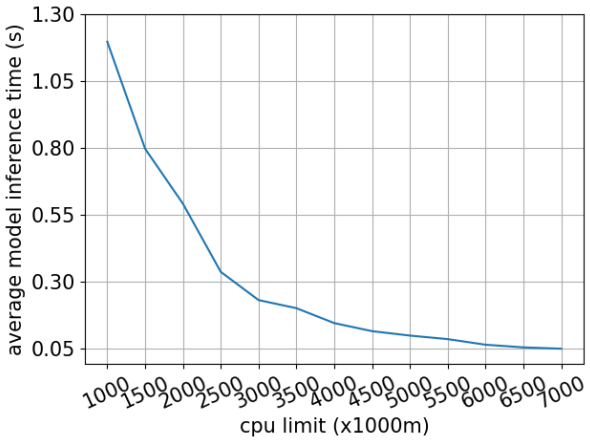

Figure 4.4: Average Model Inference Time vs CPU Limit

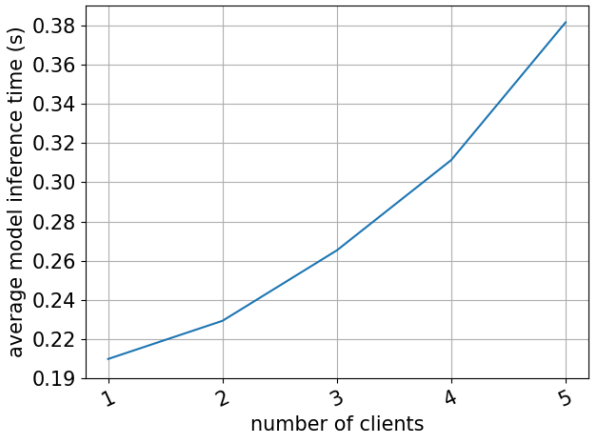

Figure 4.5: Average Model Inference Time vs Number of Clients

input to the CQR model.

Figures 4.6 to 4.10 show the confidence bands for five different client instances. The confidence band is the blue-shaded area aiming for target coverage of 95%, the actual percentage of points failing in the confidence band is shown beyond the a-xis. While the red line indicates the prediction average, the blue line shows as the ground truth average.

Accuracy of Predictions: As shown in our figures, the ground truth average closely aligns with the prediction average, indicating that the CQR model predicts well across different CPU limits. However, there are some regions, especially at CPU limits from 2500 milliCPUs to 3000

Table 4.2: Model Performance with Different Features Part 1

	num_of_cpu	**num_of_-cpu+num_-of_client**	**num_of_-cpu+image_-size**	**num_of_-cpu+num_-of_-client+image_-size**
Effective mean width score obtained by the prediction intervals (CQR) in second	0.465	0.357	0.371	0.288
Actual 95% coverage for test data (CQR)	95.93%	95.99%	88.64%	89.11%

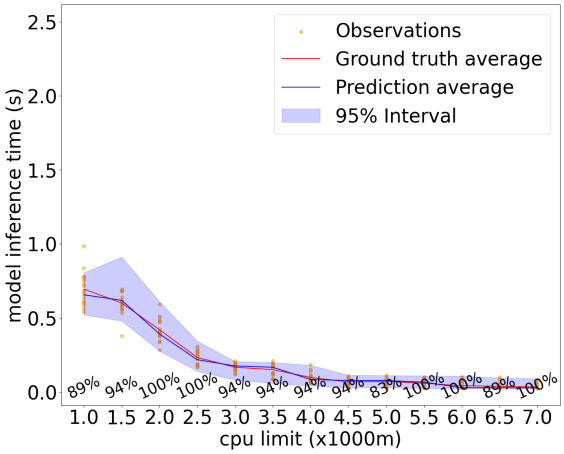

Figure 4.6: Confidence Band for CQR with 1 Client

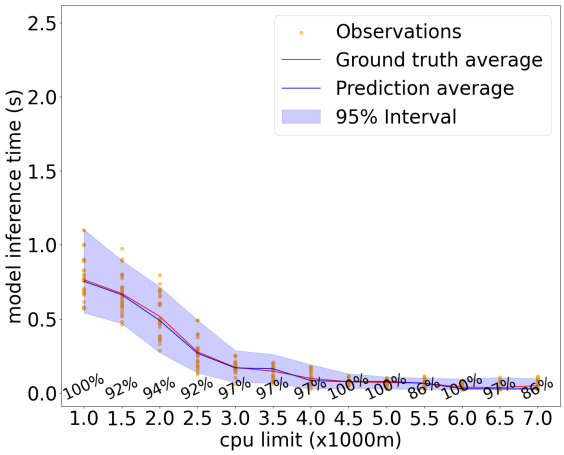

Figure 4.7: Confidence Band for CQR with 2 Clients

milliCPUs with higher concurrent client count, where the ground truth and prediction averages diverge.

Interval Estimation: As shown in our figures, confidence interval widths from Client 1 to Client 5 show a clear trend: as the number of clients increases, the intervals become gradually wider, especially at lower CPU limits (from 1000 milliCPUs to 4000 milliCPUs). For instance, with 1 Client, the interval width is approximately 0.25 seconds, which is much narrower than that of 5 clients, where the interval width escalates to around 1.5 seconds. This suggests that the uncertainty in model inference time escalates with an increasing number of clients. This could reflect a higher data variability to the CPU limit parameter as the client count increases. In contrast, at higher CPU limits (4500 milliCPUs to 7000 milliCPUs), the interval widths across different client counts demonstrate more stability, suggesting a more consistent model performance in scenarios with greater CPU resources. The coverage probability across different

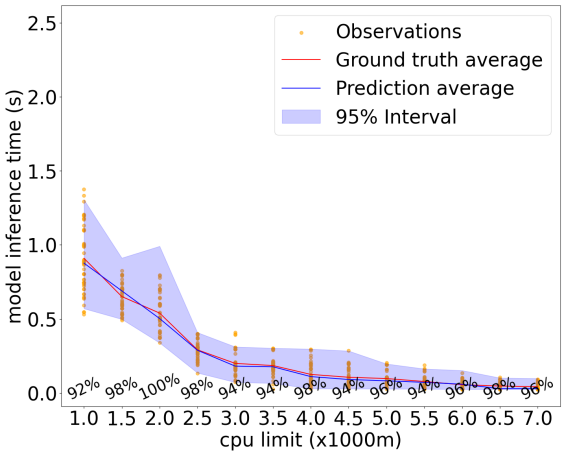

Figure 4.8: Confidence Band for CQR with 3 Clients

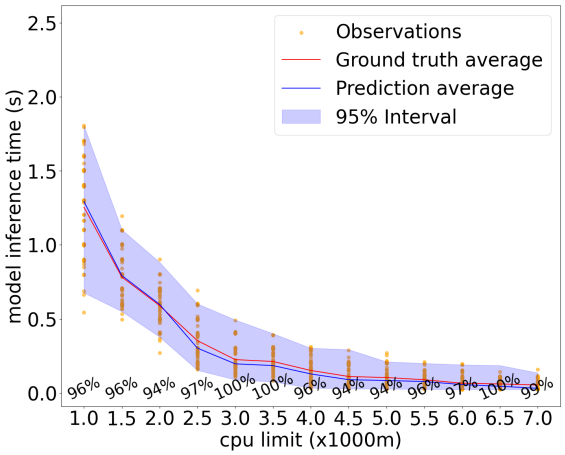

Figure 4.9: Confidence Band for CQR with 4 Clients

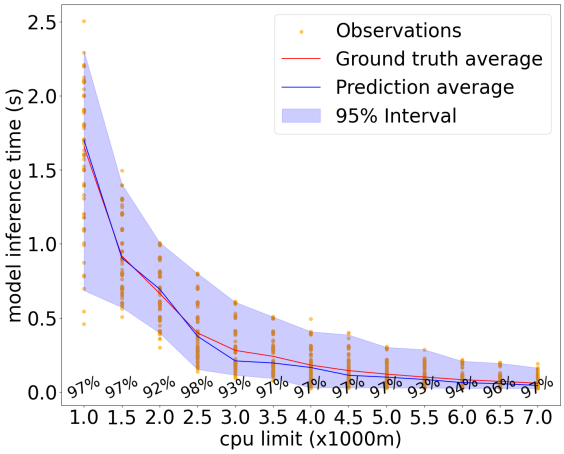

Figure 4.10: Confidence Band for CQR with 5 Clients

clients and CPU limits is satisfying with most observed values staying around 95% threshold.

4.3 Uncertainty Quantification with PNN

After using conformal theory for uncertainty quantification, we next proceed to explore deep learning with PNN for comparison.

PNN typically consists of four layers: an input layer, a hidden layer, a summation layer, and an output layer. First, the input layer is responsible for feeding feature vectors into the network. We define the input dimension as 2 since we only have two features in the dataset. Second, The neurons of the hidden layer process the input data, measure similarity, and generates

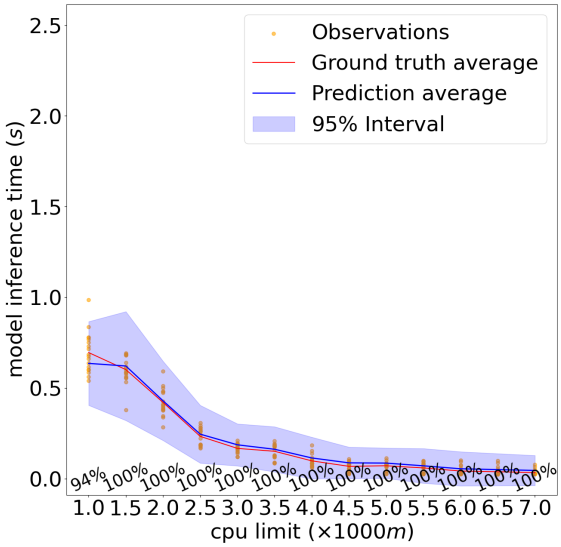

Figure 4.11: Confidence Band for PNN with 1 Client

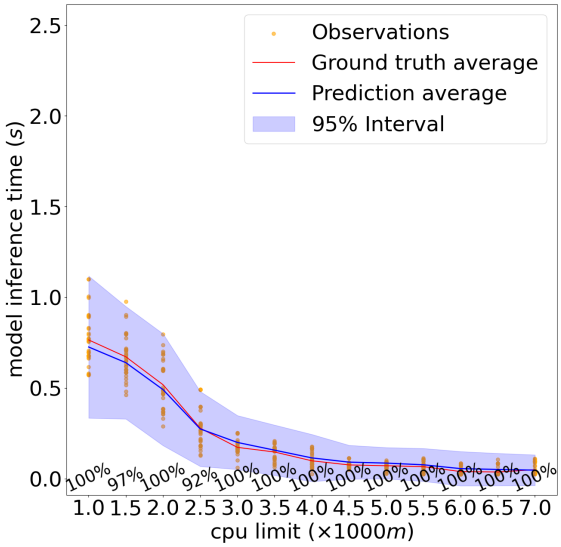

Figure 4.12: Confidence Band for PNN with 2 Clients

a pattern of activation. In our practice, we employ two hidden layers, consisting of 64 and 128 neurons respectively to capture non-linear relationships between input features and the predicted output. Third, the summation layer with 64 neurons collects the outputs from the pattern layer corresponding to each class, and sums up the contributions passing to the output layer. Finally, the output layer is responsible for outputting the mean and standard deviation of a Gaussian distribution.

In parallel with the previous assessment utilizing CQR, we will compare the PNN performance in the accuracy of predictions and interval estimation.

Accuracy of Predictions: As shown in Figure 4.11 to 4.15, the PNN tends to have better alignment between the ground truth average with prediction average, especially in CPU limits from 2500 milliCPUs to 3000 milliCPUs with higher concurrent client count.

Interval Estimation: Contrasting with the CQR approach, the PNN-generated prediction intervals are broader, showing conservativeness in uncertainty estimation. The interval width is determined symmetrically around the prediction average based on the mean and standard deviation. In contrast, CQR produces asymmetric bounds by conformal method. We can further validate the conservativeness of PNN by observing that it achieves almost 100% coverage at all resource configurations. Given our pre-defined confidence threshold of 95%, this estimation is somewhat overly conservative, even though it still satisfies a strict conformal guarantee.

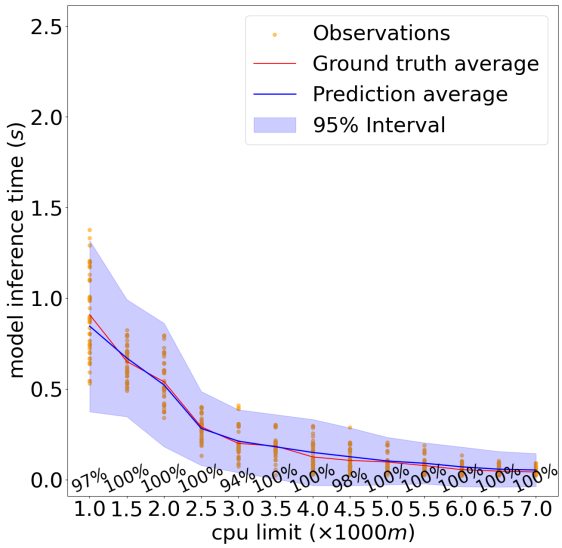

Figure 4.13: Confidence Band for PNN with 3 Clients

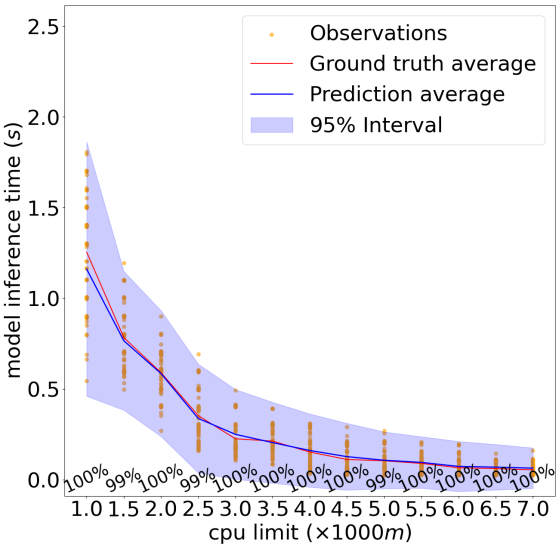

Figure 4.14: Confidence Band for PNN with 4 Clients

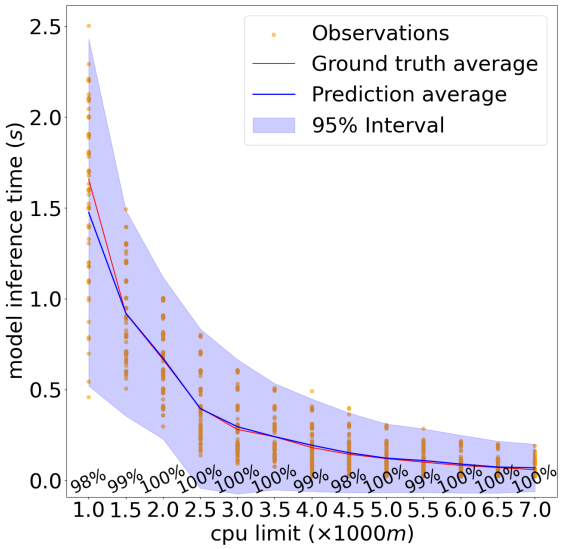

Figure 4.15: Confidence Band for PNN with 5 Clients

CHAPTER

5

DISCUSSION

In this chapter, we extend our discussion on additional features and static configuration recommendation of our application.

5.1 Discussion on Additional Features

In addition to the three surveillance system features discussed, our system also obtains observability on a wide range of system utilization metrics, captured by psutil, as well as GPU resources.

Table 5.1: Description of Additional Features

Feature	Description	Unit
memory_percent_system	Percentage of system memory used	%
memory_percent_process	Percentage of memory used by the process	%
cached_memory_system	Amount of system memory cached	gigabytes
cpu_percent_system	Percentage of CPU utilization by the system	%
cpu_percent_process	Percentage of CPU utilization by the process	%
resident_set_size_process	Resident set size for the process	gigabytes

5.1.1 System Utilization Metrics

psutil is a cross-platform library that collects information on running processes and system utilization in support of system monitoring and profiling. It supports platforms such as Linux, Windows, and macOS. With the help of psutil, we aim to explore features within the system that could possibly enhance prediction accuracy and reduce data uncertainty. The collected system utilization metrics and their correlation coefficient to model inference time is shown in Table 5.1 and Table 5.2.

Table 5.2: Correlation Coefficient Between Model Inference Time and System Utilization Metrics

	cpu_percent_system	**cpu_percent_process**	**memory_percent_system**	**memory_percent_process**	**cached_memory_system**	**resident_set_size**
model_time	-0.52	-0.52	-0.072	0.002	-0.1	0.002

Suggested by the strength of the correlation table, we select the feature with the highest coefficient, cpu_percent_system, as an additional feature for training and prediction with CQR (6). cpu_percent_process is not selected because it has a 0.98 correlation with cpu_percent_system. The result predictions are presented from Figure 5.1 to Figure 5.5. Suggested by Table 5.3, cpu_percent_system introduces overfitting to the CQR, resulting in only 70.28% accuracy with test data.

Table 5.3: Model Performance with Different Features Part 2

	num_of_cpu	**num_of_cpu+num_of_client**	**num_of_cpu+num_of_client+cpu_percent_system**
Effective mean width score obtained by the prediction intervals (CQR) in second	0.465	0.357	0.277
Actual 95% coverage for test data (CQR)	95.93%	95.99%	68.03%

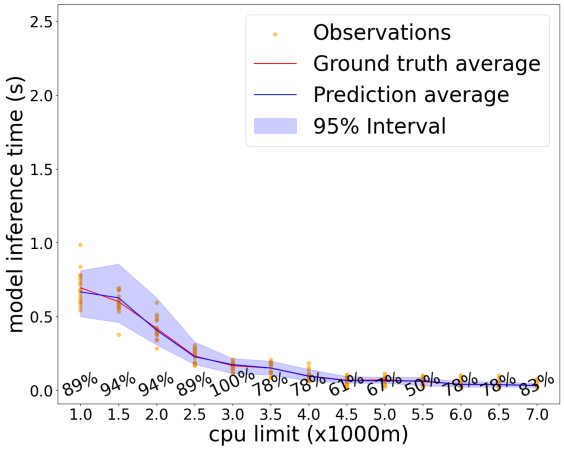

Figure 5.1: Confidence Band for CQR with 1 Client and cpu_percent_system Feature

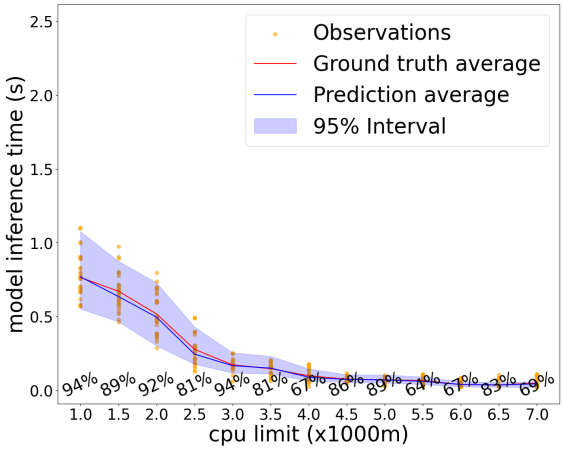

Figure 5.2: Confidence Band for CQR with 2 Clients and cpu_percent_system Feature

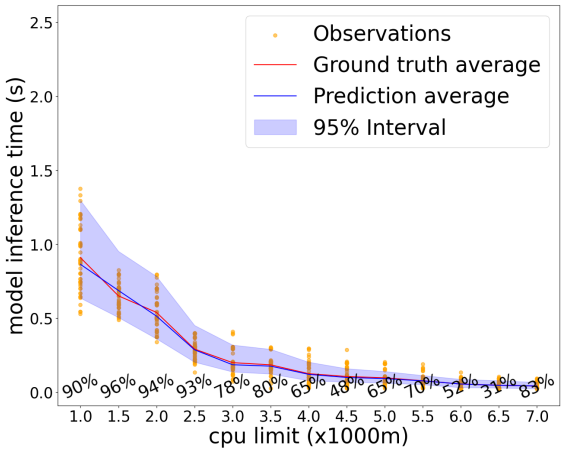

Figure 5.3: Confidence Band for CQR with 3 Clients and cpu_percent_system Feature

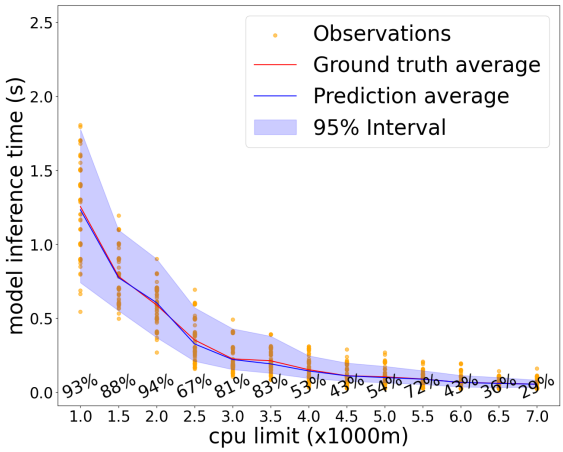

Figure 5.4: Confidence Band for CQR with 4 Clients and cpu_percent_system Feature

5.1.2 GPU Resources

Our experiment also explores multi-instance GPU (MIG) technology, initially introduced by NVIDIA in 2020. We can create GPU Instances by specifying one of the GPU Instance Profiles. An example profile name of MIG 1g.5gb means allocating 1 slice of the GPU's compute resources and 5GB of the GPU's memory (40). By harnessing GPU resource sharing, we aim to discover how different GPU profile instances affect the performance of processing our object detection application. However, it turns out that our application can not saturate the utilization even the smallest GPU partition, the MIG 1g.5gb. When comparing Table 5.4 and Table 5.5 to Table 4.1, it suggests that allocating the smallest GPU partition along with 1000 milliCPUs still outperforms an allocation of 7000 milliCPUs. Additionally, we observe that increasing GPU power has little performance improvement.

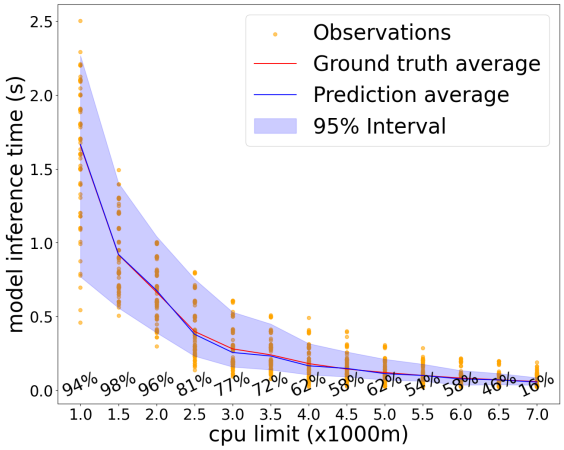

Figure 5.5: Confidence Band for CQR with 5 Clients and cpu_percent_system Feature

Table 5.4: GPU Performance with 1000 milliCPUs and 5 Clients

GPU	Model Inference Time (s)		
	mean	max	min
MIG 1g.5gb	0.017409	0.094165	0.015768
MIG 2g.10gb	0.012079	0.088560	0.010833
MIG 3g.20gb	0.011240	0.096111	0.009054

Table 5.5: GPU Performance with 7000 milliCPUs and 5 Clients

GPU	Model Inference Time (s)		
	mean	max	min
MIG 1g.5gb	0.016491	0.078189	0.015767
MIG 2g.10gb	0.011429	0.022182	0.010749
MIG 3g.20gb	0.010482	0.054224	0.009112

5.2 Static Configuration Recommendation

In the discussion, we want to demonstrate that our approach can be used to satisfy robust performance guarantees by accounting for uncertainty while also respecting resource constraints and minimizing cost. Consider we want to achieve a latency of 1 second for 5 client scenarios. If we do not consider uncertainty, we would allocate 1500 milliCPUs to minimize the cost while satisfying the latency bound. However, this would then result in s violation of the latency bound at 36.67%. On the contrary, by considering uncertainty, we would allocate 2000 milliCPUs or 2500 milliCPUs, which can meet the latency bound with a coverage of 95.56% or 100% respectively.

Another discussion topic is that the frame rate plays a crucial role in the domain of video surveillance. The rate at which frames are captured and processed significantly impacts the effectiveness of surveillance systems. The human eye is capable of perceiving roughly 15 separate images per second, making this the lower threshold for what is considered smooth motion (41). CCTV systems typically operate up to 30 fps, with anything above 15 fps regarded as presenting a fluid motion to the human observer. However, practical considerations such as

storage volume necessitate the use of lower frame rates in many systems, with 15 fps and 7.5 fps being common standards (55).

Inspired by the above, we propose a methodology that translates the frame rates supported by conventional surveillance cameras into a corresponding 'satisfactory Model Inference Time.', highlighted in yellow in Figure 5.6. This serves as a pass-fail criterion: if the average inference time of a model, given a specific resource configuration, falls below the satisfactory Model Inference Time, we endorse this configuration for our Image Processing application, indicated in green; otherwise, it's marked in red to indicate failure. The table illustrates that for a surveillance system operating at 5 fps, a CPU allocation exceeding 4000 milliCPUs is advisable. For the more prevalent 7.5 fps and 15 fps systems, CPU resources greater than 4500 milliCPUs and 6000 milliCPUs are recommended respectively. For systems aiming to achieve 30 fps, our data indicates that only configurations with GPU support fulfill the requirements.

Aggegrated FPS	**Satisfied Model Inference Time**	**CPU = 4000m**	**CPU = 4500m**	**CPU = 5000m**	**CPU = 5500m**	**CPU = 6000m**	**CPU = 6500m**	**CPU = 7000m**	**MIG 1g.5gb with CPU = 1000**	**MIG 2g.10gb with CPU = 1000**	**MIG 3g.20gb with CPU = 1000**
5	0.2	0.146	0.1159	0.0994	0.0861	0.0652	0.0553	0.0504	0.0174	0.01208	0.01124
7.5	0.1333	0.146	0.1159	0.0994	0.0861	0.0652	0.0553	0.0504	0.0174	0.01208	0.01124
10	0.1	0.146	0.1159	0.0994	0.0861	0.0652	0.0553	0.0504	0.0174	0.01208	0.01124
15	0.0667	0.146	0.1159	0.0994	0.0861	0.0652	0.0553	0.0504	0.0174	0.01208	0.01124
30	0.0333	0.146	0.1159	0.0994	0.0861	0.0652	0.0553	0.0504	0.0174	0.01208	0.01124

Figure 5.6: Configuration Recommendation based on Frame Rate

CHAPTER

6

CONCLUSIONS

6.1 Summary of Results

In this thesis, we implemented an end-to-end framework to profile an intelligent surveillance application. The microservice resource-performance profiling resulted in 369000 data points under 201 different resource and configuration setups. Based on the measurement data, we performed feature analysis and trained and evaluated on two machine learning models: Conformalized Quantile Regression and Probabilistic Neural Network.

6.2 Overarching Conclusions

In summary, our paper underscores the effectiveness of both the Conformalized Quantile Regression and Probabilistic Neural Network in performance prediction and estimating uncertainty within our benchmark dataset. However, our comparative analysis indicates that within microservices environments for video processing tasks, Conformalized Quantile Regression is a better choice than Probabilistic Neural Networks since the latter shows more conservativeness. We also conjecture that the increased variability of model inference time at low CPU resources is because of the potential random queueing effects when the worker node becomes congested with low resources.

6.3 Future Work

For future research directions, we propose the following areas of focus to build upon the foundation laid by our current work:

Exploring More Computationally Intensive Applications: Given the dominant role of GPUs in AI-driven tasks, it is evident that our current benchmarking on image processing tasks does not fully exploit the dynamic capabilities of GPU resources. Future work should involve benchmarking more computationally intensive applications to measure and model the impact of different GPU configurations on performance. This will allow for a deeper understanding of how GPU partitioning can be optimized for varying levels of computational demand.

Emulating a More Complex Surveillance Ecosystem: Our present study is limited to deploying a single image processing application and limited concurrent clients within the intelligent surveillance system. To more accurately reflect real-world scenarios, future research should aim to increase the size of the participant client base and operate multiple services concurrently, incorporating a diverse array of surveillance AI applications. This setup will ensure that the benchmark data more closely aligns with practical deployment environments, enhancing the applicability and relevance of our findings.

Implementing Dynamic Resource Allocation: The availability of dynamic resource allocation mechanisms, such as customized schedulers in Kubernetes, presents an opportunity to further optimize system performance. Based on forthcoming benchmark data, we propose to develop and employ a dynamic scheduling system. This system would allocate resources and manage applications adaptively, based on real-time demand and predictive analytics, within the intelligent surveillance framework. Such a strategy would significantly enhance the efficiency and responsiveness of surveillance systems to varying operational requirements.

By pursuing these avenues, we aim to address the current limitations and open new pathways for the advancement of intelligent surveillance systems. These efforts will not only refine our understanding of resource optimization in AI-driven environments but also contribute to the development of more robust, scalable, and efficient surveillance solutions.

REFERENCES

[1] How can you evaluate uncertainty in your deep learning. https://www.linkedin.com/advice/0/how-can-you-evaluate-uncertainty-your-deep-learning. Accessed: 2024-03-14.

[2] With mapie, uncertainties are back in machine learning. https://towardsdatascience.com/with-mapie-uncertainties. Accessed: 2024-03-14.

[3] Almamon Rasool Abd Ali. *A proposed Intelligent Surveillance System for Smart Cities Using Microservice Architecture*. PhD thesis, University of Technology, 2019.

[4] Moloud Abdar, Farhad Pourpanah, Sadiq Hussain, Dana Rezazadegan, Li Liu, Mohammad Ghavamzadeh, Paul Fieguth, Xiaochun Cao, Abbas Khosravi, U Rajendra Acharya, et al. A review of uncertainty quantification in deep learning: Techniques, applications and challenges. *Information fusion*, 76:243–297, 2021.

[5] Charles Anderson. Docker [software engineering]. *IEEE Software*, 32(3):102–c3, 2015.

[6] Agustin Garcia Asuero, Ana Sayago, and AG González. The correlation coefficient: An overview. *Critical reviews in analytical chemistry*, 36(1):41–59, 2006.

[7] Atul Saini. The emergence of microservices. https://www.fiorano.com/blogs/microservices_architecture, 2017. Accessed: 2024-03-06.

[8] David A Bacigalupo, Jano Van Hemert, Asif Usmani, Donna N Dillenberger, Gary B Wills, and Stephen A Jarvis. Resource management of enterprise cloud systems using layered queuing and historical performance models. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. IEEE, 2010.

[9] Andre Bento, Jaime Correia, Ricardo Filipe, Filipe Araujo, and Jorge Cardoso. Automated analysis of distributed tracing: Challenges and research directions. *Journal of Grid Computing*, 19:1–15, 2021.

[10] David Bermbach, Jörn Kuhlenkamp, Akon Dey, Arunmoezhi Ramachandran, Alan Fekete, and Stefan Tai. Benchfoundry: a benchmarking framework for cloud storage services. In *Service-Oriented Computing: 15th International Conference, ICSOC 2017, Malaga, Spain, November 13–16, 2017, Proceedings*, pages 314–330. Springer, 2017.

[11] Romil Bhardwaj, Kirthevasan Kandasamy, Asim Biswal, Wenshuo Guo, Benjamin Hindman, Joseph Gonzalez, Michael Jordan, and Ion Stoica. Cilantro:{Performance-Aware} resource allocation for general objectives via online feedback. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 623–643. USENIX Association, 2023.

[12] Justus Bogner, Jonas Fritzsch, Stefan Wagner, and Alfred Zimmermann. Microservices in industry: insights into technologies, characteristics, and software quality. In *2019 IEEE international conference on software architecture companion (ICSA-C)*, pages 187–195. IEEE, 2019.

[13] Amir Hossein Borhani, Philipp Leitner, Bu-Sung Lee, Xiaorong Li, and Terence Hung. Wpress: An application-driven performance benchmark for cloud-based virtual machines. In *2014 ieee 18th international enterprise distributed object computing conference*, pages 101–109. IEEE, 2014.

[14] Luiz Carvalho, Alessandro Garcia, Wesley KG Assunção, Rafael de Mello, and Maria Julia de Lima. Analysis of the criteria adopted in industry to extract microservices. In *2019 IEEE/ACM Joint 7th International Workshop on Conducting Empirical Studies in Industry (CESI) and 6th International Workshop on Software Engineering Research and Industrial Practice (SER&IP)*, pages 22–29. IEEE, 2019.

[15] Tianqi Chen, Tong He, Michael Benesty, Vadim Khotilovich, Yuan Tang, Hyunsu Cho, Kailong Chen, Rory Mitchell, Ignacio Cano, Tianyi Zhou, et al. Xgboost: extreme gradient boosting. *R package version 0.4-2*, 1(4):1–4, 2015.

[16] Douglas Cumming and Sofia Johan. Cameras tracking shoppers: the economics of retail video surveillance. *Eurasian Business Review*, 5:235–257, 2015.

[17] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering*, pages 195–216, 2017.

[18] Omar Elharrouss, Noor Almaadeed, and Somaya Al-Maadeed. A review of video surveillance systems. *Journal of Visual Communication and Image Representation*, 77:103116, 2021.

[19] Dominik Ernst and Stefan Tai. Offline trace generation for microservice observability. In *2021 IEEE 25th International Enterprise Distributed Object Computing Workshop (EDOCW)*, pages 308–317. IEEE, 2021.

[20] Martin Grambow, Lukas Meusel, Erik Wittern, and David Bermbach. Benchmarking microservice performance: a pattern-based approach. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pages 232–241, 2020.

[21] Bert Hubert et al. Linux advanced routing & traffic control howto. *Netherlabs BV*, 1:99–107, 2002.

[22] Eyke Hüllermeier and Willem Waegeman. Aleatoric and epistemic uncertainty in machine learning: An introduction to concepts and methods. *Machine learning*, 110(3):457–506, 2021.

[23] Sutrisno Warsono Ibrahim. A comprehensive review on intelligent surveillance systems. *Communications in science and technology*, 1(1), 2016.

[24] Byol Kim, Chen Xu, and Rina Barber. Predictive inference is free with the jackknife+-after-bootstrap. *Advances in Neural Information Processing Systems*, 33:4138–4149, 2020.

[25] Eunsook Kim, Kyungwoon Lee, and Chuck Yoo. On the resource management of kubernetes. In *2021 International Conference on Information Networking (ICOIN)*, pages 154–158. IEEE, 2021.

[26] Holger Knoche and Wilhelm Hasselbring. Drivers and barriers for microservice adoption–a survey among professionals in germany. *Enterprise Modelling and Information Systems Architectures (EMISAJ)–International Journal of Conceptual Modeling: Vol. 14, Nr. 1*, 2019.

[27] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7. Athens, Greece, 2011.

[28] T Kubernetes. Kubernetes. *Kubernetes. Retrieved May*, 24:2019, 2019.

[29] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review*, 44(2):35–40, 2010.

[30] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. Simple and scalable predictive uncertainty estimation using deep ensembles. *Advances in neural information processing systems*, 30, 2017.

[31] Antonis Lambrou, Harris Papadopoulos, and Alex Gammerman. Reliable confidence measures for medical diagnosis with evolutionary algorithms. *IEEE Transactions on Information Technology in Biomedicine*, 15(1):93–99, 2010.

[32] Tan N Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. Allox: compute allocation in hybrid clusters. In *Proceedings of the fifteenth european conference on computer Systems*, pages 1–16, 2020.

[33] Jing Lei, Max G'Sell, Alessandro Rinaldo, Ryan J Tibshirani, and Larry Wasserman. Distribution-free predictive inference for regression. *Journal of the American Statistical Association*, 113(523):1094–1111, 2018.

[34] Bowen Li, Xin Peng, Qilin Xiang, Hanzhang Wang, Tao Xie, Jun Sun, and Xuanzhe Liu. Enjoy your observability: an industrial survey of microservice tracing and analysis. *Empirical Software Engineering*, 27:1–28, 2022.

[35] Weixuan Li, Guang Lin, and Bing Li. Inverse regression-based uncertainty quantification algorithms for high-dimensional models: Theory and practice. *Journal of Computational Physics*, 321:259–278, 2016.

[36] Yan Liu and Ian Gorton. Performance prediction of j2ee applications using messaging protocols. In *International Symposium on Component-Based Software Engineering*, pages 1–16. Springer, 2005.

[37] MAPIE Developers. Mapie: Model agnostic prediction interval estimator. `https://mapie.readthedocs.io/en/latest/index.html`, 2023. Accessed: 2024-03-09.

[38] Víctor Medel, Omer Rana, José Ángel Bañares, and Unai Arronategui. Adaptive application scheduling under interference in kubernetes. In *Proceedings of the 9th International Conference on Utility and Cloud Computing*, pages 426–427, 2016.

[39] Jerzy Neyman. On the two different aspects of the representative method: the method of stratified sampling and the method of purposive selection. In *Breakthroughs in Statistics: Methodology and Distribution*, pages 123–150. Springer, 1992.

[40] NVIDIA. Mig user guide. `https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html`, 2023. Accessed: 2024-03-09.

[41] Optiview. Cctv camera resolution | cctv resolution chart for cameras. `https://optiviewusa.com/cctv-video-resolutions/`, 2024. Accessed: 2024-03-03.

[42] Jay Parmar, Sakshi Sanghavi, Vivek Prasad, and Pooja Shah. Microservice architecture observability tool analysis. In *International Conference on Soft Computing and Signal Processing*, pages 1–8. Springer, 2022.

[43] Fatih Porikli, Francois Bremond, Shiloh L Dockstader, James Ferryman, Anthony Hoogs, Brian C Lovell, Sharath Pankanti, Bernhard Rinner, Peter Tu, and Péter L Venetianer. Video surveillance: past, present, and now the future [dsp forum]. *IEEE Signal Processing Magazine*, 30(3):190–198, 2013.

[44] Apostolos F Psaros, Xuhui Meng, Zongren Zou, Ling Guo, and George Em Karniadakis. Uncertainty quantification in scientific machine learning: Methods, metrics, and comparisons. *Journal of Computational Physics*, 477:111902, 2023.

[45] VN Ganapathi Raju, K Prasanna Lakshmi, Vinod Mahesh Jain, Archana Kalidindi, and V Padma. Study the influence of normalization/transformation process on the accuracy of supervised classification. In *2020 Third International Conference on Smart Systems and Inventive Technology (ICSSIT)*, pages 729–735. IEEE, 2020.

[46] Raul Ramírez-Velarde, Andrei Tchernykh, Carlos Barba-Jimenez, Adán Hirales-Carbajal, and Juan Nolazco-Flores. Adaptive resource allocation with job runtime uncertainty. *Journal of Grid Computing*, 15:415–434, 2017.

[47] Yaniv Romano, Evan Patterson, and Emmanuel Candes. Conformalized quantile regression. *Advances in neural information processing systems*, 32, 2019.

[48] Rami Rosen. Resource management: Linux kernel namespaces and cgroups. *Haifux, May*, 186:70, 2013.

[49] Shubham Munde. Market research report- forecast to 2030. `https://www.marketresearchfuture.com/reports/microservices-architecture-market-3149`, 2024. Accessed: 2024-03-24.

[50] Viktória Spišaková, Dalibor Klusáček, and Lukáš Hejtmánek. Using kubernetes in academic environment: Problems and approaches. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 235–253. Springer, 2022.

[51] Vianney Taquet, Vincent Blot, Thomas Morzadec, Louis Lacombe, and Nicolas Brunel. Mapie: an open-source library for distribution-free uncertainty quantification. *arXiv preprint arXiv:2207.12274*, 2022.

[52] Team Aspecto. Jaeger tracing: A friendly guide for beginners. `https://medium.com/jaegertracing/jaeger-tracing-a-friendly-guide-for-beginners`, note = Accessed: 2024-03-09, 2022.

[53] Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[54] Johannes Thönes. Microservices. *IEEE Software*, 32(1):116–116, 2015.

[55] Vassilios Tsakanikas and Tasos Dagiuklas. Video surveillance systems-current status and future trends. *Computers & Electrical Engineering*, 70:736–753, 2018.

[56] Typito. A complete list of video resolutions and their pixel size. `https://typito.com/blog/video-resolutions/`, July 2022. Accessed: 2024-03-05.

[57] Maria Valera and Sergio A Velastin. Intelligent distributed surveillance systems: a review. *IEE Proceedings-Vision, Image and Signal Processing*, 152(2):192–204, 2005.

[58] Kush R Varshney and Homa Alemzadeh. On the safety of machine learning: Cyber-physical systems, decision sciences, and data products. *Big data*, 5(3):246–255, 2017.

[59] Cong Xu, Karthick Rajamani, and Wesley Felter. Nbwguard: Realizing network qos for kubernetes. In *Proceedings of the 19th international middleware conference industry*, pages 32–38, 2018.

[60] Fan Yang, Hua-zhen Wang, Hong Mi, Cheng-de Lin, and Wei-wen Cai. Using random forest for reliable classification and cost-sensitive learning for medical diagnosis. *BMC bioinformatics*, 10:1–14, 2009.

APPENDICES

APPENDIX

A

ACRONYMS

A summary of all acronyms is documented in Table A.1.

Table A.1: A summary of acronyms used in alphabetical order.

Acronym	Abbreviation
Application Programming Interface	API
Blue, Green, Red	BGR
Central Processing Unit	CPU
Closed-Circuit Television	CCTV
Conformal Prediction	CP
Conformalized Quantile Regression	CQR
Cross-Validation-Plus	CV+
Frames Per Second	fps
Gigabytes	gb
Graphics Processing Unit	GPU
High Definition	HD
Hypertext Transfer Protocol	HTTP
Identifier	ID
Intelligent Surveillance System Model	ISSM

Inverse Regression-based UQ	IRUQ
Kilobytes	KB
Light Gradient Boosting Machine	LightGMB
milliCPU	m
Model Agnostic Prediction Interval Estimator	MAPIE
Multi-instance GPU	MIG
Probabilistic Neural Networks	PNN
Representational State Transfer	REST
Ultra High Definition	UHD
Uncertainty Quantification	UQ
User Datagram Protocol	UDP
Yet Another Markup Language	YAML

APPENDIX

B

VARIABLES

A summary of all variables is documented in Table B.1.

Table B.1: A summary of common meteorological variables and their abbreviations in alphabetical order.

Variable	Abbreviation
Interval uncertainty level	a
Number of folds in cross-validation	K