APPLICATION OFFLOADING SCHEMAS FOR CLOUD ROBOTICS

by

Manoj R Penmetcha

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

Department of Computer and Information Technology

West Lafayette, Indiana

May 2021

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

This dissertation is dedicated to my wife, Nikhita Alluri, my parents, and my in-laws, for their endless support and love.

ACKNOWLEDGMENTS

In our life, we meet few people who can deeply influence the course of our life; one such fortunate event led me to meet Dr. Byung-Cheol Min in 2012. As my chair, Dr. Min was always supportive and encouraged me to pursue research based on my strengths; he has given me complete freedom to explore research ideas, without any objections. I will forever be indebted to Dr. Min for his constant support, guidance, and working late nights with me to improve and elevate my research. He has been my consistent source of inspiration, and association with him is something I will cherish throughout my life.

I would also like to thank my committee members, Dr. J. Eric Dietz, Dr. Baijian Yang, and Dr. Jin Wei-Kocsis, for providing valuable feedback, support, and insights. One of my early research topics involved algae detection with machine learning, and was based on ideas suggested and foundations conceived by my committee members. This research topic was the most important motivating factor for me to pursue the research presented in this dissertation, and I am very grateful for everything.

I also want to thank current and former members of the SMART Lab. I made friends whom I will cherish throughout my life. Pursuing my Ph.D. was one of the happiest phases of my life, and they are one of the main reasons for that. And a special mention to my friends, Jun Han Bae, Shaocheng Luo, Shyam Sundar Kannan, Wonse Jo, and Sangjun Lee, for whom I will always be grateful.

The person at Purdue with whom I have the longest association is my assistantship supervisor Matthew Riehle. I am pretty confident that Matt is one of the coolest, kindest, and most supportive bosses I could ever find in my life. With funding support from Purdue libraries, I was able to pursue research ideas without any limitations and with complete free rein. Thank you, Matt, you are one of the biggest reasons for my worry-free and fun-filled graduate life.

I would like to thank my mom and dad for their continued support, love, encouragement, and for instilling in me the importance of higher education. Without their encouragement, I wouldn't have dared to pursue my Ph.D. I would also like to thank my in-laws for standing by me during the last three years. Finally, to the love of my life Nikhita Alluri for all her

sacrifices, unconditional love, and unwavering support. I am very fortunate to have you on my side. This is just the beginning, and I look forward to many bright and colorful years together with you and Dhruv.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

Chapter 4

α	Learning rate
β	Tuning parameters
γ	Discount rate
$\boldsymbol{\pi}$	Policy
θ	Trainable weights
$\theta-$	Target trainable weights
A	Current position of the robot
a	Action
B	Goal position
b	Network latency
$c^{algorithm}$	Algorithm-determined action
c^{cloud}	Computational cost for cloud computation
d	Input data size of application
L	Loss
$l^{lcompletion}$	Timestamp of application completion
l^{local}	Local time of execution
l^{start}	Time stamp when the task was first assigned to the robot
n	Number of nodes
N	Replay memory capacity
Q	Q-function
$\hat{Q}$	Target Q-function
R	Replay memory
r	Reward
s	State
t	Time steps
u	CPU availability
x	Euclidean distance

Chapter 5

Ψ	Predictive algorithm
a	Application task
A	Application
B^c	Queue data structures to store cloud values
B^l	Queue data structures to store local values
c^c	Intercept for Ψ from N previous cloud values
c^l	Intercept for Ψ from N previous local values
d	Size of the data
$\overline{d}$	Mean of the input data size
g	Occupancy grid
m^c	Slope for Ψ from N previous cloud values
m^l	Slope for Ψ from N previous local values
N	Queue with previous N observations
p^c	Predicted cloud execution time
p^l	Predicted local execution time
r	Pearson correlation coefficient
t^c	Actual cloud execution time
$\overline{t^c}$	Mean of actual execution time from N previous cloud values
t^l	Actual local execution time
$\overline{t^l}$	Mean of actual execution time from N previous local values

ABBREVIATIONS

ARIMA	Autoregressive integrated moving average
AWS	Amazon web services
CNN	Convolution neural network
CPU	Central processing unit
DQN	Deep Q-network
DRL	Deep reinforcement learning
FIFO	First-in first-out
GB	Gigabyte
GHz	Gigahertz
GPS	Global positioning system
GPU	Graphics processing unit
HTTP	Hypertext transfer protocol
IaaS	Infrastructure as a service
JS	JavaScript
JSON	JavaScript object notation
LAN	Local area network
LiDAR	Light, detection, and ranging
LQM	Link quality matrix
LSTM	Long short term memory
Mbps	Megabits per second
MDP	Markovian decision process
NPM	Node package manager
OS	Operating system
PaaS	Platform as a service
QOS	Quality of service
RaaS	Robot as a service
RAM	Random access memory
ReLU	Rectified linear unit

RL	Reinforcement learning
ROS	Robot operating system
SaaS	Software as a Service
SARIMA	Seasonal autoregressive integrated moving average
SLAM	Simultaneous localization and mapping
UGV	Unmanned ground vehicle
XML	Extensible markup language
ZMQ	ZeroMQ

ABSTRACT

Robots have inherently limited onboard processing, storage, and power capabilities. Cloud computing resources have the potential to provide significant advantages for robots in many applications. However, to make use of these resources, frameworks must be developed that facilitate robot interactions with cloud services. We first propose a cloud-based architecture called Smart Cloud that intends to overcome the physical limitations of single- or multi-robot systems through massively parallel computation that is provided on demand by cloud services. Smart Cloud is implemented on Amazon Web Services (AWS) and available for robots running on the Robot Operating System (ROS) and on non-ROS systems. Smart Cloud also features a first-of-its-kind architecture that incorporates JavaScript-based libraries to run various robotic applications related to machine learning and other methods. We later develop and validate three types of application offloading algorithms for cloud robotics.

For the first algorithm, we evaluate the architecture performance with respect to a full offloading schema that offloads the whole computation onto the cloud service provider. For the full offloading schema, we deploy the application completely on the cloud without considering the local capabilities of the robot. We validated this schema with machine learning and navigation applications. We evaluated the performance of the architecture both with complete offloading and with no offloading in terms of CPU usage, latency, and security.

The key to effectively offloading tasks is an application solution that does not underutilize the robot's own computational capabilities and makes decisions based on crucial cost parameters such as latency and CPU availability. Hence for the second algorithm, we formulate the application offloading problem as a Markovian decision process and propose a deep reinforcement learning-based deep Q-network (DQN) approach. The state-space is formulated with the assumption that input data size directly impacts application execution time. The proposed algorithm is designed as a continuous task problem with discrete action space; i.e., we apply a choice of action at each time step and use the corresponding outcome to train the DQN to acquire the maximum rewards possible. To validate the proposed algorithm, we designed and implemented a robot navigation testbed. The results demonstrated that for

the given state-space values, the proposed algorithm learned to take appropriate actions to reduce application latency and also learned a policy that takes actions based on input data size. Finally, we compared the proposed DQN algorithm with a long short-term memory (LSTM) algorithm in terms of accuracy. When trained and validated on the same dataset, the proposed DQN algorithm obtained at least 9 percentage points greater accuracy than the LSTM algorithm.

Another key factor to enabling robot use of cloud computing is designing an efficient offloading algorithm that forms a quick consensus on when to offload without any prior knowledge or information about the application. The third algorithm is designed to be trained quickly after the application has been initiated. We propose a predictive algorithm to anticipate the time needed to execute an application for a given application data input size with the help of a small number of previous observations. To validate the algorithm, we train it on the previous N observations, which include independent (input data size) and dependent (execution time) variables. To understand how algorithm performance varies in terms of prediction accuracy and error, we tested various N values using linear regression and a mobile robot path planning application. From our experiments and analysis, we determined the algorithm to have acceptable error and prediction accuracy when $N > 40$.

1. INTRODUCTION

The scope of the robotics industry is immense, and the industry is poised to see huge gains in coming years [1], [2]. The International Data Corporation estimates the 2019 economic value of robotics and related services will hover around \$135.4 billion, and during the period of 2018 to 2023, the industry is estimated to register a compound annual growth of 24.52 percent [3]. This growth is aided by the ubiquitous availability of big data and by recent advancements in machine learning, which can be used to develop smarter and more responsive robots. However, most such applications involve processing large quantities of data, which requires high-performing computational resources [4]. Existing robots come with limited onboard computing capabilities, and once a robot is built, it is not easy to change the hardware configuration. By enabling cloud computing for robotic applications, robots will be able to access increased computational power and storage space as needed to carry out their assigned tasks. With the resources provided by cloud computing services, computationally-intense robotic tasks like object detection, navigation, and others can be solved more efficiently.

Cloud computing is a service-driven paradigm for hosting applications on remote infrastructure, i.e. resources are provided on demand. Since its inception, cloud computing has helped researchers and business users to host applications by providing access to distributed and shared computing resources over the Internet [5]. In practice, the services provided by the cloud can be categorized into three major types: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS) [6].

Cloud robotics, first introduced as a term by James J. Kuffner in 2010 [7], can be defined as the wireless connection of robots to external computing resources to support robot operation. Cloud-enabled robots are able to offload computing tasks to remote servers, thus relying less on their onboard computers and instead exploiting the inexpensive computing power and data storage options provided by cloud service providers. The cloud robotics market is estimated to achieve 23.2 percent compound annual growth; it was valued at around \$2.3 billion in 2017, and is expected to reach \$7.9 billion by the year 2023 [8]. To realize the full potential and scope of cloud robotics, however, it is very important to innovate and address the shortcomings currently faced by the field.

Figure 1.1. Generic cloud robotic architecture showing robots connected to cloud computing through wireless network. [4][© 2015 IEEE].

The fundamental research objective of this dissertation is to overcome the limitations of robotic hardware by leveraging elastic resources in a centralized cloud infrastructure. This study seeks to overcome the limitations of existing cloud robotics solutions through introducing novel architecture and various dynamic offloading schemas, while at the same time incorporating a major emphasis on generalizability, scalability and usability. This research primarily consists of four parts:

Cloud Robotic Architecture, presented in Chapter 3

A cloud robotic architecture is a framework that facilitates robots to connect with cloud infrastructure. The architecture is designed to:

- Support heterogeneous robots.
- Support robots using both ROS and non-ROS systems.
- Be used by robots in both Platform as a Service (PaaS) and Software as a Service (SaaS) implementations.

- Facilitate the usage of JavaScript packages for robotic applications.

Full Application Offloading, presented in Chapter 3

A full offloading schema is where the application is completely deployed on the cloud platform. The full offloading schema will be evaluated based on the following parameters:

- Percentage of CPU utilized.
- Delay in response (latency).

Dynamic Application Offloading with Deep Reinforcement Learning, presented in Chapter 4

A dynamic offloading schema based on deep reinforcement learning is where an algorithm dynamically decides when to offload the application, taking into consideration:

- The robot's local computing capabilities observed from state space variables.
- Latency difference between local computation and cloud service.
- Application input data size.

Dynamic Application Offloading with Predictive Algorithm, presented in Chapter 5

A dynamic offloading schema based on predictive algorithm is where an algorithm dynamically decides when to offload the application, taking into consideration:

- Past execution times of the application.
- Application input data size.

These research objectives will be achieved by theoretical and experimental approaches carried out in the rest of the dissertation.

1.1 Contributions

The chapter wise contributions of this research can be summarized as follows:

Chapter 3:

- Designed a two-tiered cloud robotic architecture, which is the first of its kind to provide access to open-source JavaScript libraries. The architecture is also novel in its approach as it can be used for both SaaS and PaaS depending on the needs of the application. The architecture is designed to work with heterogeneous multi-robot platforms by catering to both ROS-based and non-ROS-based robotic systems. For non-ROS robots, the framework implements a RESTful-based web service to communicate with the robot.

- To illustrate the architecture's performance, we implemented complete application offloading to the cloud. For this purpose, we used a ROS-based TensorFlow object detection package called gmapping and evaluated the performance in terms of CPU utilization, latency, and security.

Chapter 4:

- We formulate the computational offloading problem as a Markovian decision process (MDP) and propose a deep reinforcement learning-based deep Q-network (DQN) approach for its solution.

- We formulate the state space based on the assumption that input data size directly impacts application execution time and define a variable reward function that helps training converge more quickly and learning of a robust policy.

- We analyze the proposed algorithm using a robot navigation application, evaluating it on real data and also generating synthetic data to analyze whether the network is learning the appropriate policy with respect to all possible outcomes.

Chapter 5:

- We introduce a predictive algorithm to predict the execution time of an application under both cloud and onboard computation, based on the application input data size. We further validate the predictive algorithm with a linear regression model.
- The algorithm is designed to be trained after the application has been initiated. To make the algorithm faster to train, we train it on a fixed dataset of N previous observations. We experiment with various N to observe the prediction accuracy and error.
- We employ Gazebo simulation to analyze the proposed algorithm using a robot path planning application.

1.2 Structure of the Dissertation

This dissertation consists of five chapters, including this introductory chapter. Each chapter is absolute and can be read independently. The rest of the proposal is organized as follows. In Chapter 2, we discuss well-known existing solutions and challenges in the area of cloud robotics. We also talk about recently introduced computational offloading models. In Chapter 3, we propose the cloud robotics architecture. We also validate the architecture with computational offloading. In Chapter 4, we propose a deep reinforcement learning-based deep Q-network (DQN) approach for application offloading. In Chapter 5, we propose a predictive algorithm to anticipate the time needed to execute an application for a given application data input size with the help of a small number of previous observations. Finally, Chapter 6 summarizes the current work and outlines future direction for this research.

2. BACKGROUND

This chapter presents background related to cloud computing and cloud robotics. However, each application chapter includes the literature review pertaining to the proposed solution.

The field of robotics is growing at a great pace [1], [2], and one factor driving its growth is the widening range of robotic applications [4]. Robots have made inroads into a variety of complicated application spaces, which developments have mainly been driven by the introduction of highly-sophisticated robots in areas such as industrial robots, personal robots, navigation, and robotic surgery [9], [10]. Among the major impetuses for the sophistication of such high-level robots are recent advancements in information technology [11]. The rapid evolution of technology is led by AI, cloud computing, IoT, blockchain, and other developments, and these technologies have opened up many robotic application spaces that were previously unimaginable [11]. Recent advancements in machine learning [12] are being used to develop smarter robots, and so most of these robotic applications require high-performing computational capabilities to attain a satisfactory level of performance. However, it is often the case that extant robots are equipped with limited computational capabilities, and once a robot is assembled, changing its hardware configuration is not easy.

One thing that is common to all these new technologies is that they require high computational resources for application execution. Meanwhile, robots themselves often come with fixed computing capabilities and so cannot fulfill the computational demands of these systems and other new technological advancements. Providing robots with access to external computational resources such as cloud computing is an effective means of solving this problem [13]. Cloud computing can provide access to on-demand computational resources and significantly enhance application performance for robots that otherwise have only limited computational resources. Providing robot access to the on-demand computing resources offered by cloud service providers can be an effective means of solving this problem [13]. Namely, by taking advantage of the computing power and data storage options provided by cloud services, cloud-enabled robots can rely less on their local computation resources.

The invention of the World Wide Web in the late 1980s opened up the possibility of connecting a robot to an external machine over the Internet. In 1994, K. Goldberg was amongst the first to successfully connect a robot to the web, teleoperating the robot through an Internet browser [14]. Having access to the Internet opened up many verticals in robotics, of which one is cloud robotics. Cloud robotics can be defined as the wireless connection of robots to external computing resources to support robot operations. Cloud-enabled robots facilitate the offloading of computing tasks to remote servers, thus relying less on the on-board computers and allowing the robots access to the increased computational power and storage space required for a number of applications [15]. Cloud computing also facilitates robot access to powerful machine learning tools that can be used for robotic applications.

The area of study enabling robots to utilize cloud computing is termed cloud robotics, which was first coined by James J. Kuffner in 2010 [7]. Cloud robotics algorithms are designed on the basic premise that when robots have insufficient computational resources for local execution of an application, using cloud resources should improve the performance of the application in terms of execution time and energy efficiency. If a robot has the bare minimum of computational capability, full application offloading will be an obvious choice; however, many robots presently being produced are computationally capable, and so dynamic computational offloading algorithms are a wiser choice as they take into account both the robot's computational capability and the application's computational requirements.

In this literature review, we will first discuss the main software components involved in cloud robotics, followed by the latest frameworks in the area of cloud robotics and key evaluation metrics for cloud robotics. Finally, we conclude with the limitations on cloud robotics that we plan to address through this study.

2.1 Software Components

To understand the latest research in the area of cloud robotics, it is important to first be familiar with key software components used in the research. We discuss these components below.

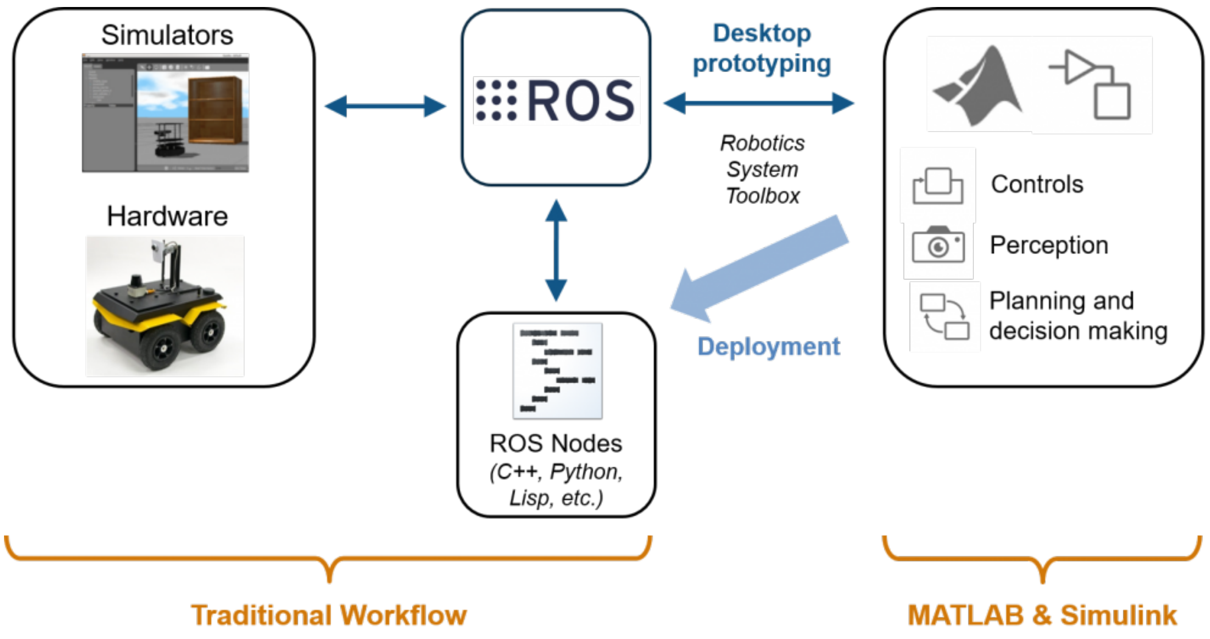


Figure 2.1. Schematic representation for traditional ROS workflow along with desktop prototyping using Matlab. [16][© 2017 Matlab].

2.1.1 Robot Operating System

The boom in robotics in the last decade led to the development of a new generation of robotics software. One of the most important contributions in this area is the 2007 development of the Robot Operating System (ROS) by Willow Garage [17]. As seen in Fig. 2.1, ROS is a framework for writing robotics software that can be used across almost every robotic platform available [18], and that helps researchers and developers to create software that is modular, concurrent, open-source, and supports code reuse. ROS has been widely accepted; the number of individuals, research teams, companies, and projects using ROS has grown exponentially [19]. These communities have worked together to collaboratively develop general-use libraries for navigation, point clouds, and more that have proved invaluable and timesaving in robotics research.

In 2012, Crick *et al.* introduced the ROS package Rosbridge, which helps ROS-based robots to interact with non-ROS systems [20]. Specifically, Rosbridge provides a mechanism within which messages generated by ROS systems are exposed to non-ROS based agents in JSON format using the WebSockets [21] API.

2.1.2 Cloud Computing

Cloud computing is a paradigm for hosting applications on remote infrastructure [13]. More precisely, a cloud computing platform is a parallel and distributed system that consists of inter-connected virtual computers which are dynamically represented as one (or more) unified computer resource based on a service-level agreement with an end user [22]. Cloud computing is service-driven, i.e. resources are provided as a service-on-demand. In practice, the services provided by the cloud can be categorized into three major types (Fig. 2.2): Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS).

- **Software as a service (SaaS)**: SaaS is used to provide access to a completely developed application over the Internet. Examples include Dropbox [23] and Salesforce [24].
- **Infrastructure as a service (IaaS)**: IaaS refers to the on-demand provisioning of bare infrastructural resources. The consumer is usually provided with fundamental computing resources on which they can deploy arbitrary software.
- **Platform as a service (PaaS)**: PaaS falls between IaaS and SaaS, where cloud service providers provide consumers with tools like a programming language and libraries that help the consumer to develop applications easily.
- **Robotics and Automation as a Service (RAaaS)**: This is the robotic equivalent of SaaS, in which software is stored on the cloud server and access is provided over the Internet to users and robots [25].

2.2 Cloud Robotics Evaluation Metrics

In this section, we will introduce key evaluation metrics that are used to evaluate the performance of these architectures. The performance of cloud robotics architecture is usually evaluated in terms of the following key metrics:

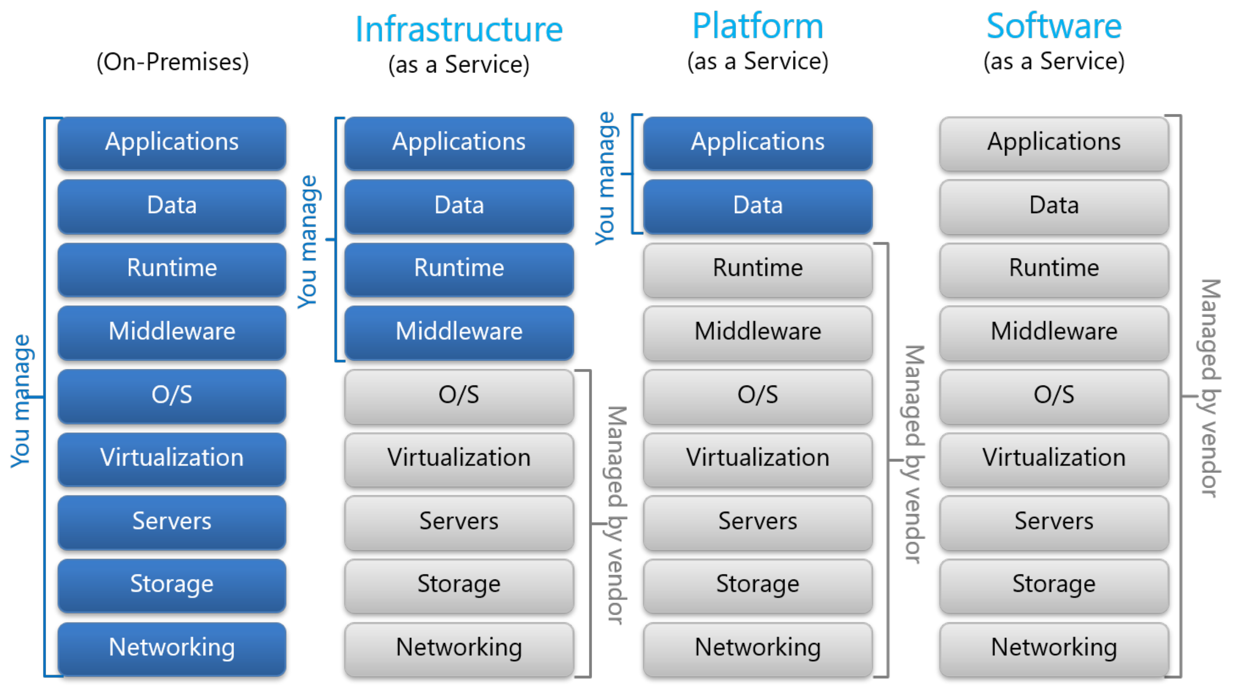


Figure 2.2. Cloud service models [26][© 2018 Chou].

2.2.1 Latency

Latency is used to describe any kind of delay that occurs during data communication over a network, and is an important consideration that researchers cannot ignore [27]. Latency in message exchange can be expensive in applications where robots need real-time feedback [28]. As cloud robotics implementations usually use a network to establish connections between cloud services and robots, network delays can affect robot communication with cloud service providers. Over the last decade, a massive leap has been made in wireless technologies, but factors such as signal loss, network failure, and malware cause network delays that cannot be controlled. Other reasons for latency include data size, network bandwidth, data pre-processing, application processing time in the cloud, and round-trip time.

2.2.2 Scalability

Scalability is an important feature in robotics, and cloud computing can aid robots in becoming more scalable. Through enabling robots to scale their computational resources up or down depending on application requirements, cloud computing makes robots be more dynamic. Cloud robotics also facilities robot crowd-sourcing [29], in which information acquired by one robot can be shared with another to accomplish shared or separate goals. The data can also be stored on the cloud infrastructure to create a meaningful dataset for later use. RoboEarth [30] is a cloud robotics platform that provides crowd-sourcing facilities by allowing robots to upload information to and download from the cloud servers.

As with single robots, multi-robots are restricted by resources, information, and communication constraints, which shortcomings can be overcome through the use of cloud robotics [31]. Use of cloud robotics allows multi-robots to transfer data to the cloud, thereby releasing the computing and storage overhead on individual robots. Cloud technology makes it possible to design multi-robot systems that have real-time performance, are energy efficient, and have low cost [32]. Cloud robotics architecture can especially provide multi-robots with a centralized communication framework that creates a generic level of abstraction for robot hardware, addressing interoperability issues and thus making robot communication easier. Cloud robotics also facilitates the robot network to be more dynamic, where a robot can join the network or become unavailable in an ad-hoc based on the immediate circumstances. Hence, using cloud robotics for multi-robots makes the robotic network more scalable.

2.2.3 Reliability and Availability

In recent years, cloud computing has received significant attention in the context of robotics and other applications. However, the reliability and availability of cloud service providers plays important roles in cloud robotics architecture. It has been reported that outages and failures of cloud services cause a loss of around 285 million dollars on a yearly basis [33]. The reliability and availability of cloud services affect the Quality of Service provided by a cloud robotics architecture [34], which denotes the level of performance, reliability, and availability of the applications hosted using cloud services. New algorithms and methods should be investigated to improve Quality of Service in robotics applications. No comprehensive studies on Quality of Service for cloud robotics had been made until recently

[35]; the importance of Quality of Service in cloud robotics architecture is only beginning to be emphasized.

2.2.4 Interoperability

Robots and sensors are developed and produced by different manufacturers, leading to differences in output data formats. These diversified data formats need to be preprocessed to match the requirements of the cloud computing platform. Similarly, data from cloud services needs to be converted to a format the robots can comprehend. Managing these diversified data formats incurs considerable overhead in cloud robotics architectures.

Multi-robot coordination similarly requires sharing individual data, and suffers from the lack of a standardized format. Cloud robotics can act as middleware and convert data to a given robot-specific format. Developing standard data formats, middleware [36], and SaaS-based [37], [38] cloud robotics architecture can address data interoperability to an extent.

2.2.5 Security

Adopting cloud computing in robotics invites new issues in the form of privacy and security. The information used in cloud robotics, such as maps, video, and images, is not meant to be accessed without permission; however, cloud services provide opportunities for hackers to gain inappropriate and unauthorized access to the data. Cloud services are also prone to breaches in data, data loss, data leakage, malware, and other security issues [39].

Furthermore, the data acquired from robots is not homogeneous. The data from various sensors are represented in the form of distributed databases; implementing security and privacy protocols for these datasets is even more challenging [40]. To address these issues and ensure the security and privacy of robot data, data encryption techniques need to be explored. It is also important to design security measures such that they don't adversely impact the performance of the architecture. Various access control techniques can also be explored to increase the security of the cloud robotics architecture.

2.3 Limitations of the Current Studies

The area of cloud robotics is a relatively new field of study in which research to date is limited. Since its inception, a variety of cloud robotics architectures and applications have been proposed; the proposed architectures share several common challenges. One of the critical aspects of cloud robotics is the decision about what, when, and how to distribute applications between on-board resources and cloud services. Robot hardware comes in different configurations and robotics applications have different requirements, producing a dynamic environment that cannot be managed with a static framework. More specifically, the dynamic nature of robotics means that uploading all computational tasks to the cloud is not an ideal solution. Over the last decade, a massive leap has been made in wireless technologies, but factors such as signal loss, network communication issues, and more cause network delays that cannot be controlled. If an architecture depends completely on cloud services, loss of network or increased latency will greatly impact robot performance. Thus, offloading schemas and algorithms are needed to maximize the utilization of a robot's on-board resources when the situation demands. New methods and algorithms should be designed for handling network latency.

Another limitation arises from the development and production of robots and sensors by different manufacturers. The data formats output from these devices are different from one another, necessitating that uploaded data be preprocessed to match the requirements of the cloud computing platform. Similarly, data from cloud services should be converted to a format the robots comprehend. The use of these diversified data formats and need for data conversion incur considerable overhead in cloud robotics architectures. Developing standard data formats and SaaS-based cloud robotics architectures can address data interoperability to an extent. However, adopting cloud computing for robotics introduces new issues in the form of privacy and security due to the data being hosted on the remote platform; cloud services provide opportunities for hackers to gain inappropriate and unauthorized access to the data. To address these issues, the use of data encryption techniques to ensure security and privacy needs to be explored. It is also important to design security measures such that they do not adversely impact the performance of the architecture.

In summary, to address these limitations, we proposed an architecture that differs from other architectures by being built for use in both SaaS-based and PaaS-based applications. A simple web interface will be provided for SaaS usage, and the code will be made publically available for users to develop applications on top of it in PaaS usage. To address offloading limitations, we introduce two novel dynamic application offloading algorithms in Chapter 4 and Chapter 5. To our knowledge, these are the first of their kinds in the area of cloud robotics. First, we address the offloading problem from Markovian Decision Making and perspectives and solve it using reinforcement learning-based neural networks. Then, we introduce a predictive algorithm that predicts the time and energy required for execution of an application based on a simple regression model

3. ARCHITECTURE AND FULL OFFLOADING

This chapter contains the material from the previously published paper. The material has been added with the consent of the all the authors on the paper.

- “Smart Cloud: Scalable Cloud Robotic Architecture for Web-powered Multi-Robot Applications,” M. Penmetcha, S. Sundar Kannan and B.-C. Min, 2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC), Toronto, ON, Canada, October 11-14, 2020.

3.1 Introduction

In this chapter, we present a new cloud robotic architecture called *Smart Cloud*, outlined in Fig. 3.1. The architecture includes several novel functionalities that make it the first of its kind. Smart Cloud can be used as both SaaS and PaaS depending on the needs of the application. In a SaaS-based approach, it provides a simple web-based interface by which a robot can make use of several ready-to-use applications, both Robot Operating System (ROS)-based and non-ROS-based. The architecture backend is built on a JavaScript (JS) server. Using JS inherently allows access to open-source libraries, including machine learning libraries like TensorFlow, data compression mechanisms to reduce network load, and more [41]. Additionally, we intend to provide the developed framework as an open-source application for other researchers to modify, as researchers frequently need the flexibility to develop or customize applications to suit their specific requirements. By making the framework open-source, the proposed architecture can also be used in a PaaS implementation.

In this chapter, the functionality of the architecture is demonstrated for several application scenarios. Furthermore, the architecture is evaluated in terms of CPU utilization, and latency. The remainder of the chapter is organized as follows: Section 3.2 presents a literature review, while Section 3.3 describes the architecture in detail. Section 3.4 discusses its applications and the evaluation of its performance. Finally, Section 3.5 presents conclusions and future directions.

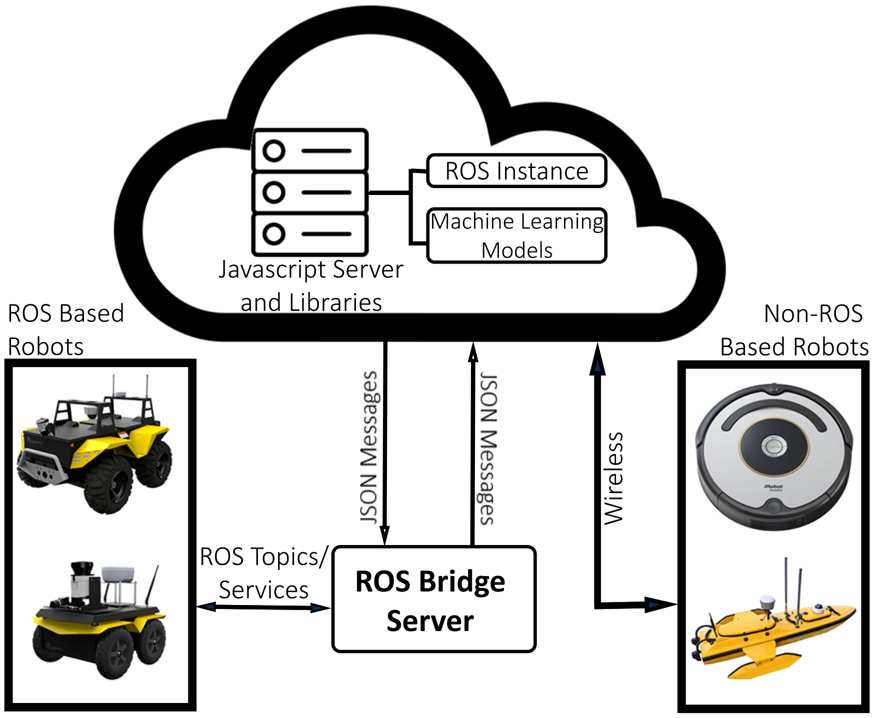


Figure 3.1. Schematic overview of *Smart Cloud* architecture.

3.2 Related Work

In 1997, Masayuki Inaba at the University of Tokyo worked on a so-called “remote brain”; in the associated publication, the authors described advantages of using remote computing for robots [42]. In 2009, the ‘RoboEarth’ project was announced with the intent to develop a World Wide Web equivalent for robots [30]. RoboEarth was developed by a team of researchers from Eindhoven University; the idea was to build a large database that allowed robots to share learned information with one another. The RoboEarth project team created a platform for cloud computing, named Rapyuta [43]. Rapyuta is a PaaS-based robotics framework that allows robots to offload all computation-intense tasks to the cloud service. This framework has access to the RoboEarth data repository, which enables robot access to all extant libraries on RoboEarth.

Interest in cloud robotics has led to the development of new vertical research involving architectures that facilitate robot communication with cloud service providers. One of the first architectures in this area was ‘DaVinci’, which used cloud computing infrastructure to generate 3-D models for robot localization and mapping much faster than possible using on-board hardware [44].

Doriya *et al.* proposed a robot cloud framework that helps low-cost robots offload computationally intense tasks to the cloud [45]. The central unit of the framework is equipped with a ROS master node that facilitates all communication. In 2016, Wang *et al.* proposed a hybrid frame work called RoboCloud [46]. RoboCloud differs from other cloud robotics architectures by introducing a task-specified mission cloud with controllable resources and predictable resources. For tasks beyond the capability of the mission cloud, the framework opts to utilize public clouds, the same as any other cloud robotics architecture.

In addition to the above-mentioned PaaS architectures, re searchers have also worked on SaaS-based frameworks, also called Robot as a Service (RaaS). Notably, SaaS-based architectures can more easily overcome interoperability issues that arise due to differing robot hardware. Frameworks such as C2tam [47] and XBotCloud [48] focus on specific applications or algorithms, for example object recognition or object grasping. Tian *et al.* [49] proposed a RaaS-based robotics model called Brass, which allows robots to access a remote server that hosts a grasp planning technique. The framework leverages Docker to allow for implementing algorithms by writing simple wrappers around existing code.

In summary, most currently-available architectures are geared towards PaaS models; few of them take a SaaS approach. These architectures are designed to offload robotic applications to the cloud infrastructure. The proposed framework differs from extant frameworks by being built for use in both SaaS-based and PaaS-based applications. A simple web interface will be provided for SaaS usage, and the code will be made publically available for users to develop applications on top of it in PaaS usage. Furthermore, the proposed framework is the first of its kind to provide access to open-source JS libraries; users are not limited to available robotic libraries and do not need to develop applications from scratch, but can make use of available JS libraries for robotic applications.

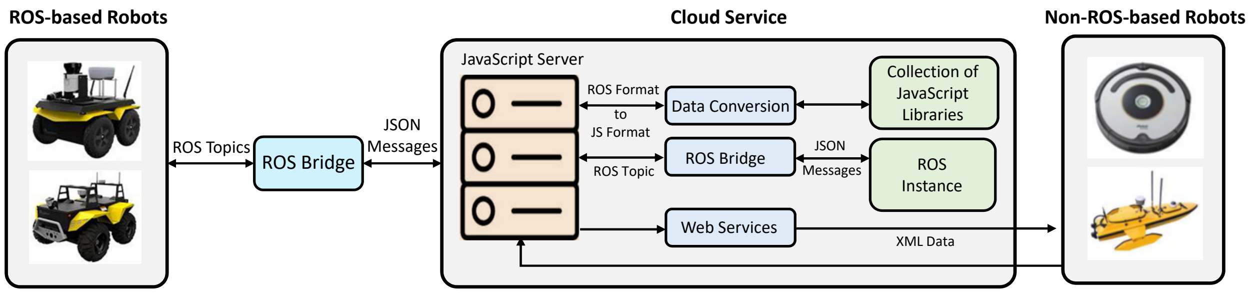


Figure 3.2. Smart cloud architecture; (Middle) Cloud server (Cloud layer): The JS server interacts with JS libraries and ROS, (Left) ROS-based robots (Robot layer with ROS robots): Robots interact with the server using the ROSbridge protocol, and (Right) Non-ROS-based robots (Robot layer with non-ROS robots): Robots send information through wireless data transfer and receive responses as web services.

3.3 Architecture

As shown in Fig. 3.2, the Smart Cloud architecture consists of two main components: the Robot layer (ROS-based robots and non-ROS-based robots) and the Cloud Service layer. We chose JS for the development of this framework because of its ubiquitous nature, support for ROS, and the vast availability of libraries. In terms of available open-source libraries, JS outnumbers every other programming language [41]. Additionally, the JS library Roslibjs allows interaction with the ROS interface through Rosbridge, which is developed for non-ROS users and used to send and receive data in the form of JavaScript Object Notation (JSON) packets. Roslibjs supports essential ROS functionalities such as publishing and subscribing to topics, services, actionlib, and more.

3.3.1 Robot Layer

The robot layer consists of a single robot or a multi-robot system that runs on either ROS or any generic robot software. In the next subsection, we discuss how the architecture handles data based on the robot's software and the type of application service requested.

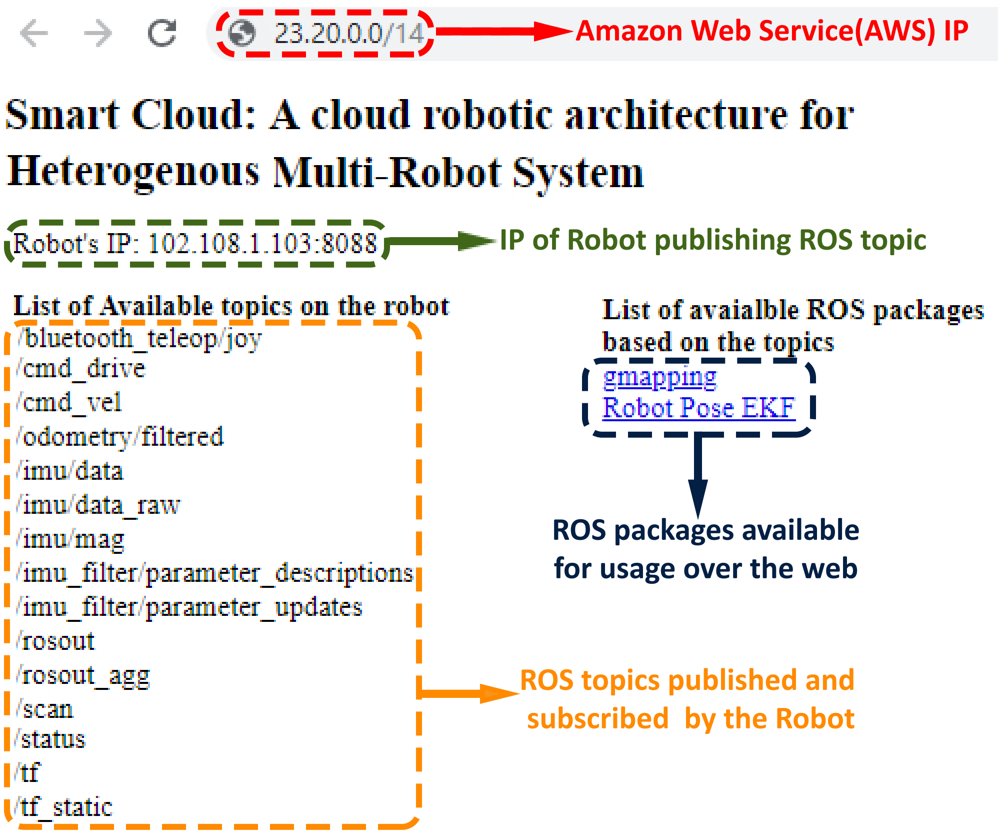
ROS Based Robots

ROS is a framework for writing robotic software. The Rosbridge package allows a ROS-based robot to interact with any non-ROS system [50] through web sockets. In this architecture, we use Rosbridge to establish a communication protocol between robots and cloud services. On the cloud side of the architecture, a master instance of ROS helps the robots to offload computationally intense tasks to the cloud.

Through this framework, robots can access ROS packages and JS libraries; however, to use ROS packages on the cloud, the robots themselves need to run on ROS. Data from the robot is published as ROS topics and received on the cloud side using Rosbridge with the following code:

```
function rosTopics()
{
    var topicsClient = new ROSLIB.Service({
    ros : ros,
    name : '/rosapi/topics',
    serviceType : 'rosapi/Topics'
    });
}
```

Almost every ROS package takes topics from the robot as input. Once the cloud framework receives this list of topics, it parses through the list of available ROS packages to find those that can be used with the given input. The list of matching ROS packages is displayed on the web interface for the user to choose between. After the user picks a package, the result is computed and sent back to the robot over the Rosbridge. For example, the ROS gmapping package requires *tf* and *scan* topics. If the framework finds *tf* and *scan* topics available, the user will be provided with the option to use gmapping. Fig. 3.3 shows the web interface displayed to a user.


Smart Cloud: A cloud robotic architecture for Heterogenous Multi-Robot System

Robot's IP: 102.108.1.103:8088

List of Available topics on the robot

/bluetooth_teleop/joy
/cmd_drive
/cmd_vel
/odometry/filtered
/imu/data
/imu/data_raw
/imu/mag
/imu_filter/parameter_descriptions
/imu_filter/parameter_updates
/rosout
/rosout_agg
/scan
/status
/tf
/tf_static

List of avaialble ROS packages based on the topics

gmapping
Robot Pose EKF

23.20.0.0/14
Amazon Web Service(AWS) IP
IP of Robot publishing ROS topic
ROS packages available for usage over the web
ROS topics published and subscribed by the Robot

Figure 3.3. Smart cloud: SAAS-based ROS interface. The robot provides a list of topics and the architecture displays corresponding packages that can be used.

Non-ROS Based Robots

The data from robots not using ROS will be transferred directly to the cloud using wireless protocols. Once the data is received, the architecture provides a set of JS-based libraries to choose from based on the message type. For example, if the message is in the form of an image, the framework provides libraries related to object detection, object tracking, and the like. If the message is in the format of GPS coordinates, the framework provides various GPS-based applications.

On the server side, the framework implements a RESTful-based web service that is used to communicate the results back to the robot. Web service is a consistent medium for communication between the client and the server over the internet. The communication between the client and the server is carried out through Extensible Markup Language (XML).

In our architecture, we treat the robots as a client and the architecture as a server. The robot sends a HTTP request to the architecture and the architecture sends the response in the form of an XML.

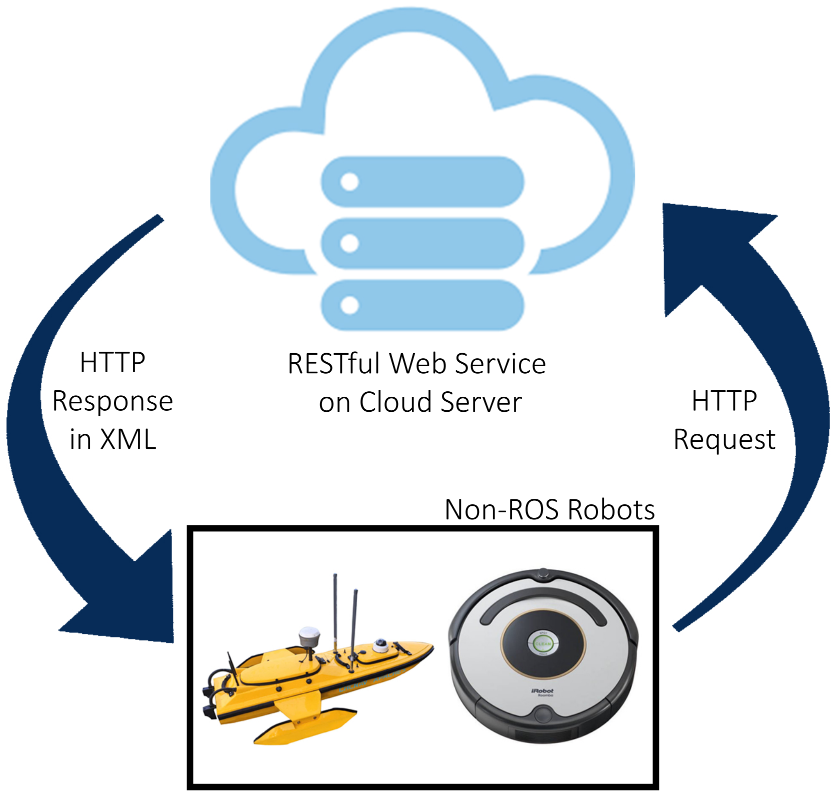


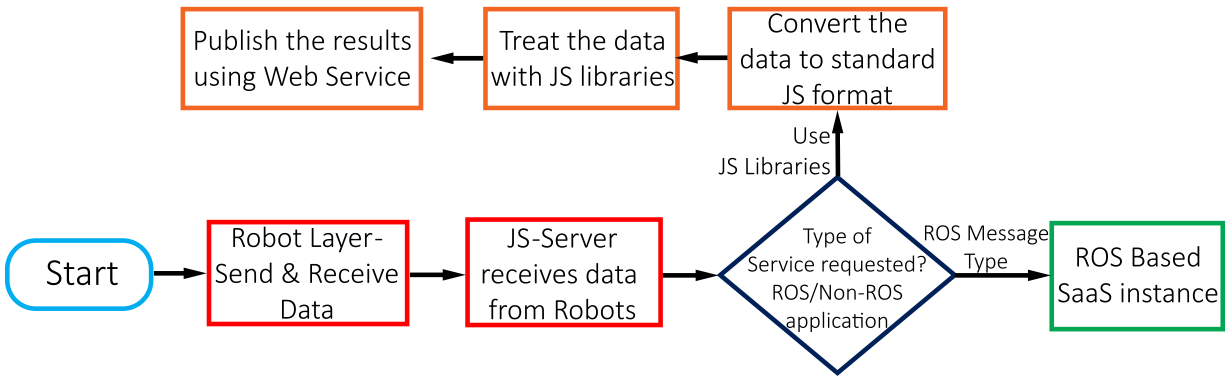
Figure 3.4. General flow of information between robot and cloud through the RESTful web service.

3.3.2 Cloud Layer

Robots are connected to the JS server on the remote cloud infrastructure through web sockets. Depending on application requirements, the cloud layer runs a single or multiple instances of Linux-based operating systems. On these instances, JS Server and ROS are deployed. In this section, we will look into how data is received and processed by the JS server and how various libraries are used for robotic applications.

JavaScript Server

In this architecture, we use a JS server based on Node.js [51]. Node.js is an open-source and cross-platform JS server that runs JS scripts outside a browser. The framework has different mechanisms for handling data from ROS and non-ROS systems.



Publish the results using Web Service
Treat the data with JS libraries
Convert the data to standard JS format
Use JS Libraries
Start
Robot Layer- Send & Receive Data
JS-Server receives data from Robots
Type of Service requested? ROS/Non-ROS application
ROS Message Type
ROS Based SaaS instance

Figure 3.5. Flow chart illustrating the data flow between the robot and Cloud.

For ROS-based data, the user can choose applications from ROS packages or JS libraries. If the user chooses a JS-based library, the framework decodes the data provided by the robot using appropriate JS libraries. We use inherent JS programming tools like *DataType.type*() to classify the type of data and then decode it into an appropriate form. For ROS image formats, we use the *Canvas* library to the convert it to Base64 image format. Figure. 3.5 illustrates the data flow on the cloud.

JavaScript Libraries

For the past five years, JS has consistently ranked among the top ten programming languages. It is used everywhere: in web browsers, mobile applications, games, the Internet of Things, robotics, and more. Due to the high usage of JS, the ecosystem around it is growing at a fast pace. Node Package Manager (NPM) is a popular package manager for JS, with more than 35,000 open-source packages. We intend to use some of this vast set of publically-available libraries in our proposed framework, including the popular machine learning library TensorFlow, which has recently been introduced to the JS ecosystem. To demonstrate the working of the architecture, we used TensorFlow.js for object recognition.

3.4 Application and Evaluation

We conducted an experiment to demonstrate the working of the architecture and also to evaluate various metrics. This experiment used two robots, a Clearpath Jackal Unmanned ground vehicle (UGV) and an iRobot Roomba powered by ODROID-XU4. The hardware configurations of the devices are summarized in Table 3.1.

Table 3.1. Hardware configuration of the robots

	Jackal UGV	ODROID-XU4
CPU	Intel(R) Celeron(R) CPU G1840 @ 2.80 GHz	Samsung Exynos5422 A7 Octa-core CPUs
RAM	2 GB	2 GB

3.4.1 Offloading ROS Applications to the Cloud

To validate the proposed architecture, a computationally intense application is an ideal choice. One such application is gmapping, it continuously takes various sensor inputs such as transforms and laser-scans, and it provides output in the form of a map and entropy. The gmapping package is an ROS wrapper for Openslam's gmapping [52] that is used to create a map, while the robot navigates through the environment. The gmapping package provides Simultaneous Localization and Mapping (SLAM) using the laser input provided by the robot. The Jackal robot used for the experiment was equipped with Velodyne laser scanner, and the input from this laser scanner was used to generate the map of the environment.

For this experiment, we implemented the ROS package gmapping over the cloud and also on the robot. The architecture subscribed to the topics from the robot and executed the gmapping application on the cloud. We evaluated the performance impact using two metrics: CPU utilization and latency.

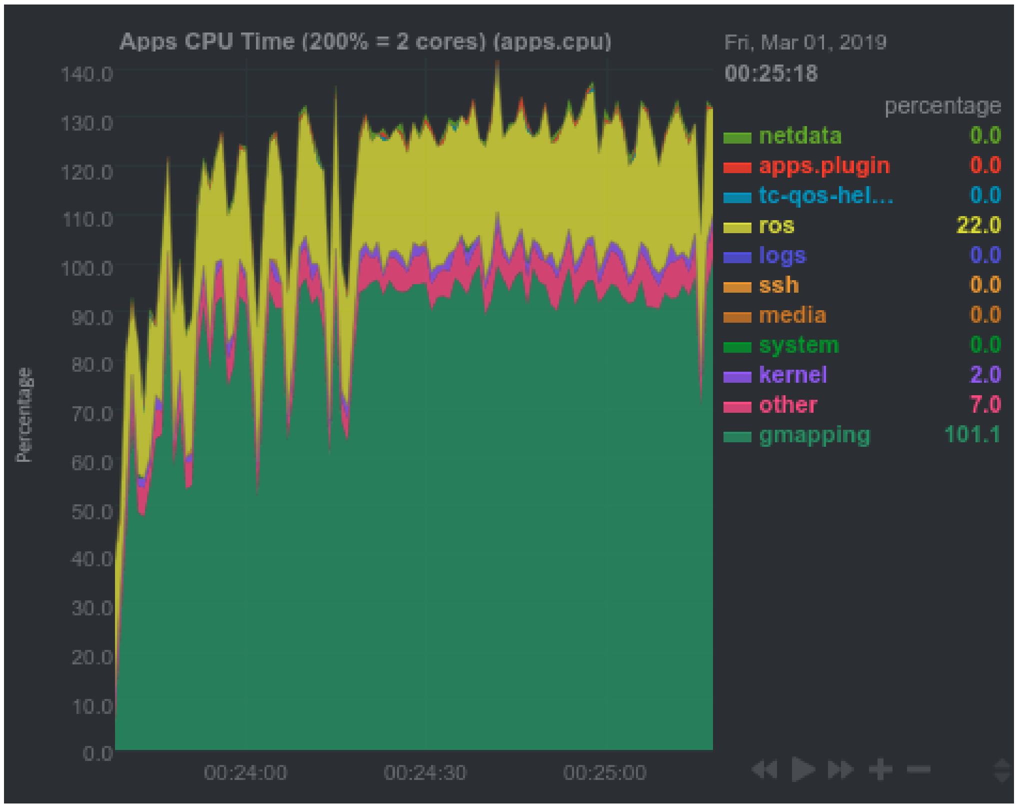


Figure 3.6. CPU utilization with gmapping deployed on the robot (Jackal). On balance, gmapping fully consumes one processor core.

CPU Utilization

Execution of the ROS gmapping package onboard the Jackal was compared against the cloud architecture to observe differences in CPU utilization. The Jackal robot has two cores CPU and each core has a capacity of 100 percent; hence, the total CPU capacity is represented as 200 percent. CPU utilization was measured using the open-source tool netdata [53], which is a real-time performance monitoring tool for Linux-based systems. When offloading applications, the architecture subscribed to the topics published by the robot and executed all back-end computation related to ROS packages on the cloud. By using the architecture, no load associated with ROS packages was incurred to the onboard CPU, hence

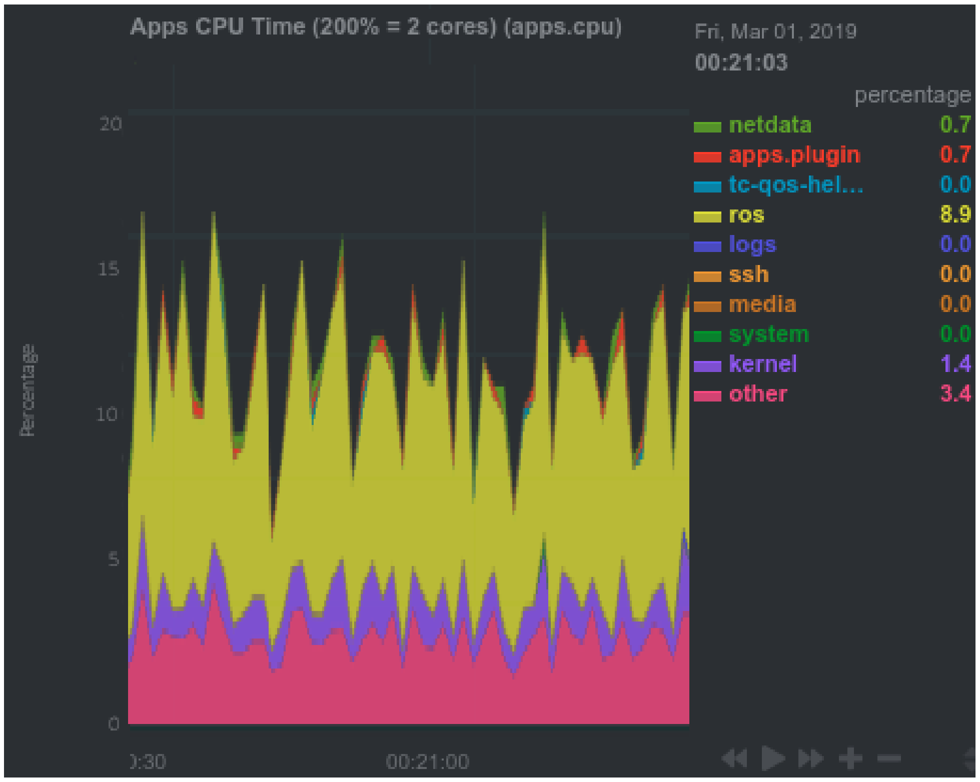


Figure 3.7. CPU utilization with gmapping deployed on the cloud. Demand on the robot CPU is significantly decreased.

we observed a dramatic decrease in CPU utilization. We were able to demonstrate a decrease in CPU utilization by offloading gmapping computation on to the proposed architecture, CPU utilization of the robot decreased by an average of ten-fold compared to running the package on the robot. Fig. 3.6 and Fig. 3.7 show the difference in the CPU utilization on the robot. The applications that were mainly responsible for the reduction of CPU usage are ROS-Master and gmapping. The ROS-Master consumed an additional 20 to 25 percent of CPU with gmapping running on the robot, whereas with the proposed architecture, the ROS-Master consumed less than 10 percent of CPU. The gmapping application was a computationally intense application for robot navigation. When gmapping was deployed on the robot, we observed an average CPU consumption of 85 percent by the gmapping application, whereas with the architecture there was zero associated computational load on the robot CPU.

Latency

Latency is the term used to describe any kind of delay that occurs during data communication over a network. As the Smart Cloud architecture requires an exchange of information between a robot and a cloud service provider, some latency exists between robot requests and cloud service responses. For this experiment, we used an Amazon Web Services (AWS) server located in North Virginia.

To measure the time delay between robot requests and cloud service responses, we implemented a ROS service to exchange information and recorded the message timestamps. On average, we observed a time delay of around 35 milliseconds, of which an average of 32 milliseconds was associated with AWS data round trip time. Thus, the time delay contributed by the framework was approximately three milliseconds for processing the application on the cloud.

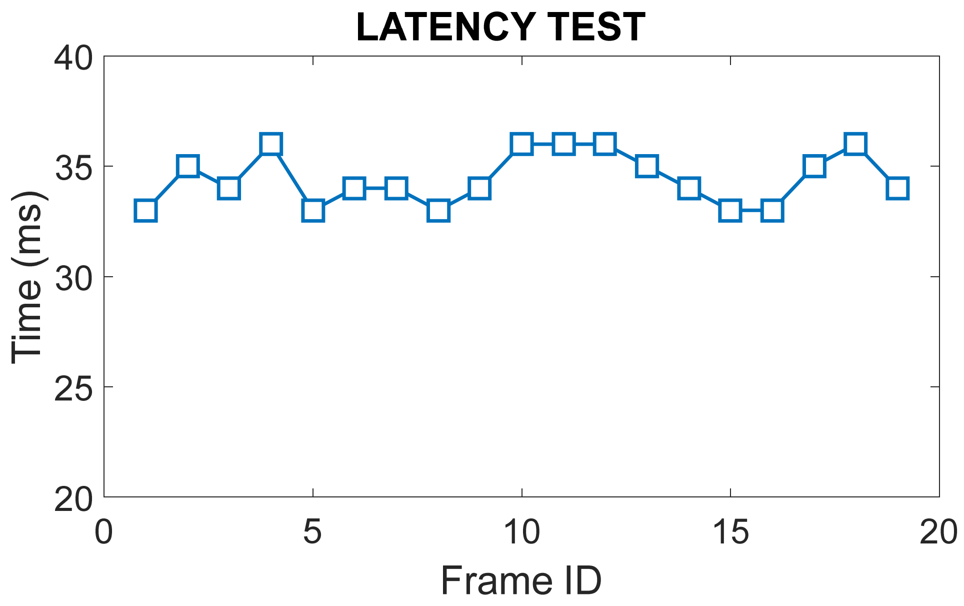


Figure 3.8. Round trip delay for information exchange between the robot and the cloud server.

3.4.2 Object Detection Using TensorFlow JS Library With Odroid (Non-ROS)

In this experiment, we streamed a video from a Roomba equipped with an ODROID-XU4 computer to the architecture. We used Aruco markers along with OpenCV for Roomba to follow Jackal robot. The architecture then identified objects in the video stream using the TensorFlow.js library with an ImageNet dataset and generated XML-based web services to report the results (Fig. 3.9). The following XML file is the web service output generated for Fig. 3.9:

```
<?xml version="1.0"?>
<Response>
  <Message>
      <MessageID>1</MessageID>
      <ReferenceID></ReferenceID>
      <Result>
          <Class>Trash Can</Class>
           <Probability>0.66</Probability>
      </Result>
      <Result>
         <Class>Swivel Chair</Class>
         <Probability>0.72</Probability>
      </Result>
      <Result>
         <Class>File Cabinet</Class>
         <Probability>0.44</Probability>
      </Result>
  </Message>
</Response>
```

Between the robot publishing a frame and the resultant web service feedback, we observed an average time delay of 34 milliseconds. The same functionality can be implemented for ROS Image messages by converting the image to base64 format. The following lines of code can be used to convert a ROS Image to JPEG format:

```
var imgResponse = new Image();
var byteCharacters = atob(message.data);
var abc = "data:image/jpeg;base64,"+byteCharacters;
imgResponse.src = abc;
```

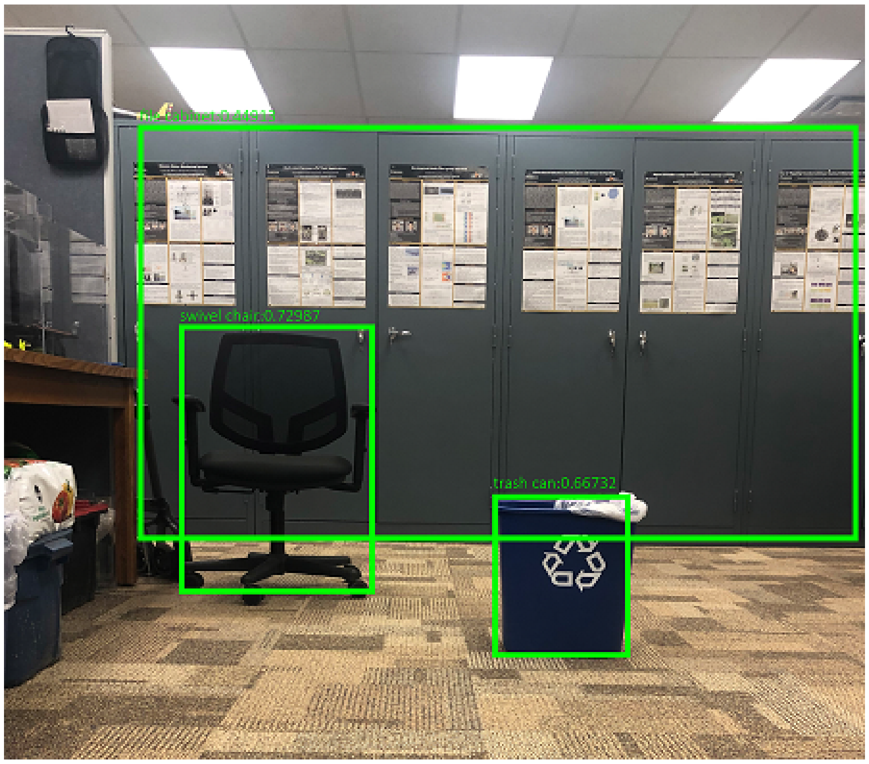


Figure 3.9. Object (i.e. cabinet, swivel chair, and trash can) detected using TensorFlowJS.

3.4.3 Application Scenario Using a Heterogeneous Multi-Robot

In this section, we demonstrated a scenario applying a heterogeneous multi-robot system to a collaborative search and rescue operation that was implemented using the proposed architecture. Figure. 3.10 illustrates this heterogeneous system setup.

The multi-robot system consisted of a non-ROS-based iRobot Roomba equipped with a Microsoft Kinect Camera and a ROS-based Clearpath Jackal equipped with Velodyne 3D

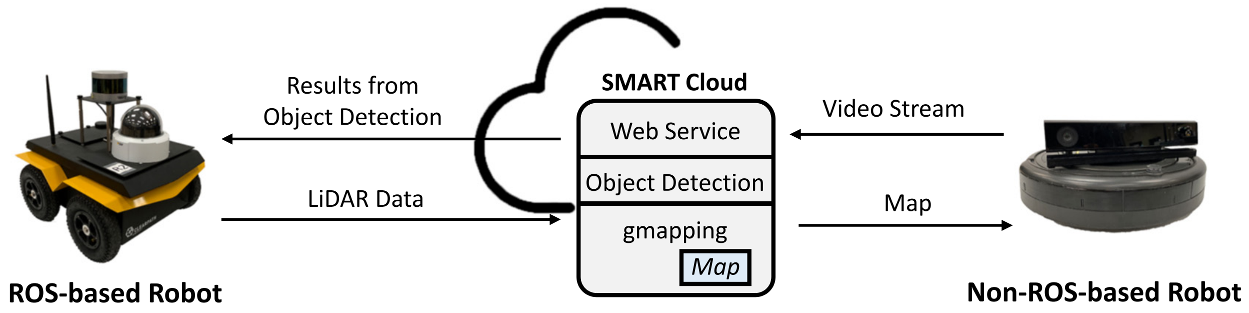


Figure 3.10. Heterogeneous multi-robot setup.

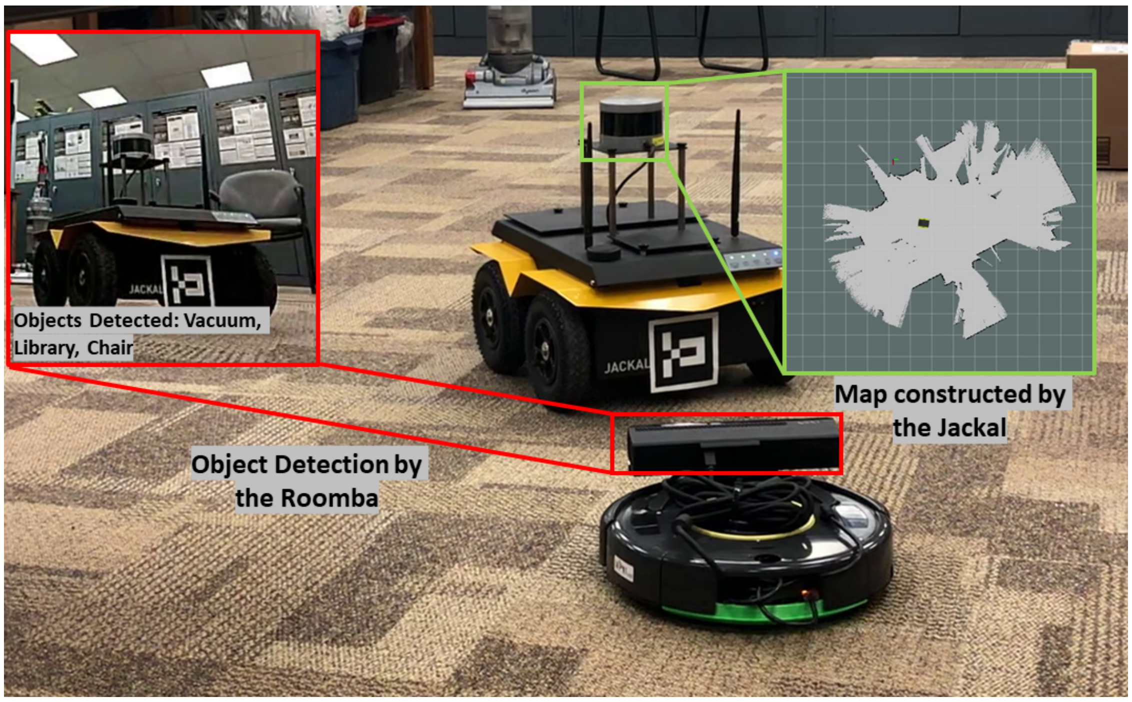

Objects Detected: Vacuum, Library, Chair
Object Detection by the Roomba
Map constructed by the Jackal
JACKAL

Figure 3.11. Heterogeneous multi-robot application with jackal and roomba. Jackal generates a map of the environment and roomba detects objects in the environment.

LiDAR. The goal of the search and rescue operation was to generate a map and find an object of interest within the map. Figure. 3.11 shows the heterogeneous multi-robot collaboration between the Roomba and the Jackal. The Jackal robot used the LiDAR data and generated a map of the environment. The Roomba was responsible for object detection in the generated map. The data from the Microsoft Kinect camera on the Roomba was continuously streamed to the cloud service using wireless protocol, and TensorFlow.js was used to detect objects in the video stream. The object detection results were continuously streamed back using the web service. Meanwhile, the ROS-based Jackal subscribed to the web service results and generated the map until the Roomba found the object of interest.

Therefore, we successfully demonstrated the bi-directional subscription through web services and the other novel architecture features through this experiment.

3.5 Conclusion

In the chapter, we present the *Smart Cloud* architecture, which is the first of its kind to incorporate JavaScript-based libraries for running diverse robotic applications related to machine learning and more. *Smart Cloud* also leverages the resources provided by cloud service providers for use with robotic applications. The architecture can be used with heterogeneous and homogeneous multi-robot systems as well as single-robot systems. We additionally demonstrated the working of ROS and non-ROS based robot systems with the architecture and the incorporation of JS libraries for robotic applications. We measured the performance of the architecture in terms of onboard CPU usage, and latency. We were able to show significant reduction in onboard CPU usage and achieved an average latency of 35 milliseconds.

4. A DRL-BASED DYNAMIC APPLICATION OFFLOADING METHOD

This chapter contains the material from the previously published paper. The material has been added with the consent of the all the authors on the paper.

- "A Deep Reinforcement Learning-based Dynamic Computational Offloading Method for Cloud Robotics," M. Penmetcha and B.-C. Min, IEEE Access.

Robots come with a variety of computing capabilities, and running computationally-intense applications on robots is sometimes challenging on account of limited onboard computing, storage, and power capabilities. Meanwhile, cloud computing provides on-demand computing capabilities, and thus combining robots with cloud computing can overcome the resource constraints robots face. The key to effectively offloading tasks is an application solution that does not underutilize the robot's own computational capabilities and makes decisions based on crucial cost parameters such as latency and CPU availability. In this chapter, we formulate the application offloading problem as a Markovian decision process and propose a deep reinforcement learning-based deep Q-network (DQN) approach. The state-space is formulated with the assumption that input data size directly impacts application execution time. The proposed algorithm is designed as a continuous task problem with discrete action space; i.e., we apply a choice of action at each time step and use the corresponding outcome to train the DQN to acquire the maximum rewards possible. To validate the proposed algorithm, we designed and implemented a robot navigation testbed. The results demonstrated that for the given state-space values, the proposed algorithm learned to take appropriate actions to reduce application latency and also learned a policy that takes actions based on input data size.

4.1 Introduction

Cloud robotics algorithms are designed on the basic premise that when robots have insufficient computational resources for local execution of an application, using cloud resources should improve the performance of the application in terms of execution time and energy efficiency. If a robot has the bare minimum of computational capability, full application offloading will be an obvious choice; however, many robots presently being produced are computationally capable, and so dynamic computational offloading algorithms are a wiser choice as they take into account both the robot's computational capability and the application's computational requirements.

Robots are equipped with a wide range of sensors. Usually, a robotic application gets input from these sensors and processes that input to provide an output action for the robot [54]. The amount of sensory data that the application needs to process at a given time significantly affects its computational consumption [55]; if the application requires more computational resources than the robot can accommodate, its onboard execution might be extended to a degree that degrades the robot's performance. Hence, we designed the offloading problem to consider application input data size and used a deep reinforcement learning (DRL)-based deep Q-network (DQN) for dynamic application offloading. Deep reinforcement learning [56] is a subfield in machine learning that combines neural networks with reinforcement learning (RL), and has opened up avenues for solving problems that were difficult to solve otherwise [57]. DRL has been applied in a wide range of robotic applications related to navigation, social robotics, motion control, manipulation, and more [57]–[59]. DRL-based algorithms have been studied for application offloading in mobile devices [60]–[62], but only a few studies have used DRL for application offloading from a robotics perspective [34]. Most extant offloading algorithms for mobile devices use mobile edge computing (MEC), and by design cater to mobile-specific applications. To our knowledge, we are the first to provide a DRL-based dynamic application offloading solution for cloud robotics that considers input data size and is designed as a continuous task problem with discrete action space, i.e., we apply a choice of action at each time step t and use the corresponding outcome to train the DQN and learn a policy to acquire maximum rewards at a given time step t'. We validated the algorithm on a robotic path planning application running on the Robot Operating System (ROS).

The area of dynamic application offloading for cloud robotics is still in the early stages. In this chapter, we propose a novel dynamic application offloading algorithm for cloud robotics; moreover, the proposed architecture includes several novel functionalities that make it the first of its kind.

The remainder of the chapter is organized as follows: In Section 4.2, we describe related work on dynamic application offloading and DQN fundamentals. In Section 4.3, we define the problem formulation based on DQN. In Section 4.4, we introduce the algorithm's application with a robot navigation framework. In Section 4.5, we provide experiment results and their analysis. Finally, we conclude our work and present our future directions in Section 4.6.

4.2 Background

This section presents related work concerning application offloading in cloud robotics and also introduces the basic DQN fundamentals that our algorithm uses as a design foundation.

4.2.1 Related Work

From 2009 on, several architectures have been proposed that facilitate application offloading for cloud robotics [30], [42]–[47], [49], [63], [64]. The aforementioned works focus on catering to application-specific solutions like navigation, object detection, etc. Most of the architectures propose full offloading without any consideration for the robot's local computing capabilities and the costs associated with offloading. Namely, communication between robots and cloud services, including for complete offloading, has to consider costs such as latency, energy, CPU utilization, and security.

To our knowledge, there are only a few studies in cloudrobotics that have focused on dynamic application offloading that account for the cost parameters involved when making an offloading decision [65]. One prior study based its offloading decision on network connectivity and robot mobility; it used a genetic algorithm and concluded that motion- and connectivity-aware offloading leads to more efficient performance in terms of Quality of Service (QoS) and minimum resource consumption [66]. In 2017, Wang *et al.* [67] proposed a resource allocation strategy based on a hierarchical auction mechanism, namely a link quality matrix (LQM) auction; the algorithm was designed and demonstrated for firm real-time applications and outperformed other state-of-the-art algorithms. Later, Hong *et al.* [68] proposed a QoS-aware cooperative computational offloading solution for robot swarms to minimize latency and maximize energy efficiency; they formulated the optimization problem as a multi-hop cooperative computation-offloading game.

Some other proposed offloading solutions were based on edge computing [69]–[73]. Shuja *et al.*[74], [75] have comprehensively surveyed machine learning-based approaches for in-network caching in edge networks for mobile devices. These solutions allow robots to offload computationally-intense applications to the computing infrastructure in their vicinity. However, none of these papers considered deep reinforcement learning-based techniques for decision–making regarding offloading.

Application offloading for mobile devices is a well-studied area, and several such studies have proposed dynamic application offloading solutions using DRL. As a case in point, Qiu *et al.*[76] proposed a collective and distributed DRL algorithm that considered experience from distributed systems to obtain an optimum offloading policy using MEC. Later, Qiu *et al.*[77] also proposed a DRL-based offloading solution for computationally-intense mining applications that employed multi-hop multi-user blockchain-empowered MEC. Meanwhile, Tang *et al.*[78] proposed a distributed DRL solution to minimize the long-term cost for non-divisible and delay-sensitive tasks using MEC. Wang *et al.*[79] proposed a meta reinforcement learning-based algorithm that leveraged recurrent neural networks for faster loss convergence, and and represented mobile applications as directed acyclic graphs for validation of the algorithm. Finally, Dai *et al.*[80] proposed a DRL-based computation offloading and resource allocation algorithm to reduce overall energy consumption; it used a multi-user end-edge-cloud orchestrated network. Notably, the aforementioned algorithms were all designed for mobile devices and mobile-specific applications using MEC, whereas our proposed algorithm is designed from the robotics perspective and validated with a robotic application using cloud computing.

Some researchers have applied DRL in application offloading solutions in cloud robotics; for example, Chicachali *et al.* [81] formulated the offloading problem as a sequential decision-making problem and used deep reinforcement learning for object detection applications, and their findings suggest that RL is likely an effective choice for optimizing offloading decision policies. Another prior study proposed a resource allocation scheme based on RL that allowed the cloud to decide whether a request should be accepted and the amount of resources to be allocated to the application; this work also demonstrated better performance of RL algorithms relative to other greedy resource allocation scheme [82]. Finally, Peng *et al.* [83] proposed an online resource scheduling framework based on DQN that implemented a trade-off between energy consumption and task makespan by prioritizing the rewards. However, most of these DRL-based works were only published in the last couple of years, and there remains a lot of room for improvement.

The area of dynamic application offloading is still in its early stages, and most of the work we present here is the first of its kind in cloud robotics. Our proposed algorithm broadly diverges from existing solutions in two important ways. First, we introduce a novel offloading strategy based on DQN; the state space is built with the assumption that the size of the input data for an application directly impacts its execution time. Second, we use a variable reward rather than a fixed reward, which led the algorithm to converge faster and to learn an optimal policy efficiently and quickly.

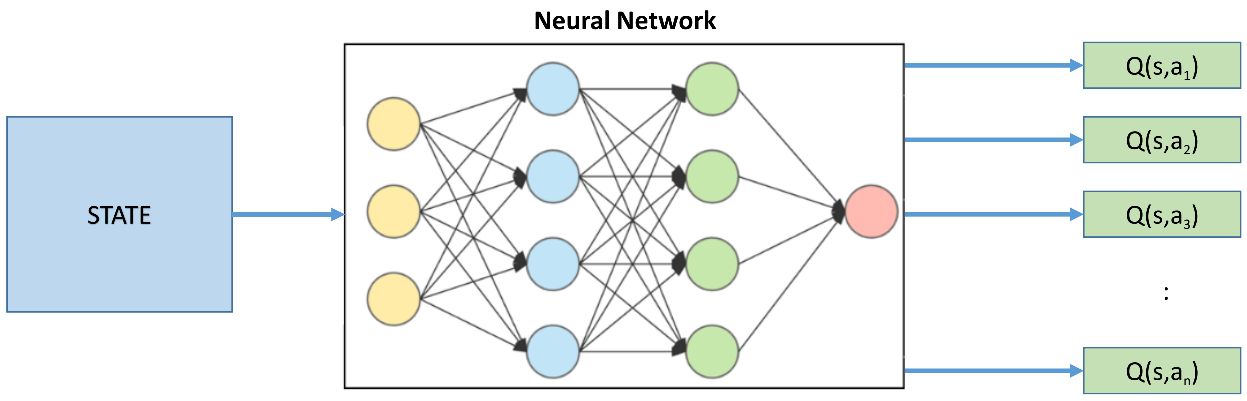


Figure 4.1. Deep Q-Network architecture.

4.2.2 Deep Q Network

Conventional RL algorithms do not scale well with increasing numbers of applications and robots, as this leads to an explosion in state space [84] and becomes an NP-hard problem [85]. DQN is an off-policy, model-free RL algorithm [86] that overcomes several drawbacks faced by traditional RL algorithms [87]–[90]. Using DQN, agents are able to continuously learn and optimize their decision-making through trial and error. DQN models work on the principle of selecting those actions that maximize overall reward in the long term. The agent receives a reward r for change in state s when action a is performed. Observing these rewards, the agent forms a consensus about a policy π that helps it in achieving the maximum reward. Hence, we have modeled the proposed algorithm as a MDP and used DQN to derive an optimal policy π for offloading decision-making.

DQN uses the Q-function [91], [92] as its foundation for calculating expected reward values. $Q(s,a)$ is the reward of the current state-action pair, represented as the summation of the reward for the current state and the maximum reward value expected in the future. Mathematically, $Q(s,a)$ is represented as,

$$Q(s,a) = r(s,a) + \gamma \max_{a'} Q(s',a') \tag{4.1}$$

which can be further generalized using Bellman's equation [93], resulting in the following,

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a) \right] \tag{4.2}$$

where γ represents the discounting rate, α is the learning rate, and the variables s,a,r respectively denote the state space, action taken, and associated reward for a state-action pair. If s and a denote the current state and action, then s' and a' denote the next state and action.

Traditional Q-learning requires a lot of memory to generate a Q-table, and if the application space and number of robots are large, Q-learning also becomes computationally intense. As DQN provides a way to learn a Q-function using a deep neural network, it is a better choice than traditional Q-learning for the application-offloading scenario. As with other machine learning models, DQN has a well-defined cost function that the network tries to minimize. This cost function is given as,

$$L(\theta) = \left[Q(s,a|\theta) - (r(s,a) + \gamma \max_{a'} Q(s',a'|\theta^{-})) \right]^2 \tag{4.3}$$

where θ represents the trainable weights of the network.

We will use the foundations of the DRL and DQN (Fig. 4.1) explained in this subsection for problem formulation. The loss function in Eq. (4.3) is what the neural network tries to minimize using the state space, action space, and reward defined in the next section.

4.3 Problem Formulation

As stated earlier, we modeled the dynamic application offloading algorithm using DQN. DQN is based on MDP, which is usually defined in terms of (s, a, r), where s is the state space, a is the action space, and r is the reward. In this section, we talk about how we defined each of these key parameters for the proposed offloading algorithm.

4.3.1 State Space

The time of execution for an application is proportional to the size of the input data [94]. Hence, we designed the state space to reflect the data sizes of the applications running at any given time, along with other system parameters, namely CPU availability and round-trip time for communicating with the cloud. The state space of the model includes the full observation of the system and is defined for every time step as,

$$s_t = \{d_t, u_t, b_t\} \tag{4.4}$$

where the meaning of each variable at state initialization is as follows:

- $t = 1, 2, 3..., T$ denotes the time steps for the given episode.
- d_t denotes the input data size of application.
- u_t denotes the CPU availability at state initialization.
- b_t denotes the network latency for data making a round trip between the robot and the cloud.

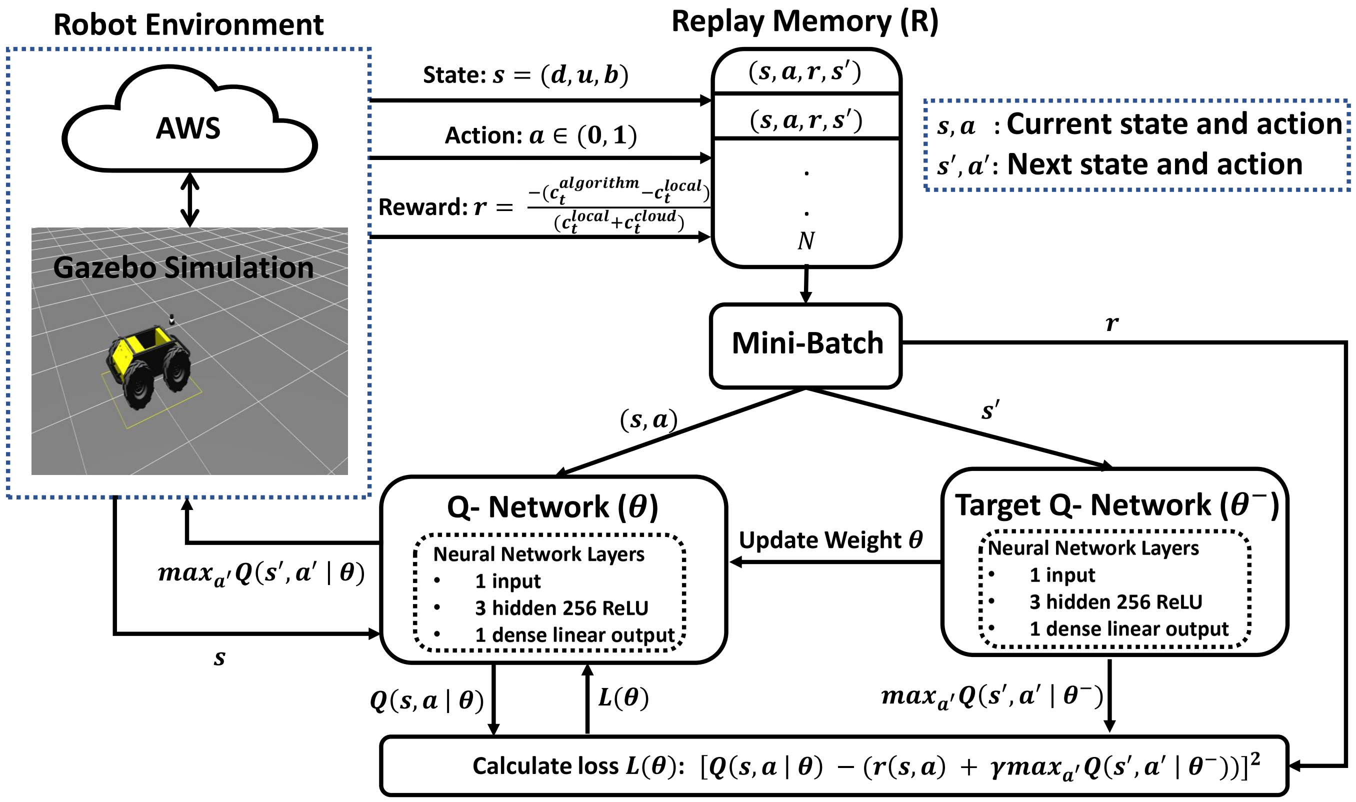


Figure 4.2. Dynamic computational offloading framework based on DQN. The robot environment provides input in the form of state space, action performed, and reward acquired. The DQN learns from these inputs and sends back a response in the form of a Q-value for the state and action pair. The neural network used for the navigation application has one input layer, three hidden ReLU layers with 256 neurons each, and one dense linear output layer.

4.3.2 Action Space

The action space defines what actions an agent can perform in the environment. Our model is designed for binary decision-making; hence our discrete action space is defined as $a_t = (a_0, a_1, ..., a_T) | a_t \in \{0,1\}$. This definition implies that an application can be executed either locally ($a = 0$) or on the cloud service ($a = 1$).

4.3.3 Reward

In the following section, we define variable reward (r_t) and its associated variables for a given state-action pair (s_t, a_t).

When the robot performs on-board computation of a navigation application task, the associated local time of execution (l_t^{local}) is derived from

$$l_t^{local} = l_t^{lcompletion} - l_t^{start}. \tag{4.5}$$

Similarly, when a robot delegates the computation of an application to the cloud, the associated time of execution (l_t^{cloud}) is derived from

$$l_t^{cloud} = l_t^{ccompletion} - l_t^{start} \tag{4.6}$$

where $l_t^{lcompletion}$ represents the timestamp of application completion and l_t^{start} represents the time stamp when the task was first assigned to the robot.

By incorporating tuning parameters α and β, the robotic operator has the opportunity to prioritize between offloading and local computation. These parameters are set according to the relative importance of executing the application on the cloud or on the robot, where $\alpha + \beta = 1$ and $\alpha, \beta \in [0,1]$. When there is no predefined priority between offloading and local computation, $\alpha = \beta$; when offloading is prioritized over local computation, $\alpha > \beta$; and when local computation is prioritized over offloading, $\alpha < \beta$. In our experiment, we do not define a predefined priority, thus always set $\alpha = \beta$.

Combining the tuning parameter α with local execution time (l_t^{local}) gives us total computational cost on the local machine,

$$c_t^{local} = \alpha l_t^{local}. \tag{4.7}$$

Similarly, combining the tuning parameter β with cloud execution time (l_t^{cloud}) gives us total computational cost on the cloud,

$$c_t^{cloud} = \beta l_t^{cloud}. \tag{4.8}$$

During its learning phase, the proposed algorithm randomly chooses between local or cloud execution. This algorithm-determined action $c_t^{algorithm}$ can be either c_t^{local} or c_t^{cloud}, and we can represent the algorithm-determined action as,

$$c_t^{algorithm} = c_t^{local} \vee c_t^{cloud}. \tag{4.9}$$

Then, we define the variable reward r_t for (s_t, a_t) as,

$$r_t = -(c_t^{algorithm} - c_t^{local})/(c_t^{local} + c_t^{cloud}) \tag{4.10}$$

where r_t will always be in the range between -1 and 1. When $c_t^{algorithm} = c_t^{local}$, the reward assigned will be zero, and when $c_t^{algorithm} = c_t^{cloud}$, the reward values will be either positive or negative. The agent is given a positive reward for choosing to offload when cloud execution time (c_t^{cloud}) is less than local execution time (c_t^{local}), and a negative reward for choosing to offload when cloud execution time (c_t^{cloud}) is greater than local execution time (c_t^{local}).

4.3.4 DQN Algorithm for Dynamic Offloading

Now that we have clear definitions of the state space s, action space a, and reward r, we can define our DQN-based offloading algorithm as given in Algorithm 1. The DQN architecture used for validating the proposed algorithm in robot navigation is illustrated in Fig. 4.2. Our problem formulation is based on acquiring maximum rewards and not on the end goal success criterion. As such, the algorithm is designed as a continuous task problem with discrete action space, i.e., we apply a choice of action at each time step t and use the corresponding outcome to train the DQN and learn a policy to acquire maximum rewards at a given time step t' .

Algorithm 1 Proposed DQN-based offloading algorithm

```
1: Initialize replay memory $R$ with capacity $N$;
2: Initialize action-value Q function with random weights $\theta$;
3: Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$ ;
4: **Input**: State space $s_t = \{d_t, u_t, b_t\}$
5: **Output**: Q value for state-action pair
6: **for** episode $= 1$ and $t = 1$ **do**
7:     Receive initial state observation;
8:     **repeat**
9:         Load the state values $s_t$;
10:        Choose a random action $a_t$ from action space $A$;
11:        Calculate reward $r_t(s_t, a_t)$ by Eq. (4.10);
12:        Load the next state values $s_{t+1}$;
13:        Store the experience $(s_t, a_t, r_t, s_{t+1})$ in replay memory $R$;
14:        Select a random minibatch $(s_{\mathrm{j}}, a_{\mathrm{j}}, r_{\mathrm{j}}, s_{\mathrm{j}+1})$ from $R$;
15:        Set $y_{\mathrm{j}} = r_{\mathrm{j}} + \gamma \max_{a_{\mathrm{j}+1}} \hat{Q}(s_{\mathrm{j}+1}, a_{\mathrm{j}+1}|\theta^{-})$ from Eq. (4.1);
16:        Perform a gradient descent to minimize loss using equation $(y_{\mathrm{j}} - Q(s_{\mathrm{j}}, a_{\mathrm{j}}|\theta))^2$
17:          with network parameter $\theta$ from Eq. (4.3);
18:        Every few steps, copy weights from $Q$ to $\hat{Q}$;
19:        Set $t = t + 1$;
20:     **until** A predefined stopping condition is reached, i.e., loss function reached
            convergence;
21: **end for**
```

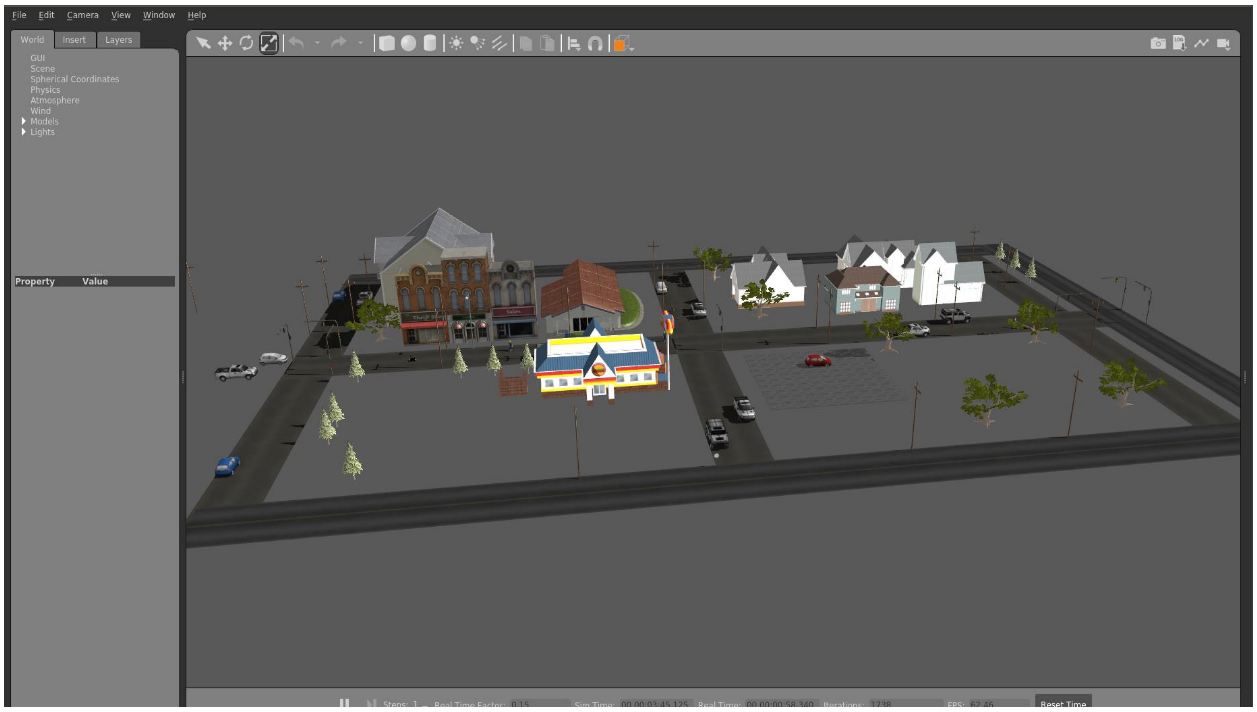
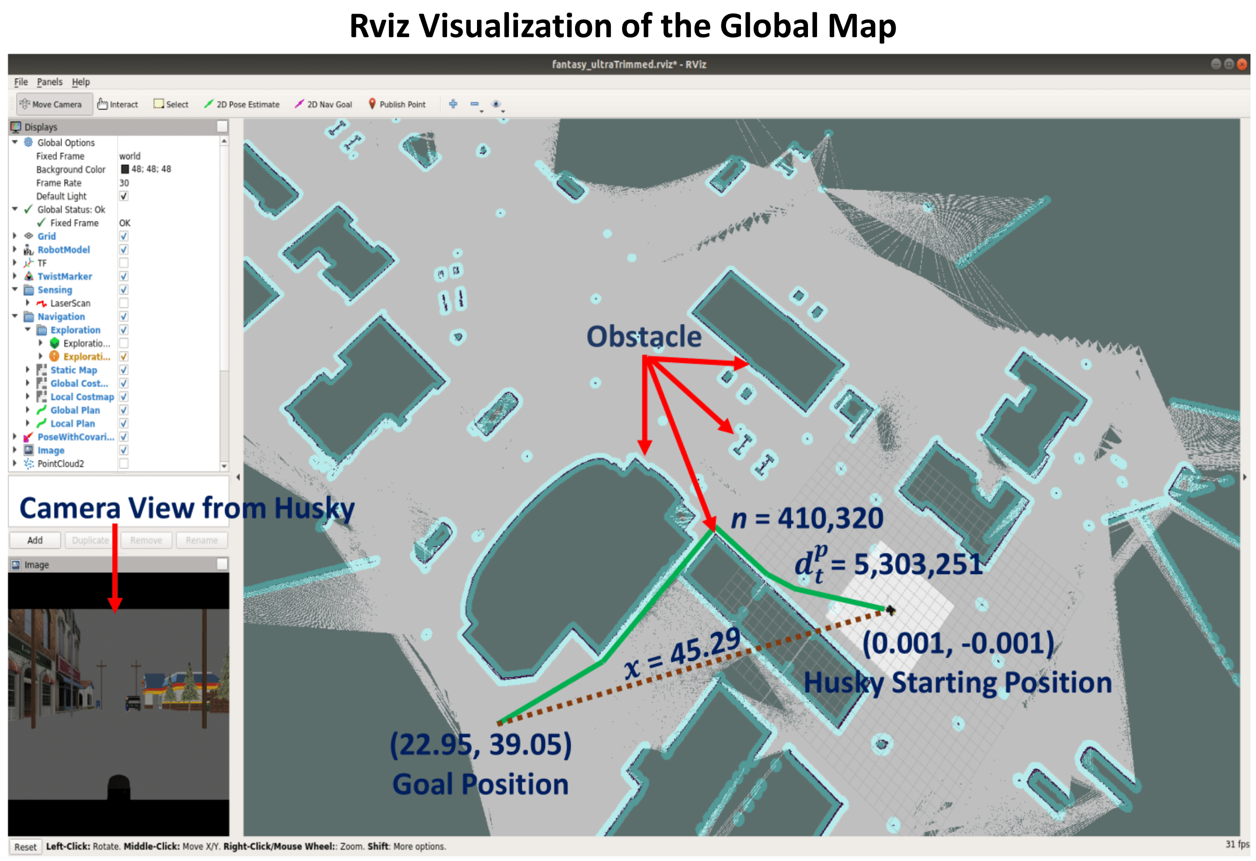


Figure 4.3. The graphical simulation environment that was used to validate the proposed algorithm.

4.4 Experimental Setup With Robot Navigation Application

To validate the proposed algorithm, we used it with a robot navigation application. The graphical representation of the simulation environment is shown in Fig. 4.3. In Fig. 4.4, the global cost map of the navigation environment along with obstacles, the Husky robot, and the camera view from the robot is shown. The simulation environment was constructed using Robot Operating System (ROS) with a gazebo simulator, which we will briefly talk about in the next paragraphs.

The navigation application utilized here is built on top of the ROS framework for writing robotics software, which can be used across most robotic platforms [18] and helps researchers and developers to create software that is modular, concurrent, open-source, and supportive of code reuse. ROS *messages* are structures that contain data of various types, and these messages are transmitted using ROS *nodes*. A brief overview of the ROS application framework is shown in the ROS Layer of Fig. 4.5.



```
Global Options
  Fixed Frame          world
  Background Color     48; 48; 48
  Frame Rate           30
  Default Light        ✓
Global Status: Ok
  Fixed Frame          OK
Grid                   ✓
RobotModel             ✓
TF
TwistMarker            ✓
Sensing                ✓
  LaserScan
Navigation             ✓
  Exploration          ✓
    Exploratio...
    Explorati...       ✓
  Static Map           ✓
  Global Cost...       ✓
  Local Costmap        ✓
  Global Plan          ✓
  Local Plan           ✓
PoseWithCovari...      ✓
Image                  ✓
PointCloud2
```

Figure 4.4. Visualization of gazebo global cost map. The panel shows a global map of the environment with obstacles indicated (Red arrows). In this instance, the husky's starting position is (0.001, -0.001) and the goal position is (22.95, 39.05). Between the starting position and goal, the shortest path is represented by a green line and the euclidean distance is $x = 45.29$. The number of nodes to traverse is $n = 410{,}320$, and the input data size is $d_t^p = 5{,}303{,}251$.

Gazebo is a 3D dynamic simulator that can accurately and efficiently simulate the real-world behaviors of robots, environments, and their interactions [95]. It can be easily integrated with ROS, thereby allowing robots to avail themselves of the hundreds of open-source ROS tools and packages while within the gazebo environment. Finally, we used the open-source Husky robot [96] as the navigation vehicle in the environment.

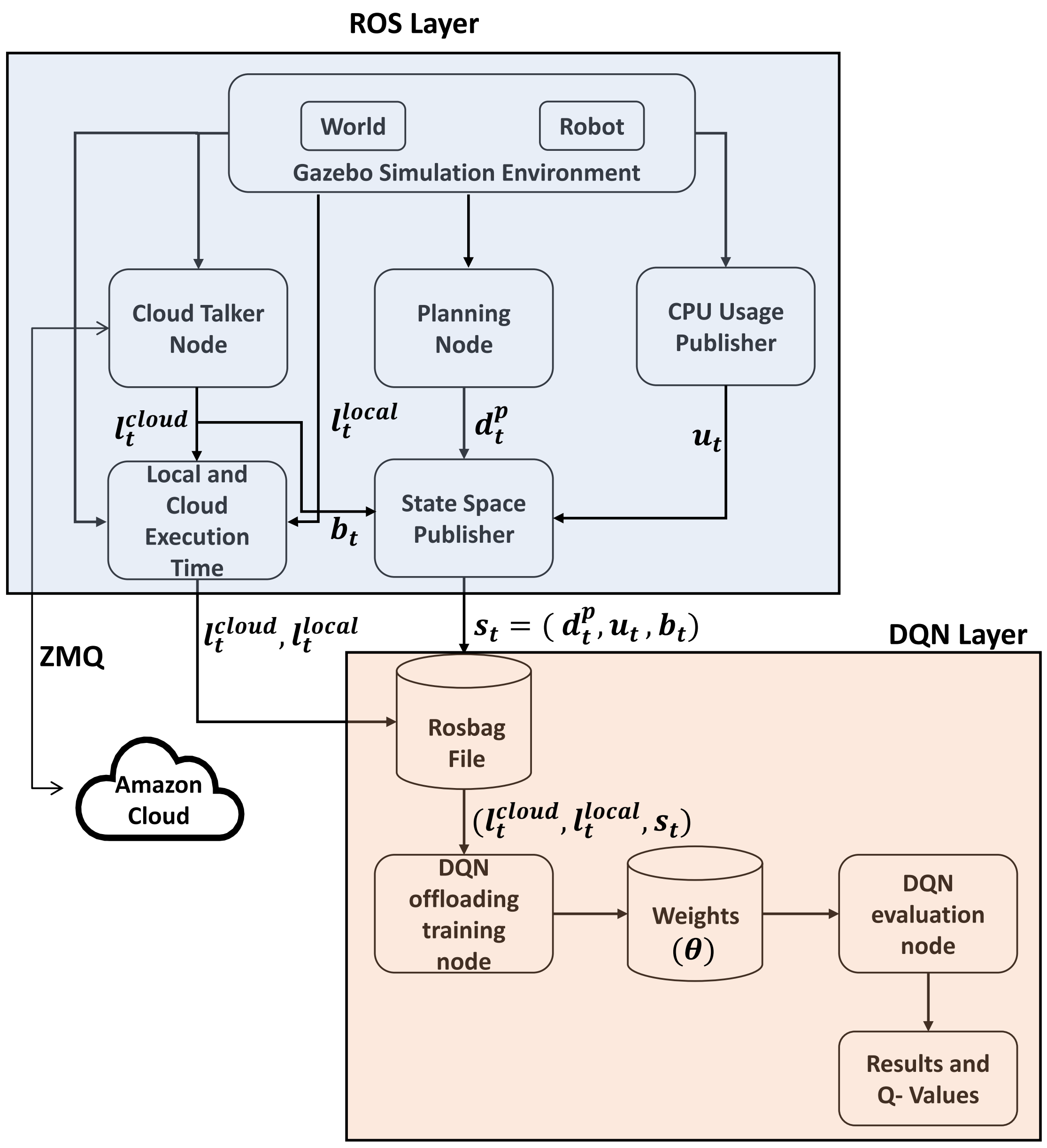


Figure 4.5. Two-layered navigation framework for algorithm validation. ROS layer (Top): A robot interacts with a gazebo world and AWS to generate state-space values along with the local and cloud execution times for path-planning. DQN Layer (Bottom): Derives an optimal offloading policy using inputs from the ROS layer.

4.4.1 Navigation Application Framework

The gazebo simulation environment provides access to a Husky robot, designed by Clearpath Robotics [96], which was equipped with sensors such as a camera, LiDAR, and wheel encoders, and programmed to perform path planning. The main goal of this experiment was to determine if the robot could learn a policy from state space values regarding when to offload the path planning application to the cloud.

The navigation application framework used for validating the proposed algorithm is depicted in Fig. 4.5. The framework can be broadly categorized into two layers: ROS and DQN. In the ROS layer, the robot uses the simulation environment to generate state space data, local path-planning execution time (l_t^{local}), and cloud path-planning execution time (l_t^{cloud}). In every time step, random goal coordinates are assigned to the robot for path planning. For simplicity's sake, the robot does not actually navigate to the destination, but just calculates the shortest path from the origin to the assigned destination using the Dijkstra algorithm [97].

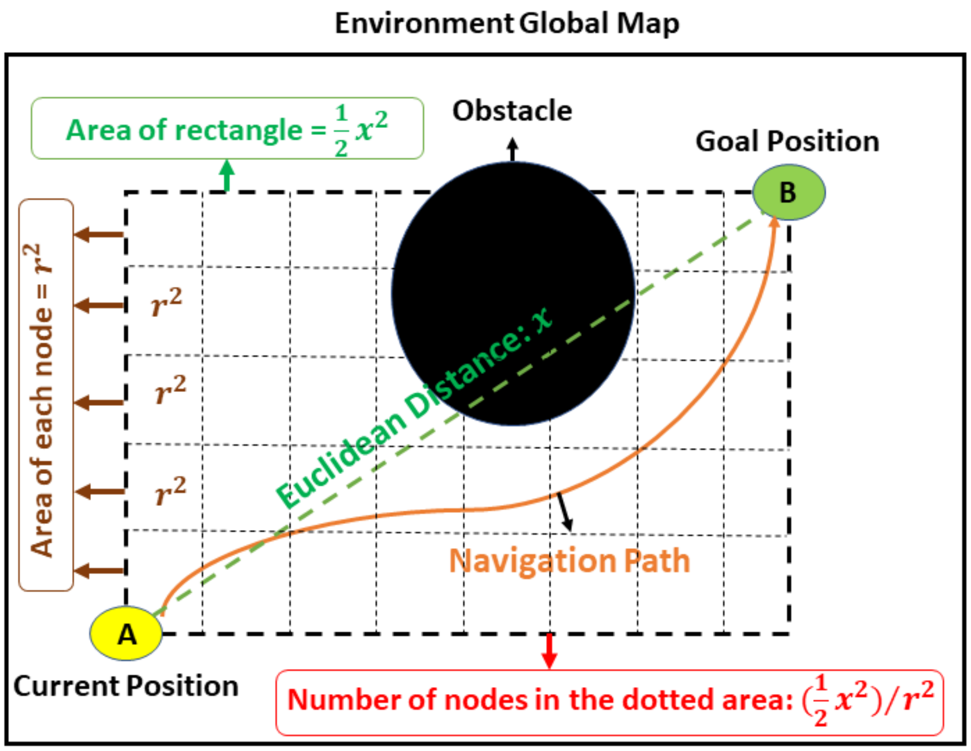
The values from the ROS layer are recorded in a Rosbag file, and that file is then used to train the DQN network and generate Q estimates for the state-action pair.

As mentioned earlier, the state space is built on the assumption that the size of input data (d_t) provided to an application directly impacts its execution time. In this experiment, we consider path planning for robot navigation and demonstrate how we derive the input data sizes for this application (d_t^p).

4.4.2 Path Planning

Path planning is a well-studied research area in robotics. The main objective of path-planning algorithms is to provide the shortest obstacle-free path between origin and destination. Among the well-known and well-studied algorithms for path planning are the Dijkstra [97], A* [98], breadth-first search [99], and depth-first search algorithms [99]. To ensure faster training and data collection, we limited the task in the present study to only plotting a path between the current position and the goal position, without actually moving the robot. Random goal coordinates were assigned inside the global map, and the Dijkstra algorithm was used to compute the shortest path.

A proper representation of computational size (d_t^p) is vital for the DQN algorithm to learn optimal policy. The objective in our experiment was not only to determine the shortest



Environment Global Map
Area of rectangle = $\frac{1}{2}x^2$
Obstacle
Goal Position
B
Area of each node = r^2
r^2
r^2
r^2
Euclidean Distance: x
Navigation Path
A
Current Position
Number of nodes in the dotted area: $(\frac{1}{2}x^2)/r^2$

Figure 4.6. Schematic representation for calculating the number of nodes that the robot needs to explore before reaching its destination.

obstacle-free path but to do so in the least possible execution time. The inputs for the path planner are the current position of the robot (A), the goal position (B), and the global map in the form of an image. Its output is an obstacle-free global path from the current position to the goal position, as depicted in Fig. 4.6. Dijkstra's algorithm uses a node-based approach to calculate the shortest path, and input size can be represented by $n\log n$ [100], where n represents the nodes. As is evident in Fig. 4.6, the number of nodes in the dotted rectangular space gives us a good estimate of the number of nodes (n) that the algorithm needs to explore before reaching the goal position. The algorithm will traverse these nodes several times to find the shortest path and hence the input size is represented as $n\log n$. The Euclidean distance from A to B is the diagonal for the rectangle. We can obtain the number of nodes in the rectangle by dividing the area of the rectangle by the area of each node inside it. Hence, for path planning we define the number of nodes n as,

$$n = (\frac{1}{2}x^2)/r^2 \tag{4.11}$$

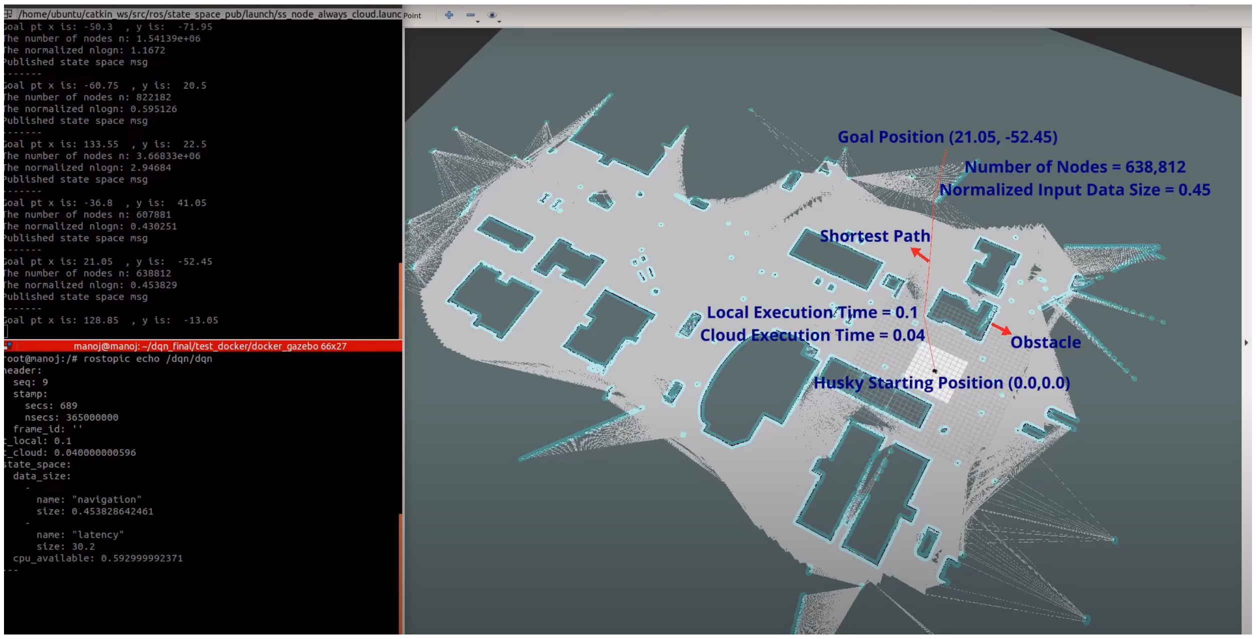
where the numerator represents the area of the rectangle with diagonal $\bar{AB}$ and the denominator represents the area of each node. The variable x is the Euclidean distance from A to B, and r is the length of each side in a node.

When ROS launches a navigation module, the granularity of the occupancy grid (r) is usually set to 0.05 meters [101], but can be manually changed as required by the application. A granularity of 0.05 means that each side of the square grid is 0.05 meters and the area of each square is $(0.05)^2$. Hence, the resolution of each node in the occupancy grid map can be represented as r and area of each node as r^2.

We can use the number of nodes n derived from Eq. (4.11) to determine the computational cost of path planning as,

$$d_t^p = n\text{log}n. \tag{4.12}$$

We demonstrate in Fig. 4.4 an example of how we calculated n and d_t^p. The starting position of the Husky was (0.001, -0.001) and the assigned goal position was (22.951, 39.054). The Euclidian distance (x) between those two positions can be calculated as approximately 45.09. The area of the occupancy grid r^2 as stated above was 0.0025. By substituting the values of x and r^2 in Eq. (4.11), we get the number of nodes (n) as being approximately 410,320. Substituting n in Eq. (4.12), we get d_t^p as approximately 5,303,251; the Husky will need to explore the nodes several times in order to obtain a shortest path. This implies that for the Husky to reach its goal position, it will need to traverse around 5,303,251 nodes ($n$log$n$) and the number of nodes in the in the rectangular area that husky will traverse several times to find the shortest path, can be represented as 410,320. In the experiment, we normalized the value of d_t^p by dividing by the global map size, which yielded values in the range of 0 to 5.



```
/home/ubuntu/catkin_ws/src/ros/state_space_pub/launch/ss_node_always_cloud.launc
Goal pt x is: -50.3  , y is:  -71.95
The number of nodes n: 1.54139e+06
The normalized nlogn: 1.1672
Published state space msg
-------
Goal pt x is: -60.75  , y is:  20.5
The number of nodes n: 822182
The normalized nlogn: 0.595126
Published state space msg
-------
Goal pt x is: 133.55  , y is:  22.5
The number of nodes n: 3.66833e+06
The normalized nlogn: 2.94684
Published state space msg
-------
Goal pt x is: -36.8  , y is:  41.05
The number of nodes n: 607881
The normalized nlogn: 0.430251
Published state space msg
-------
Goal pt x is: 21.05  , y is:  -52.45
The number of nodes n: 638812
The normalized nlogn: 0.453829
Published state space msg
-------
Goal pt x is: 128.85  , y is:  -13.05
```

```
manoj@manoj: ~/dqn_final/test_docker/docker_gazebo 66x27
root@manoj:/# rostopic echo /dqn/dqn
header:
  seq: 9
  stamp:
    secs: 689
    nsecs: 365000000
  frame_id: ''
t_local: 0.1
t_cloud: 0.040000000596
state_space:
  data_size:
    -
      name: "navigation"
      size: 0.453828642461
    -
      name: "latency"
      size: 30.2
  cpu_available: 0.592999992371
---
```

Point
Goal Position (21.05, -52.45)
Number of Nodes = 638,812
Normalized Input Data Size = 0.45
Shortest Path
Local Execution Time = 0.1
Cloud Execution Time = 0.04
Obstacle
Husky Starting Position (0.0,0.0)

Figure 4.7. A screenshot while the simulation environment is running. The left panel shows the linux terminal with state-space value (d, u, b) and actual execution times (l^{local}, l^{cloud}). The right panel shows a global map of the environment with the shortest path from the origin (0.0, 0.0) to the goal (21.05, -52.45) coordinates with a red line.

4.4.3 AWS and Latency

In this experiment, we utilized Amazon web services (AWS) as the cloud service provider. There are several means by which a robot can communicate with AWS; for this application, we chose ZeroMQ (ZMQ) [102]. ZMQ is a high-performance asynchronous messaging library that can be used in both distributed and concurrent applications, and furthermore is known for its excellent performance scalability and low latency. ZMQ provides a ROS-like publisher-subscriber messaging that supports several communication protocols, including WebSockets; in addition, the broker-less framework provided by ZMQ is faster than the inherent ROS communication framework. Given all these exciting features, ZMQ makes an excellent framework for establishing a communication protocol between robots and AWS. For the experiment, we used the Amazon Ohio instance with a static IP to ensure easier communication with the cloud service. As depicted in Fig. 4.8, the local machine sends obstacle information, the current robot position, and the goal position to the cloud. The cloud sends back a response in the form of a planned path.

The state space includes round trip latency (b_t) for each time step, which was calculated in real-time by pinging the cloud. The simulation was carried out on a computer that had a stable internet connection, and latency was constant at around 30 milliseconds with less than 5 percent variation for 99 percent of cases. This can be considered a drawback in the current experiment, as the latency values had minimal variation. In any case, the latency parameter did not play a meaningful role in deriving policy.

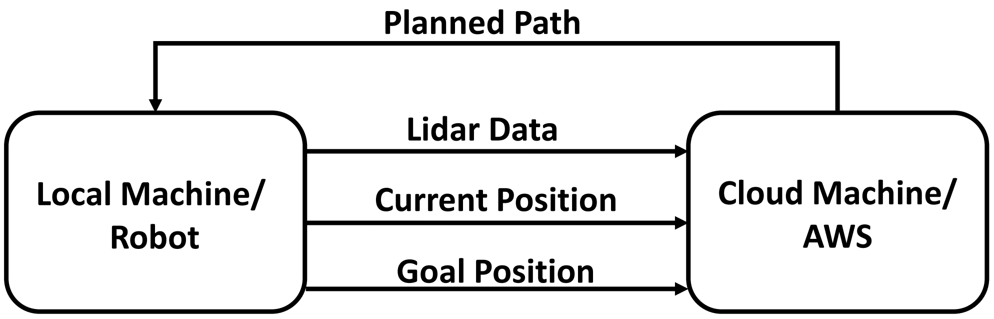


Figure 4.8. Robot-AWS communication framework. The robot forwards LiDAR data (Obstacles) and the current and goal positions to AWS. AWS calculates and forwards the shortest path between those positions to the robot.

4.4.4 DQN Network

We used a neural network consisting of one input layer, three hidden ReLU layers (each having 256 neurons), and one dense linear output layer. The configuration of hidden layers and the number of neurons can be altered based on convergence, training time, or any other performance criteria [103]. The choice of neural network can also be adapted to fit any specific learning problem, such as using a convolution neural network (CNN), long short term memory (LSTM), etc. [104]. Using the network parameters given in Table 4.1, we obtained a satisfactory convergence for the results.

Table 4.1. DQN network parameters used for training.

Parameters	**Values**
Number of hidden layers	3
Number of nodes	256; 256; 256
Mini-batch size	128
Learning rate	0.01
Exploration rate	0.1
Discount factor	0.9

4.5 Results and Analysis

In supervised learning, algorithm model evaluation is reasonably straightforward: the data is labeled and an evaluation set is used to assess the accuracy of the results [105]. However, model evaluation is tricky for algorithms based on DRL as they do not have a labeled dataset to be used as ground truth to validate the performance of the algorithm. As our problem formulation is based on acquiring maximum rewards and not the end-goal success criterion, our algorithm is designed as a continuous task problem with discrete action space, i.e., we apply a choice of action at each time step and use the corresponding outcome to train the DQN. When replay memory reaches a threshold of 1,000 experiences, experiences are replaced according to the first-in first-out (FIFO) approach.

In the first part of the results, we analyze the network performance with reference to actual data collected from the cloud and robot. Unlike in episodic problems, it is difficult to judge an agent's performance in a continuous problem as there is no terminal state that defines if the action was a success or a failure. Hence, we need to somehow generate a dataset that can be used to verify the agent's expected behavior. In addition, a synthetic dataset can be used to analyze whether the network is learning appropriate policy with respect to all possible outcomes. Accordingly, we generated three synthetic datasets where for each dataset we had a policy in mind that the agent should learn. Finally, we evaluated these datasets using the loss function and rewards acquired to determine if the agent performed as expected. The hardware configuration of the AWS (p2.xlarge) instance and robot are given in Table 4.2.

Table 4.2. Hardware configuration of the robot and AWS.

	Robot - Local	**AWS (p2.xlarge) - Cloud**
CPU	Intel Core i7-6700 CPU @ 3.40GHz	2.7 GHz (turbo) Intel Xeon E5-2686 v4
GPU	1 GeForce GTX 1050 - 768 processing cores and 4 gb of GPU memory	1 NVIDIA K80 - 2496 parallel processing cores and 12 gb of GPU memory
RAM	16 gb	61 gb
Cores	8	4
OS	Ubuntu-18.04	Ubuntu-18.04

As our model is designed for binary decision-making, to efficiently validate the proposed algorithm we need to have data distributed between both action space choices; i.e., both cases $l_t^{cloud} < l_t^{local}$ and $l_t^{cloud} > l_t^{local}$ need be reasonably represented. Using the hardware configuration seen in Table 4.2, we obtained a dataset with a split of about 60:40; i.e., 60 percent of decisions were for cloud computation ($l_t^{cloud} < l_t^{local}$) and 40 percent for local computation ($l_t^{cloud} > l_t^{local}$). Comparatively, if we choose a less capable hardware configuration, the dataset will be skewed towards cloud computation ($l_t^{cloud} < l_t^{local}$); similarly, if we choose a more powerful hardware configuration, the dataset will be skewed towards local computation ($l_t^{cloud} > l_t^{local}$) due to the additional round-trip time required for cloud communication.

4.5.1 Real Dataset

In this part of the experiment, we collected a real dataset where the robot and cloud were connected through ZMQ. We also implemented a cloud timeout functionality (5 s) to handle any network failures. This feature was added to punish the agent for choosing cloud computation if the cloud did not respond in a given timeframe. For every time step (t), the path-planning execution time was collected for both cloud execution (l_t^{cloud}) and local execution (l_t^{local}), along with state-space values (s_t) and the action (a_t) performed. These values were used to train the DQN network and evaluate the performance of our proposed algorithm.

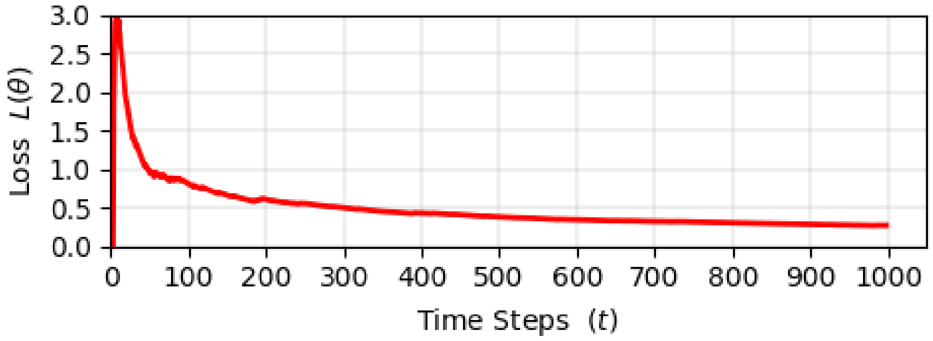


Figure 4.9. Plot showing loss convergence over time for the real dataset. Loss value plotted against time step ($t = 1000$); Plot shows average loss decreasing and convergence.

In Fig. 4.9, we plot the average loss ($L(\theta)$) against time step (t) using the loss function defined in Eq. (4.3). The plot demonstrates convergence and shows average loss as decreasing over time. This can be interpreted as the weight (θ) parameters of the network being optimized by gradient descent over time [106] and also as the network learning a more efficient policy. In Fig. 4.10, cumulative reward (r_t) is plotted against time step (t). As our algorithm is based on the foundation of acquiring maximum rewards rather than on the end-goal success criterion, DQN learning occurs over one single episode with t time steps, where $t = 1,2,3,..,T$. The episode reward plot shows rewards increasing over time, implying that the network learned the policy for acquiring maximum rewards, i.e., the action (a_t) to take for the given state space (s_t) in order to acquire maximum rewards ($\max(r_t)$). Rewards decreased at some time steps, mainly due to the network performing exploration [105], which helps it to form a better policy. The exploration rate is the probability that our agent will explore the environment rather than exploiting the original policy consensus; we set the exploration rate (epsilon-greedy) [105] value to 0.1.

We also assessed the accuracy of the entire dataset by evaluating the correctness of the action taken in context of the respective execution times. That is, the correct action should

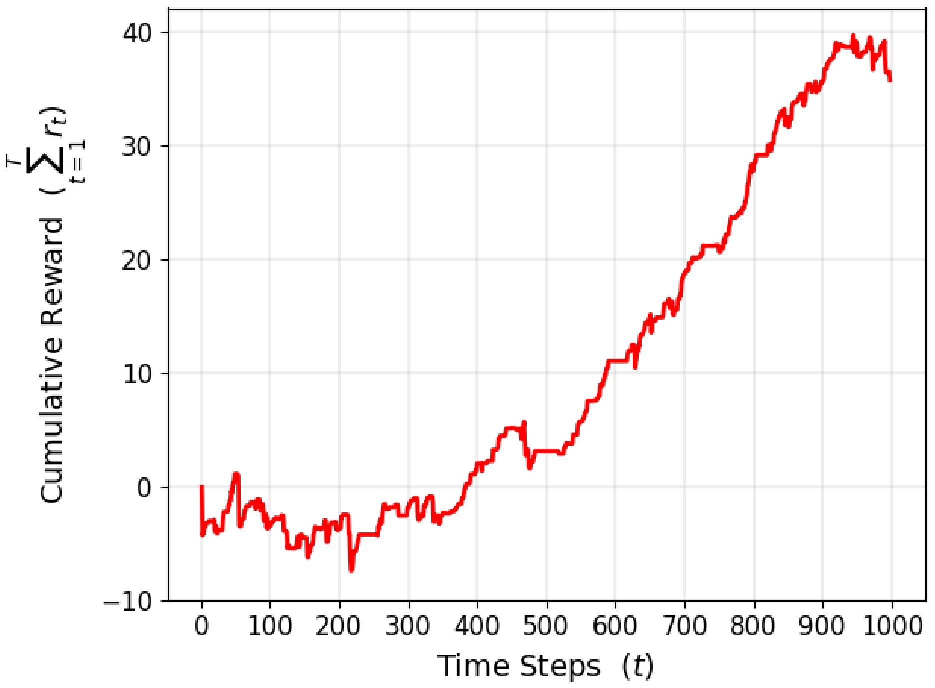


Figure 4.10. Plot showing rewards acquired over time for the real dataset. Rewards plotted against time step ($t = 1000$); Plot shows rewards increasing over time, implying that the network learned a policy for acquiring maximum rewards.

result in less execution time. We achieved a final accuracy of 84 percent on the dataset, suggesting that the algorithm learned to take correct actions with respect to the input state space over time. The overall mean execution time of actions selected by the algorithm was 71.28 milliseconds, while the respective means for local and cloud execution were 88.38 and 73.46 milliseconds. Thus, the proposed algorithm achieved execution time savings of almost 23 percent and 3 percent when compared with wholly local or wholly cloud execution. The cumulative execution times were 140.88, 146.09, and 175.81 seconds for algorithm-selected, cloud, and local execution, respectively. Hence, dynamic offloading using this algorithm reduced the latency of the application.

Even though the DQN network performed well with respect to average loss and rewards acquired, it is hard to intuitively ascertain what the agent learned. We plotted action (a_t) with respect to the size of path input data (d_t^p) and time step sequence (t) (Fig. 4.11) and observed that when input data size surpassed 1.1, the agent chose in the majority of

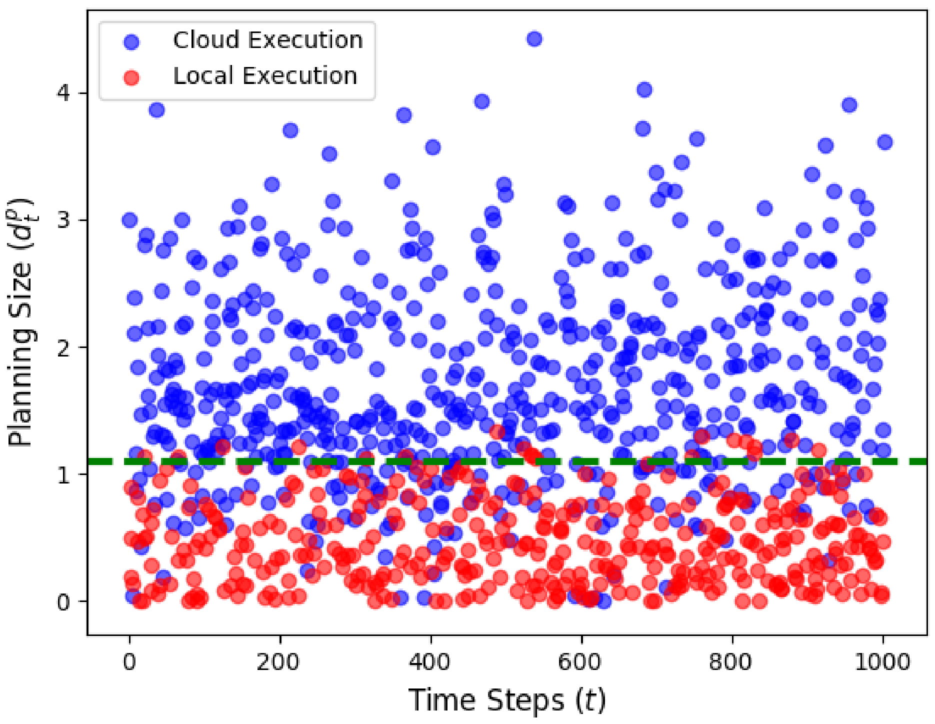


Figure 4.11. Choice analysis plot for the size of path planning data input (d_t^p). For normalized values $d_t^p > 1.1$, cloud computing was the preferred choice of the policy.

cases to offload the application to the cloud. Hence, data size impacted offloading choice, and when the planning problem data had a size greater than 1.1, executing the application on the cloud was the better choice; the application took less time to execute even with network latency. This observation also strengthens our hypothesis that the size of a problem is directly proportional to execution time. Thus, we conclude that the policy learned by the network is to choose cloud computation for path planning over larger areas.

4.5.2 Synthetic Dataset

In the previous section, we saw how the algorithm performed on a real data set. To further evaluate the algorithm's performance and behavior, we generated three different synthetic datasets with three different goals. We wanted to see if first, the network could learn to do onboard computation for a given state (i.e., local computation); second, if the network could learn to offload an application for a given state (i.e., cloud computation); and finally, if the network could learn a constant CPU availability value and use that as the basis for offloading decisions (i.e., learning a CPU availability at which to offload). Additionally, this section also provides insights into how reward (r_t) assignment varies for local and cloud computation.

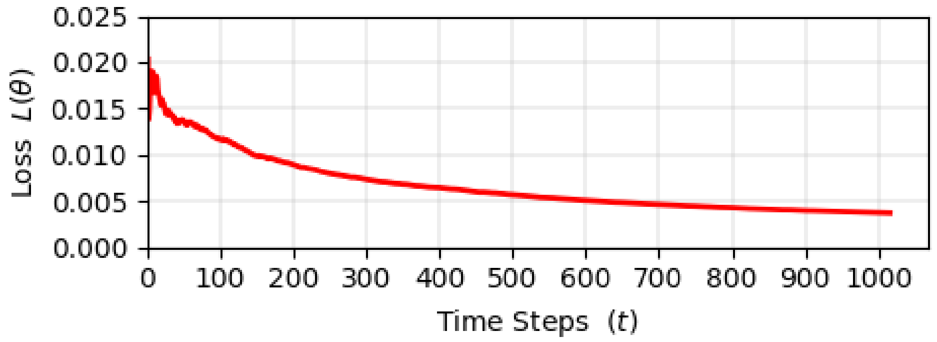


Figure 4.12. Loss convergence over time for a local only synthetic dataset. Loss value plotted against time step ($t = 1020$); Plot shows average loss decreasing and convergence.

Local Computation

To generate the synthetic dataset for local computation, we first obtained the local execution time (l_t^{local}) for path planning at each time step, then multiplied it with a random number from 0.9 to 1.9 to obtain the cloud execution time (l_t^{cloud}),

$$l_t^{cloud} = (0.90 + rand()\%10/10.00) * l_t^{local}. \tag{4.13}$$

Fig. 4.12 shows that average loss decreases and converges over time, implying that the algorithm has learned the policy to acquire maximum rewards. Fig. 4.13 plots cumulative rewards against time step. One important observation is that in this scenario, the possible reward r_t ranges from -1 to 0 inclusive, i.e., $r_t \in [-1, 0]$. While learning, the algorithm can choose to either execute the application onboard or to offload it; that is, in Eq. (4.10), the value for $c_t^{algorithm}$ can be either c_t^{local} or c_t^{cloud}. In this particular dataset, when $c_t^{algorithm}$ is c_t^{cloud}, the reward is always negative as $l_t^{cloud} > l_t^{local}$. In contrast, when $c_t^{algorithm}$ is c_t^{local}, the reward is zero.

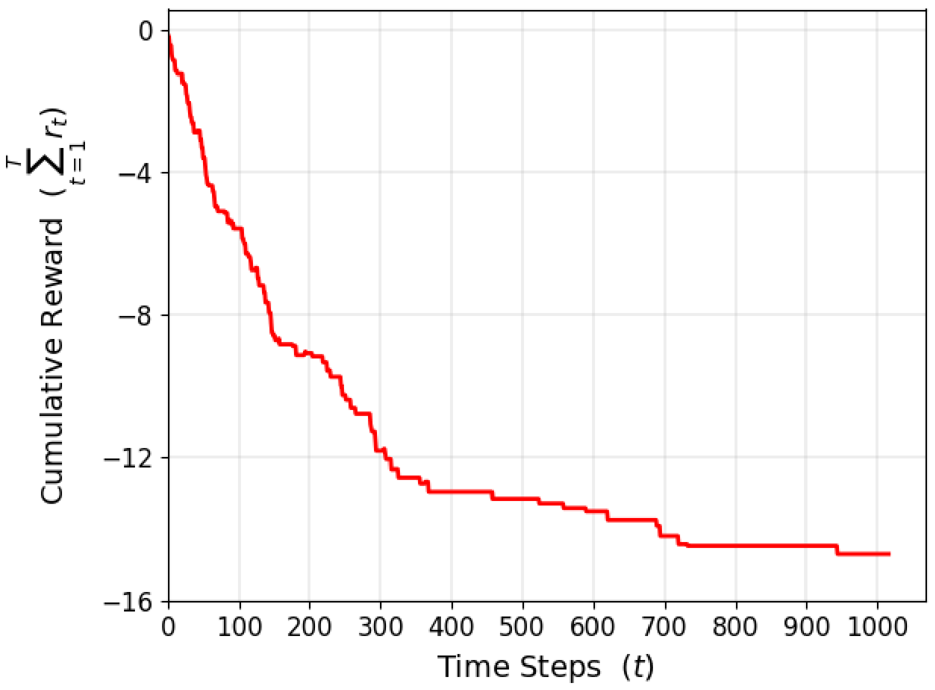


Figure 4.13. Rewards acquired over time for a local only synthetic dataset. Rewards plotted against time step ($t = 1000$); Plot shows rewards increasing over time, implying that the network learned a policy for acquiring maximum rewards.

The reward for the network (Fig. 4.13) started to stabilize at around time step 300, with no further negative rewards being gained; the algorithm had by that point learned the best possible action is not to gain any further negative rewards and to always choose local computation instead of cloud computation.

Cloud Computation

To generate the cloud computation synthetic dataset, we first obtained the local execution time (l_t^{local}) for path-planning at each time step, then multiplied that with a random number from 0.1 to 1.1 to obtain the cloud execution time (l_t^{cloud}),

$$l_t^{cloud} = (0.1 + rand()\%10/10.00) * l_t^{local}. \tag{4.14}$$

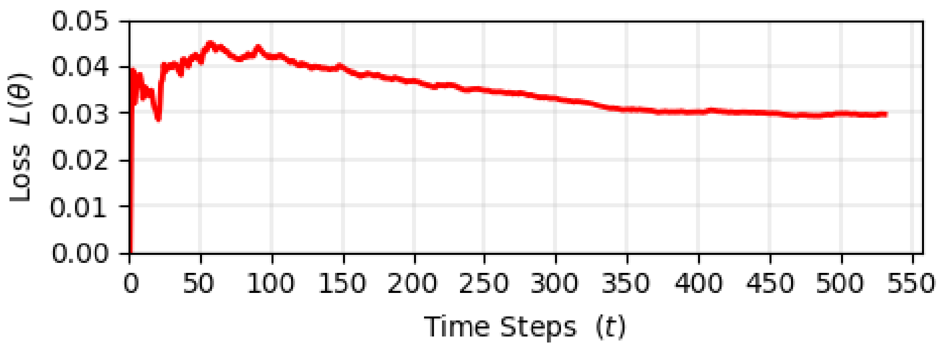


Figure 4.14. Loss convergence over time for a cloud only synthetic dataset. Loss value plotted against time step ($t = 530$); Plot shows average loss decreasing and convergence.

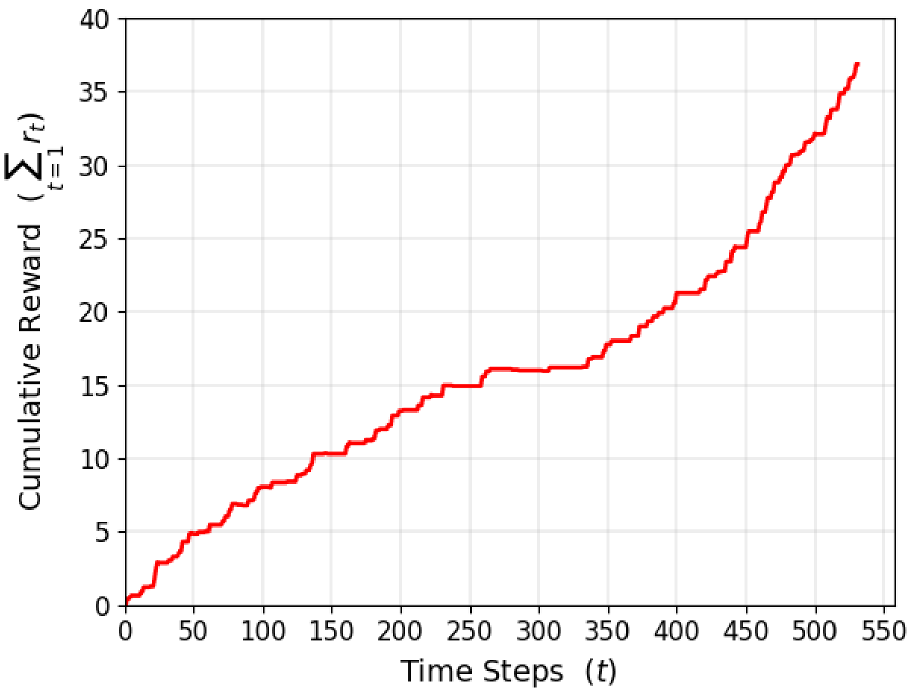


Figure 4.15. Rewards acquired over time for a cloud only synthetic dataset. Rewards plotted against the time step ($t = 530$); Plot shows agent learned a policy that choosing cloud computing over local computing gives it a positive reward.

Fig. 4.14 shows the average loss with this dataset as converging over time, implying that the algorithm learned the policy to acquire maximum rewards. Fig. 4.15 plots cumulative rewards against time steps. In this scenario, the synthetic data is skewed to favor cloud computation and the possible reward r_t for the agent ranges from 0 to 1 inclusive, i.e.,

$r_t \in [0,1]$. Thus, when $c_t^{algorithm}$ is c_t^{cloud}, the reward is always positive as $l_t^{cloud} < l_t^{local}$, and when $c_t^{algorithm}$ is c_t^{local}, the reward is zero. That the reward accumulated was always increasing indicates the agent learned that choosing cloud computing over local computing gives it a positive reward, and thus always chose cloud computing instead of local computing.

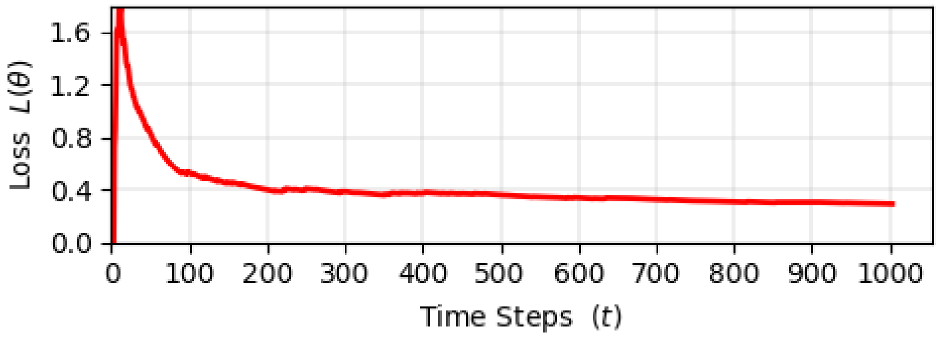


Figure 4.16. Loss convergence over time for a CPU based synthetic dataset values. Loss value plotted against time step ($t = 1050$); plot shows average loss decreasing and convergence.

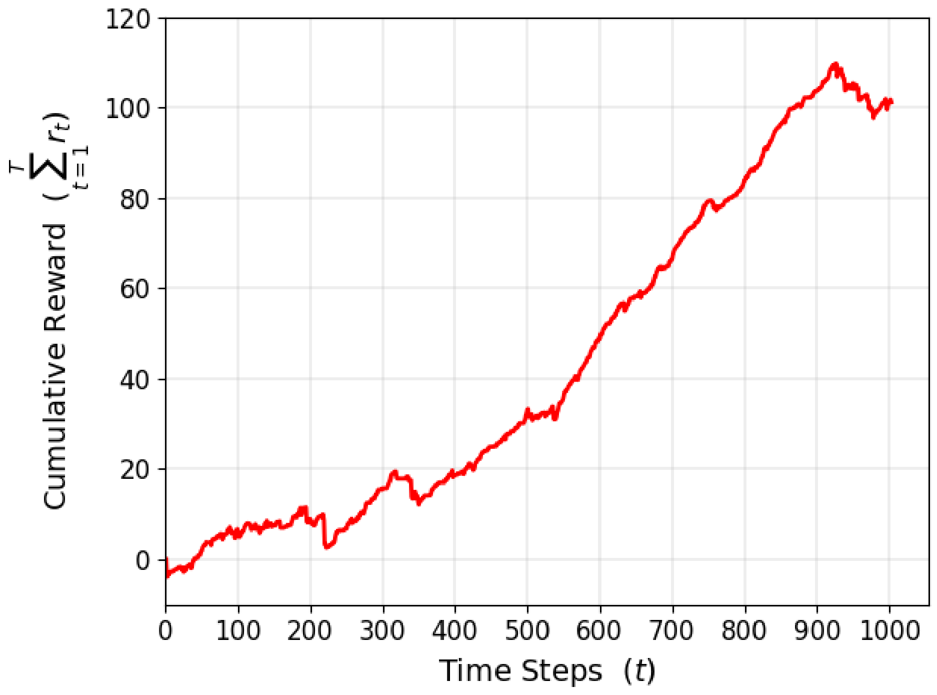


Figure 4.17. Rewards acquired over time for a CPU based synthetic dataset. Rewards plotted against the time step ($t = 1050$); Plot shows agent learning a policy to acquire positive rewards.

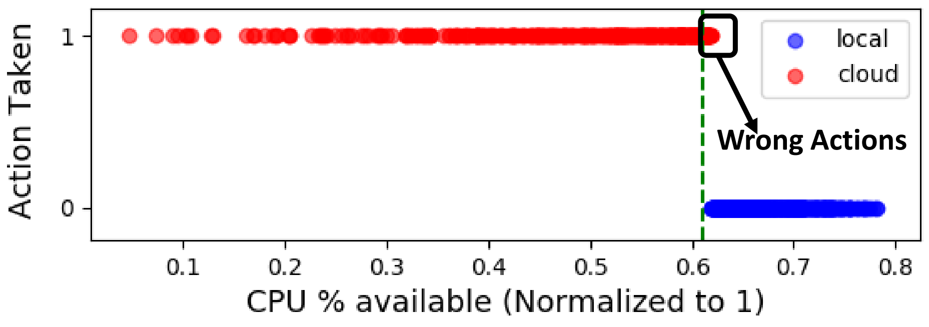


Figure 4.18. Choice analysis plot illustrating the CPU threshold value learned by the offloading decision policy, which was 0.61.

Learning a CPU Value to Offload

In the previous sections, we observed how data input size (d_t^p) affects the decision to offload. In this section, we wanted to see if the network can learn about a CPU value $u_t = x$ and use this value to decide whether to offload or do local computation. We generated a synthetic dataset where execution times l_t^{cloud} and l_t^{local} were set as follows,

$$\begin{aligned} l_t^{cloud} < l_t^{local} \quad when \quad u_t < x \\ l_t^{cloud} > l_t^{local} \quad when \quad u_t > x. \end{aligned} \tag{4.15}$$

When collecting the synthetic data, we observed CPU availability on the local machine to hover between 50 and 70 percent, and hence set the threshold value x as 60 percent. This allowed us to collect data that was distributed on both sides of x.

Fig. 4.17 plots cumulative reward against time step. It is difficult to interpret from this plot what the agent has learned; accordingly, we also rendered a scatter plot as illustrated in Fig. 4.18 and a confusion matrix as given in Fig. 4.19 in order to decode what the agent learned from the dataset. The goal of the policy was to extract the CPU availability value (y) from the synthetic data and use it as the basis for deciding between offloading and local computation. The learned threshold value (y) was around 0.61 (Fig. 4.18), while the pre-set x value for generating l_t^{cloud} and l_t^{local} was 0.6; we can therefore conclude that the network was within a reasonable margin of error. This is further supported by the training accuracy (Fig. 4.19), with the network having achieved 94 percent accuracy; most wrong actions took

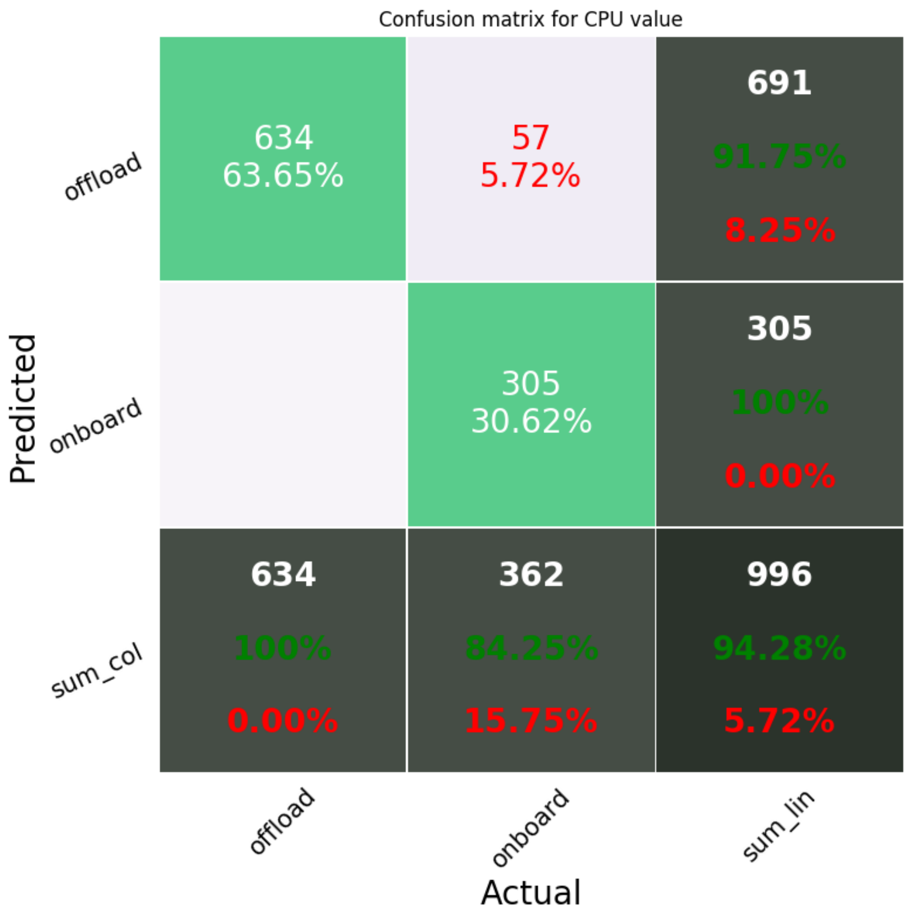


Figure 4.19. Confusion matrix used to determine the accuracy of the network; Over 94 percent accuracy was achieved.

place when c_t was around 0.6. Finally, the average loss decreased and converged over time (Fig. 4.16), implying that the algorithm learned the policy of offloading the application whenever CPU availability was less than 61 percent and of doing local computation when it was greater than 61 percent.

4.5.3 Comparative Evaluation With Long Short-Term Memory Algorithm

We further evaluate the proposed DRL algorithm by comparing it with another state-of-the-art machine learning model. One approach for such evaluation is to compare the algorithm with other DRL models such as DoubleDQN [107] or DuelingDoubleDQN [108]. These models are quite similar to each other as they all learn value functions and act greedily

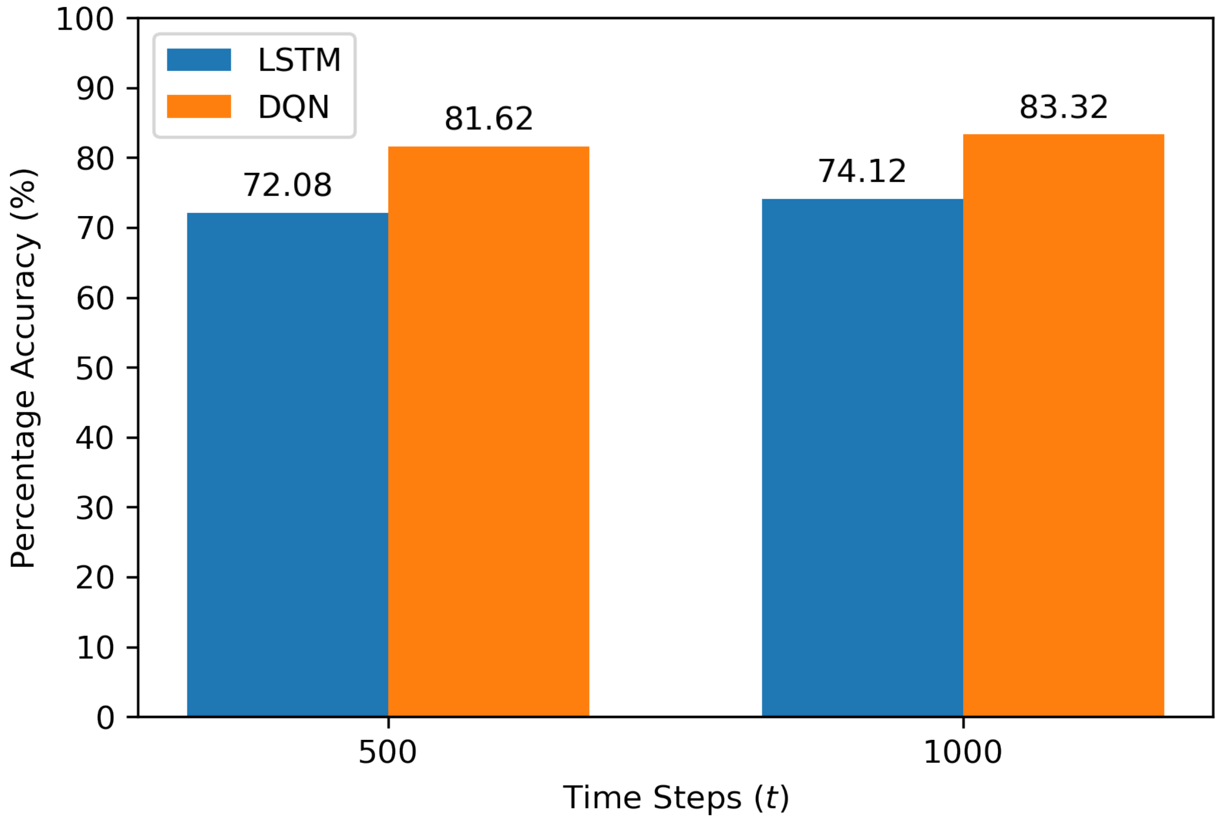


Figure 4.20. Bar chart comparing application offloading prediction accuracy for LSTM and proposed DRL algorithms. Prediction accuracy was calculated by training the models with the first 500 time steps t and with the entire dataset of 1000 time steps. Our proposed DRL algorithm achieved better accuracy in both cases.

based on those values; the major difference between them lies in performance [89], and hence, comparing these models will imply comparing performance parameters such as how loss convergence varies for various batch sizes, learning rates, discounted factors, etc. However, the major emphasis of this paper is to understand the policy learned by the agent and the final accuracy of the algorithm. As such, rather than comparing performance among various DRL models, a more appropriate evaluation is to compare the accuracy of the proposed algorithm with other state-of-the-art machine learning models.

The literature has predominantly suggested using long short-term memory (LSTM) models for predicting execution time [109]. Hence, we implemented a LSTM model based on [110] (hidden layers = 4, batch size = 50, steps/batch = 8) to predict application execution time from input data size. Fig. 4.9 and Fig. 4.10 demonstrate that for the DRL model, loss starts to converge and cumulative rewards to steadily increase from around $t = 400$; this implies that by that step, the agent has figured out a policy to acquire positive rewards. Thus, we used actual data from the first 500 time steps to train the LSTM model, with an 80:20 training:validation split. We then used the trained LSTM to predict execution times for both cloud and local execution of the application and compared resulting predicted actions with actual actions obtained from the proposed DRL algorithm. As shown in Fig. 4.20, while the LSTM model achieved a final accuracy of 72.08 percent, the proposed DRL algorithm achieved a greater accuracy of 81.62 percent. Similarly, when using the entire 1000-time step dataset with an 80:20 split to train the LSTM model, it had a final accuracy of 74.12 percent, whereas the proposed DRL algorithm achieved 83.32 percent. Hence, we can conclude that our proposed DRL algorithm is comparable to or better than LSTM in predicting appropriate actions to take concerning application offloading with reference to the application data input size.

4.5.4 Discussion

The results demonstrated that the agent was able to learn a policy that maximized reward and reduced overall application execution latency. We also observed that the size of application input data from the state space played an important role in the policy forming a consensus. In this section, we will present some of the limitations and key observations of the present study. Notably, the overall accuracy after training was greater than 80 percent for all experimental cases we tested. Considering we only carried out the simulation for around 1000 time steps, this accuracy gives us confidence in the policy learned by the agent. We are also certain that the accuracy can be increased by increasing the time steps (t) taken to train the network.

One important observation from the results is the algorithm convergence. From the loss plots, we can discern that the algorithm started to converge before 500 time steps in all cases. Such rapid convergence is vital in reducing the amount of time that needs to be spent on training and enables the algorithm to start taking correct actions (a_t) more consistently in less time.

One limitation of this study is that there is no one true guideline that defines how to correctly represent the size of input data for an application, and sometimes it falls to the personal intuition of the human operator to correctly represent this metric. Representations of input data size can vary based on factors such as the algorithm implemented, the computational model used, and also on the application; for example, we represented the size of path-planning input data as $n\log n$. Similarly, if we want to search n randomly ordered elements, the input data size is n; binary search is $\log n$; and sorting is $n\log n$ [111]. For image detection-based algorithms, we can assume that frames per second multiplied by the data size of each frame would be a good representation of the input data [112]. Ultimately, the accuracy of the proposed algorithm depends on the correctness of the input data size determination, and so it's not a foolproof system.

One of the other limitations of this study is that the Gazebo-based experimental setup might not offer a perfect representation of the network latency in a robotic environment. The experiment was carried out on a computer that was connected to the internet through a LAN cable with capacity greater than 400 Mbps. The latency between the computer and AWS on this setup was constant at around 30 milliseconds. In a real-world scenario, this case might not hold as robots are mostly connected through a wireless connection, and so might experience fluctuating internet speeds along with network dropouts depending on the environment. Even though we introduced a latency parameter (b_t) in the state space, the network never truly learned anything meaningful about connection latency as the parameter had minimal variation.

Finally, even though we verified the algorithm for a robot navigation application, it can be generalized to a majority of robotic applications. This research also verifies that there is a relationship between input data size and the time needed for execution. Hence, if we model our state space to capture information concerning data size, the algorithm will converge and learn a policy for offloading decision-making.

4.6 Conclusion

A dynamic computational offloading solution based on DQN has been proposed in this chapter. The proposed algorithm was able to learn an optimal policy on when to offload an application based on state-space values. The state space was built on the assumption that the size of input data submitted to an application directly impacts its execution time, and we successfully validated this assumption using the size of path input data (d_t^p) in a navigation application. The algorithm was designed as a continuous task problem with discrete action space for every time step (t), and at each step the execution time for path planning in robot navigation was collected for both cloud execution and local execution, as were state-space values and the action performed. These values were used to train the DQN network and evaluate the performance of our proposed algorithm.

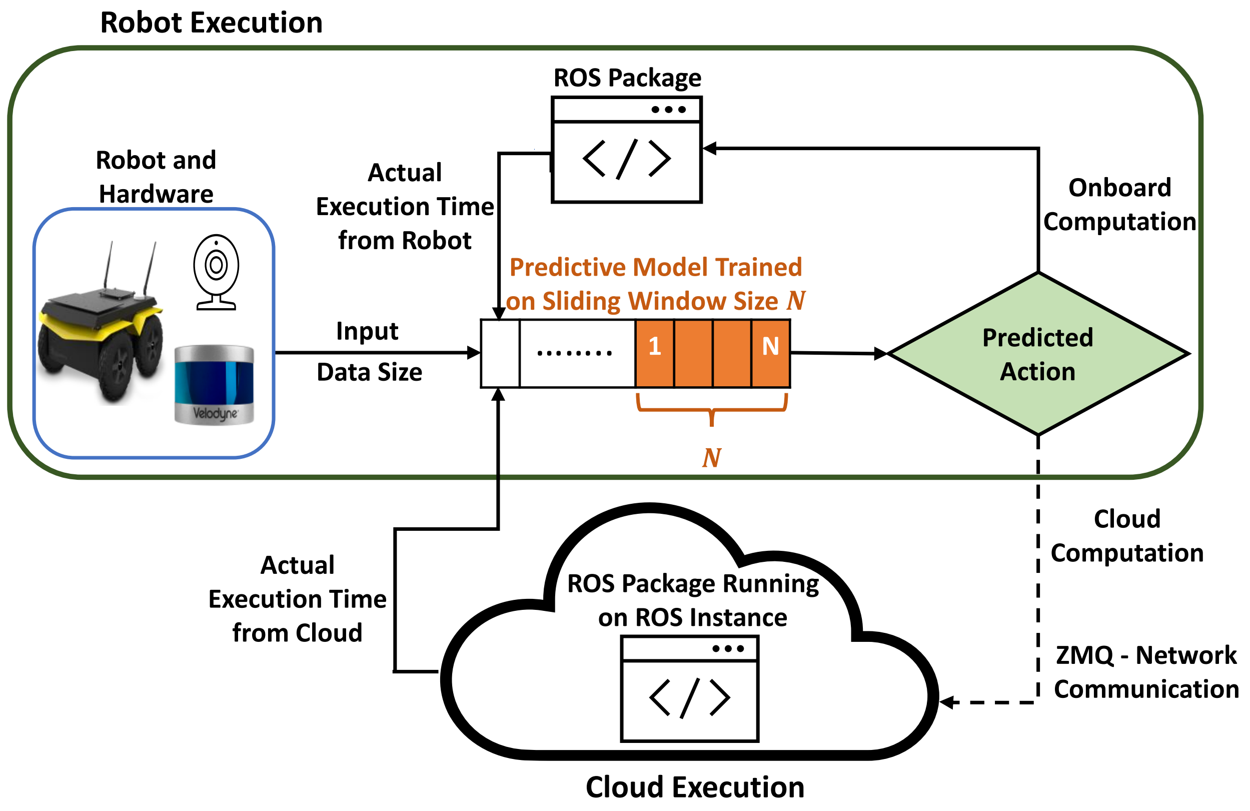
The effectiveness of the algorithm was evaluated by observing the loss and rewards over time steps. The results showed convergence and the agent learning a policy to maximize rewards over time. To further evaluate the algorithm, we also generated three synthetic datasets, each designed around a particular policy that the agent should learn. The network successfully extracted the key features from these datasets, and the agent learned the policy that we intended. All told, the results have validated the effectiveness of our proposed algorithm.

5. PREDICTIVE OFFLOADING

Many robotic applications that are critical for robot performance require immediate feedback, hence execution time is a critical concern. Furthermore, it is common that robots come with a fixed quantity of hardware resources; if an application requires more computational resources than the robot can accommodate, its onboard execution might be extended to a degree that degrades the robot's performance. Cloud computing, on the other hand, features on-demand computational resources; by enabling robots to leverage those resources, application execution time can be reduced. The key to enabling robot use of cloud computing is designing an efficient offloading algorithm that makes optimum use of the robot's onboard capabilities and also forms a quick consensus on when to offload without any prior knowledge or information about the application. In this chapter, we propose a predictive algorithm to anticipate the time needed to execute an application for a given application data input size with the help of a small number of previous observations. To validate the algorithm, we train it on the previous N observations, which include independent (input data size) and dependent (execution time) variables. To understand how algorithm performance varies in terms of prediction accuracy and error, we tested various N values using linear regression and a mobile robot path planning application. From our experiments and analysis, we determined the algorithm to have acceptable error and prediction accuracy when $N > 40$.

5.1 Introduction

Although robot onboard resources are often fixed, modern robots do typically come with a fair amount of computational resources, and it is important to consider these resources in offloading decision-making. Ideally, the offloading algorithm should make a decision based on cost parameters such as execution time, energy usage, and CPU availability. Researchers have recently proposed several dynamic offloading solutions that use machine learning algorithms and consider cost parameters [113]. However, these machine learning algorithms require training on large datasets to make accurate predictions. Also, no two robots and applications are the same, thus a machine learning model trained to work for a certain type of application and robot is not guaranteed to work well for other combinations, and such models are not easy to generalize. On top of all that, obtaining large datasets is itself a challenging task. Hence, it is important to also consider an algorithm that can be trained quickly and learn on the fly using smaller datasets. We introduce here a predictive algorithm, outlined in Fig. 5.1, that is designed to help robots make offloading decisions without any prior knowledge about the application.



Robot Execution
ROS Package
Robot and Hardware
Actual Execution Time from Robot
Onboard Computation
Predictive Model Trained on Sliding Window Size *N*
Input Data Size
1
N
Predicted Action
N
Cloud Computation
Actual Execution Time from Cloud
ROS Package Running on ROS Instance
ZMQ - Network Communication
Cloud Execution

Figure 5.1. Illustration of the proposed predictive algorithm for offloading decision-making. (Top) Input data size from robot hardware (e.g. LiDAR, camera) along with actual execution times from cloud and robot executions are used to train the predictive model. The training set consists of the previous N values to facilitate learning on the fly and faster training. (Bottom) An instance of ROS running on the cloud subscribes to ROS topics from the robot and responds with application output.

The remainder of the chapter is organized as follows: In Section 5.2, we describe related work on application offloading algorithms. In Section 5.3, we introduce a predictive algorithm. In Section 5.4, we present linear regression and a mobile robot path planning application to validate the generalized algorithm. In Section 5.5, we provide extensive experimental results and analysis. Finally, we conclude our work and present our future directions in Section 5.6.

5.2 Related Work

For robots with bare minimum computational capabilities, full application offloading will be an obvious choice. However, many robots currently produced are computationally capable, hence are capable of onboard execution of many applications; this leads to dynamic application offloading being more appropriate, with offloading decisions taking into account both the robot's computational capability and the application's computational requirements. Hence, application offloading approaches can be broadly categorized into full offloading and dynamic offloading.

Since the inception of cloud robotics in 2009, several architectures have been proposed that mainly focused on full offloading; these cater to specific applications such as navigation, image classification, and localization [4], [43], [63], [114].

To our knowledge, the area of dynamic application offloading for cloud robotics is understudied, with only a few studies published to date that focus on dynamic application offloading. In one example, Rahman *et al.* proposed an offloading solution for cloud-networked multi-robot systems that was based on a genetic algorithm and focused on energy efficiency as the criterion informing decision-making [115]. Alli *et al.* proposed an offloading solution based on the Neuro-Fuzzy model, a machine learning method, which aims to minimize latency and energy for smart city applications [116]. Some other solutions based on edge computing and machine learning have also been proposed [60], [69], [70], [117]–[119].

In the last couple of years, additional approaches have tried applying deep reinforcement learning (DRL) [56] -based algorithms for making application offloading decisions. Chicachali *et al.* [81] used such a strategy to offload an object detection application to the cloud, while Peng *et al.* [83] used a DRL-based Deep-Q-Network algorithm for offloading based on energy consumption and task makespan.

In reviewing the literature, we observed various extant solutions based on genetic algorithms, machine learning, and DRL. All of these algorithms require large datasets and substantial time in training to achieve convergence and make accurate predictions. Also, there is no guarantee that the trained algorithm will work for other types of applications and robots. Hence, it is important to design a lightweight offloading algorithm that can learn after the application is initiated. Our proposed algorithm is mainly designed to being adaptable to diverse applications and not requiring any pretraining. As the proposed algorithm is not pre-trained, it is important for it to quickly learn optimal decision-making. To make the algorithm faster to train, we train it on a fixed dataset of N previous observations. We experiment with various N to observe the correlation, prediction accuracy and error.

5.3 Generalized Predictive Algorithm

In this section, we introduce a generalized version of the predictive algorithm (Ψ) for application offloading. The simple nature of the proposed predictive algorithm empowers the user to proceed without any of the pre-computing that is usually required for machine learning algorithms or other optimizing techniques. This chapter is interested in a useful, generalized dynamic offloading algorithm that can be applied to robotic applications and has an independent variable that can used to predict a dependent variable.

An application execution usually consists of subtasks, where each task can be independent or dependent on another. These tasks process the received input and give a resulting output. Here, we categorize an application A into application tasks a_{i}, where $A \in a_{\mathrm{i}}$ and i is the application task sequence number, and for simplicity we assume that application tasks are executed in sequence. The time needed for execution of an application depends on its input arguments, such as the size of the data (d_{i}). Our basic premise is that we can predict both cloud execution (p^c_{i}) and local execution (p^l_{i}) times using the input arguments provided to the application, thus all applications can be generalized as the sum of all components dependent on application arguments. These predicted values p^c_{i} and p^l_{i} can be used in offloading decision-making, and generally we can represent the offloading choice as:

$$\begin{aligned} &\text{Cloud computation} \quad &when \quad p^c_{\mathrm{i}} < p^l_{\mathrm{i}} \\ &\text{Onboard computation} \quad &when \quad p^c_{\mathrm{i}} \geq p^l_{\mathrm{i}}. \end{aligned} \tag{5.1}$$

To compare p_{i}^{c} and p_{i}^{l}, a predictive function is needed; however, estimating execution time can be simple or complex. Many algorithms and functions can be created to predict execution time by means of an algorithm built from the values of the application's input variables. For small systems with straightforward relationships, n-dimensional regressions are the obvious choice; meanwhile, for systems with many variables and complex relationships, machine learning is commonly used. Some of the most widely-used predictive algorithms are linear regression [120], random forest [121], time-series algorithms [122], and k-means clustering [123]. The time complexity of an application with one major variable can be written as $O(u)$, where $u = 1$, n, $n\log n$, n^2, 2^n, etc. For an application carrying out the same task repeatedly on a dataset of constant size, the complexity is $O(1)$. For an application using only one variable, regardless of complexity, estimating p_{i}^{c} and p_{i}^{l} with regression from sample data coordinates is straightforward.

One key aspect of the proposed predictive algorithm is that it includes a fixed number N, the window size of the most recent execution data (cloud and local) that is used for training the algorithm to form a consensus and predict the execution times p_{i}^{c} and p_{i}^{l}. Figure. 5.2 and Algorithm 2 describes the predictive model, where we use First-In First-Out (FIFO) queue data structures B^l and B^c to store the N training values. Having a fixed training size keeps memory low and keeps prediction runtimes low. By retaining the most recent observations, the algorithm can also account for exogenous factors that influence execution time. For example, even though we define that application execution time depends on the input arguments, there will be instances where the execution time of the application varies based on external factors not captured in the input arguments, such as network loss and high CPU usage by other applications. By storing the N most recent executions, we can capture some of the external factors that affect execution time. After training the algorithm with the stored queue values, the algorithm will receive current state values and predict execution times that can be used for offloading decision-making.

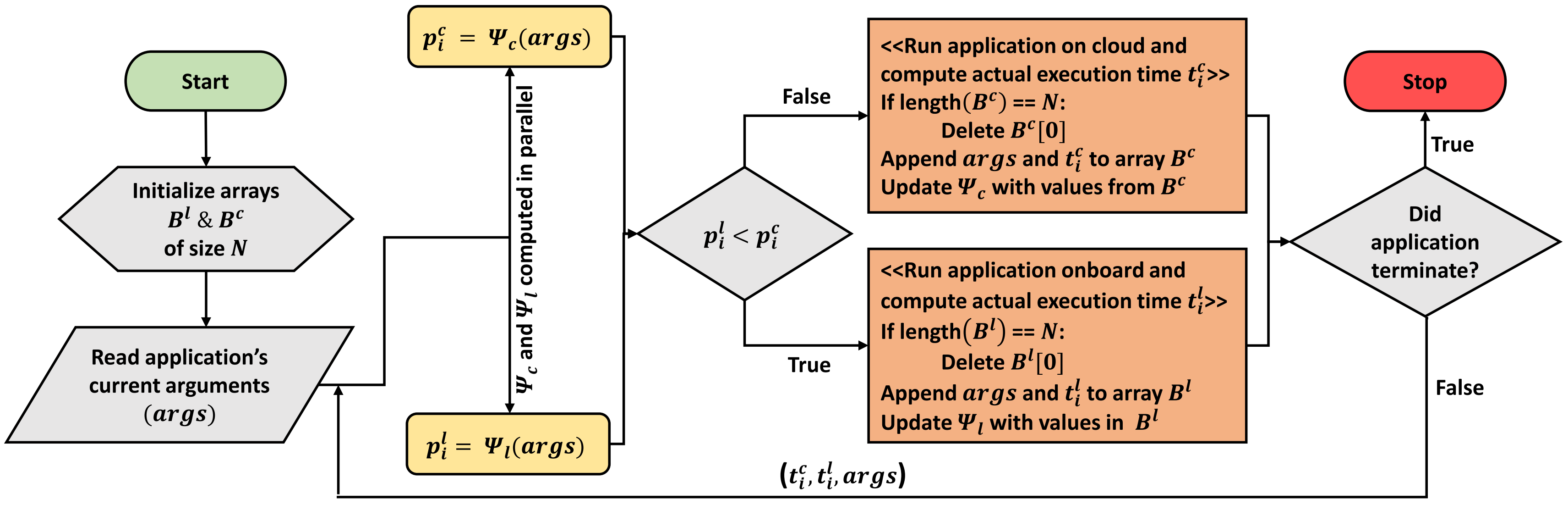

Start
Initialize arrays B^l & B^c of size N
Read application's current arguments $(args)$
$p_i^c = \Psi_c(args)$
Ψ_c and Ψ_l computed in parallel
$p_i^l = \Psi_l(args)$
$p_i^l < p_i^c$
False
True
<<Run application on cloud and compute actual execution time t_i^c>>
If length$(B^c) == N$:
Delete $B^c[0]$
Append $args$ and t_i^c to array B^c
Update Ψ_c with values from B^c
<<Run application onboard and compute actual execution time t_i^l>>
If length$(B^l) == N$:
Delete $B^l[0]$
Append $args$ and t_i^l to array B^l
Update Ψ_l with values in B^l
Did application terminate?
True
Stop
False
$(t_i^c, t_i^l, args)$

Figure 5.2. Flowchart of the generalized predictive algorithm. The specific predictive algorithms Ψ_c and Ψ_l are trained on values from the arrays B^l and B^c to predict the execution times p_{i}^c and p_{i}^l. Arrays B^l and B^c contain the previous N observations of actual execution times t_{i}^c and t_{i}^l along with arguments $args$ passed to the application.

Algorithm 2 Predictive algorithm for application offloading

```
Initialize FIFO based Queues $B^l$ and $B^c$ with size $N$;
for For application task sequence i = 1 do
    repeat
        Fit $\psi^l$ and $\psi^c$ with $B^l$ and $B^c$;
        Load input data size $d_{\rm i}$ for $a_{\rm i}$;
        Predict $\psi^l(d_{\rm i}) = p^l_{\rm i}$ and $\psi^c(d_{\rm i}) = p^c_{\rm i}$;
        if $size(B^l)$ and $size(B^c) == N$ then
            if $p^l_{\rm i} < p^c_{\rm i}$ then
                Execute the application onboard;
                Calculate $t^l_{\rm i}$;
                Append $B^l$ with $(t^l_{\rm i}, d_{\rm i})$;
            else
                Execute the application on AWS;
                Calculate $t^c_{\rm i}$;
                Append $B^c$ with $(t^c_{\rm i}, d_{\rm i})$;
            end if
        else
            Execute application onboard and AWS;
            Calculate $t^l_{\rm i}$ and $t^c_{\rm i}$;
            Append $B^l$ with $(t^l_{\rm i}, d_{\rm i})$;
            Append $B^c$ with $(t^c_{\rm i}, d_{\rm i})$;
        end if
        Set $\rm i = i + 1$;
    until Application is terminated;
end for
```

5.4 Linear Regression and Mobile Robot Path Planning Application

In this section, we introduce the linear regression and the mobile robot path planning application that we use to validate the proposed algorithm.

5.4.1 Linear Regression Model

The goal of the algorithm (Algorithm 2) is to predict $p^c_{\rm i}$ and $p^l_{\rm i}$ (dependent variables) from an independent variable that has some correlation with execution time. Application execution time is proportional to the input data size, and there exists a linear relation between them [94]. Hence, we want to predict execution time from the input data size ($d_{\rm i}$). For predictions involving linear relationships, n-dimensional regressions are the obvious choice, hence we used a lightweight linear regression-based predictive model (ψ) to predict p^c_i and p^l_i from d_i. We term this algorithm lightweight is because we train it on a fixed dataset of size N. The duration of training will thus be considerably shorter relative to other models trained with larger datasets. Finally, to validate the proposed algorithm, we need a dataset with actual application execution times for both cloud computation (t^c_i) and local computation (t^l_i) and also the corresponding input data sizes (d_i). We derived this information for a mobile robot path planning application, the process of which will be explained in sections 5.4.2 and 5.4.3.

To predict local execution time (p^l_i), we used linear regression with the least squared method, as follows:

$$p^l_\mathrm{i} = m^l * d_\mathrm{i} + c^l \tag{5.2}$$

where m^l is the slope and c^l is the intercept, which are derived as follows:

$$\begin{aligned} m^l &= \frac{\sum_{\mathrm{i}=1}^{N}(d_\mathrm{i} - \overline{d}) * (t^l_\mathrm{i} - \overline{t^l})}{\sum_{\mathrm{i}=1}^{N}(d_\mathrm{i} - \overline{d})^2} \\ c^l &= \overline{t^l} - m^l * \overline{d}, \end{aligned} \tag{5.3}$$

where N is the size of the window of previous observations used for fitting the linear regression model, $\overline{d}$ is the mean of the input data size, and $\overline{t^l}$ is the mean of the actual execution time from N previous observations.

Similarly, to predict cloud execution time (p^c_i), we can represent the linear regression equation with the least squared method as follows:

$$p^c_\mathrm{i} = m^c * d_\mathrm{i} + c^c \tag{5.4}$$

where m^c is the slope and c^c is the intercept, which are derived as follows:

$$\begin{aligned} m^c &= \frac{\sum_{\mathrm{i}=1}^{N}(d_{\mathrm{i}} - \overline{d}) * (t_{\mathrm{i}}^c - \overline{t^c})}{\sum_{\mathrm{i}=1}^{N}(d_{\mathrm{i}} - \overline{d})^2} \\ c^c &= \overline{t^c} - m^c * \overline{d}, \end{aligned} \tag{5.5}$$

where $\overline{t^c}$ is the mean of the actual execution time from N previous observations.

5.4.2 Robot Path Planning Platform

Using the framework shown in Fig. 5.3, we obtain actual execution times t_{i}^c and t_{i}^l for the corresponding d_{i} for a robot path planning application. Based on the predicted execution times p_{i}^c and p_{i}^l, the robot executes the application either on the cloud or locally. Afterwards, the robot stores actual execution times (t_{i}^c and t_{i}^l) along with corresponding data size (d_{i}) values in a queue of size N. Finally, the predictive algorithm Ψ iteratively updates with new values from that queue for each prediction.

To validate the proposed algorithm in a simple and practical environment, we used the same Gazebo world environment seen in chapter 4.4.2. For path planning, the robot needs to have a map of its environment and also needs to be capable of performing simultaneous localization and mapping (SLAM) in the given environment to obtain an obstacle-free path from the origin to the destination [124]. In our experiment, we used Dijkstra's algorithm [125] to compute the shortest path as seen in chapter 4.4.2

5.4.3 Cloud Platform

On the cloud side of the framework, we used Amazon Web Services (AWS) with an Ohio instance. The simulation was carried out on a laptop with a stable wired internet connection having speed greater than 400 Mbps. The average latency observed for data making a round trip between cloud and robot was around 30 milliseconds. To establish communication between the robot and AWS, we used the ZMQ communication protocol [102], which is a high-performance asynchronous messaging library that provides a ROS-like

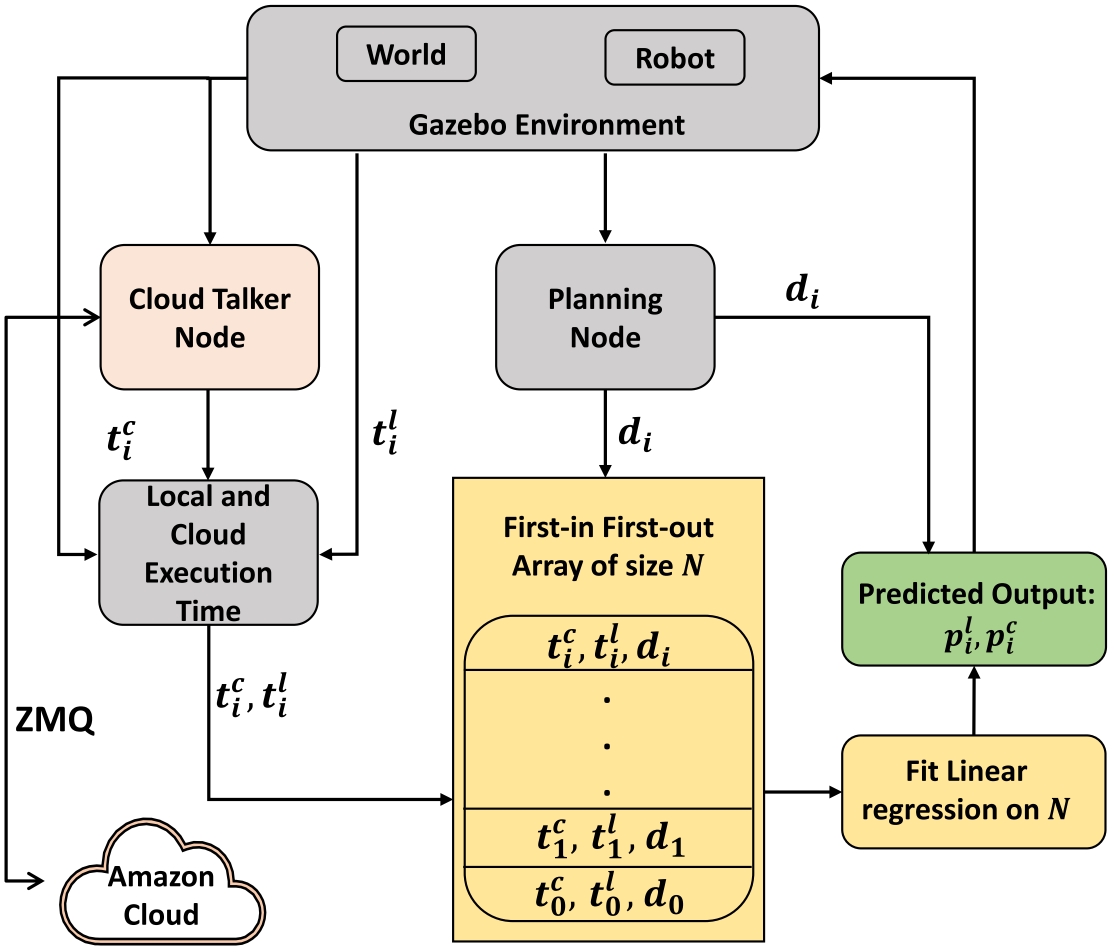


Figure 5.3. Navigation framework for algorithm validation. A mobile robot performs path planning inside an gazebo world by interacting with AWS. The data from the simulation are fitted to a regression model to predict execution times for a given input data size.

publisher-subscriber functionality. These along with other functionalities [126] made ZMQ an ideal choice for our application.

On AWS, we had a ROS instance running that subscribed to ROS topics from the robot. In this experiment, the robot published LiDAR data and its current and goal positions. The ROS instance on the cloud subscribed to these topics and published back a planned path. In parallel, the robot also computed a planned path on its local ROS instance. We calculated the actual execution times t_{i}^c and t_{i}^l by subtracting the time of data publication from the time at which each instance's planned path was received.

5.5 Results and Analysis

To evaluate the performance of the decision-making algorithm with training queue of varying size N, we programmed a linear regression model based on the logic illustrated in Algorithm 2 on top of the navigation framework described in Fig. 5.3. To maintain consistency across the results, we used a singular dataset that consisted of 1,000 rows with actual execution times t_i^l, t_i^c and corresponding d_i, and evaluated training with $N = 5, 10, 20, 30, 40, 50, 75, 100$, and 500 samples. We did not consider queue of size $N < 5$, as with so little data there is a good chance of underfitting and insufficient variation for the model to learn. The hardware configuration of the robot and the AWS instance (p2.xlarge) used to generate this dataset are given in Table 5.1.

Table 5.1. Hardware configuration of the robot and AWS instance used to generate performance evaluation data.

	Robot - Local	**AWS (p2.xlarge) - Cloud**
CPU	Intel Core i7-6700 CPU @ 3.40GHz	2.7 GHz (turbo) Intel Xeon E5-2686 v4
GPU	1 GeForce GTX 1050 - 768 processing cores and 4 gb of GPU memory	1 NVIDIA K80 - 2496 parallel processing cores and 12 gb of GPU memory
RAM	16 gb	61 gb
Cores	8	4
OS	Ubuntu-18.04	Ubuntu-18.04

After initialization of the application, the first N elements are solely used to train the linear regression algorithms (ψ^l, ψ^c) that are in turn used to predict execution times (p_i^l and p_i^c) for informing offloading decision making. During execution, the actual time of execution is computed (t_i^l or t_i^c) and, along with the corresponding d_i, are appended to the corresponding queue, from which the first (oldest) value is then deleted. The corresponding predictive algorithm ψ^l or ψ^c will then be refitted based on the updated queue to predict the execution time for the next application task ($a_{\mathrm{i}+1}$) from a given input data size. The results of the performance evaluation are presented in Tables 5.2 - 5.3 - 5.4, and in the next subsections we analyze those results in terms of correlation, residuals, and accuracy.

Table 5.2. Average correlation values for actual time of execution t^l and t^c

N	**Average Correlation t^c and d (Pearson r)**	**Average Correlation t^l and d (Pearson r)**
5	0.31726	0.73545
10	0.31717	0.71970
20	0.30455	0.71198
30	0.30245	0.70822
40	0.30282	0.70922
50	0.30273	0.70641
75	0.29914	0.69775
100	0.28395	0.70290
500	0.28060	0.69707
1000	0.27761	0.69221

5.5.1 Correlation

We used bivariate Pearson correlation [127] to estimate the correlation coefficient r, which helps us determine the strength of association between independent (d) and dependent variables (t^l and t^c). This analysis suggested a low positive correlation of $r(1{,}000) = .27761$ between cloud execution time (t^c) and application input data size (d) and a strong positive correlation of $r(1{,}000) = .69221$ between local execution time (t^l) and application input data size (d). Both coefficient values were statistically significant with $p < 0.005$. These correlations imply that with an increase in input data size (d), there is also an increase in actual execution time, whether local or cloud (t^l and t^c). Furthermore, the evident stronger correlation of local execution time (t^l) with input data size (d) reflects the limited availability of computational resources in onboard execution; when application input data size increases, the application required more computational resources than the robot could accommodate,

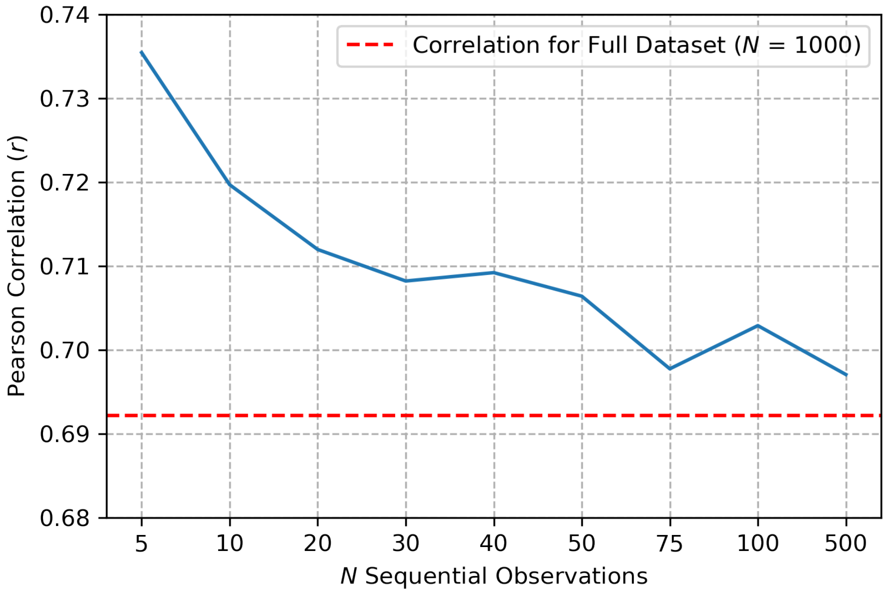


Figure 5.4. Pearson r correlation plot for actual cloud execution time (t^l) and input data size (d).

hence extending execution time. In contrast, cloud execution is better supported in terms of computational resources, and an increase in application input data size (d) resulted in a smaller increase of the cloud execution time (t^c).

We additionally investigated the impact of N on the correlation between dependent and independent variables. Ideally, as N increases, correlation values should remain the same or become stronger, reflecting better prediction values. We calculated correlation values for a moving window size of N across the training dataset, then averaged the results to obtain an average correlation value for the given N. As seen in Table 5.2, the average correlation values were slightly better for smaller N, implying that smaller training datasets captured slightly better association between independent (d) and dependent variables (t^l and t^c). However, the differences were slight, indicating that training data size did not substantially affect the correlation between independent and dependent variables.

Hence, we can assume that a larger sample size does not imply a stronger correlation; in fact, in our case, a larger sample actually weakens the correlation by a slight degree as seen in Fig. 5.4 and Fig. 5.5. This goes to show that the homogeneity of the sample is more

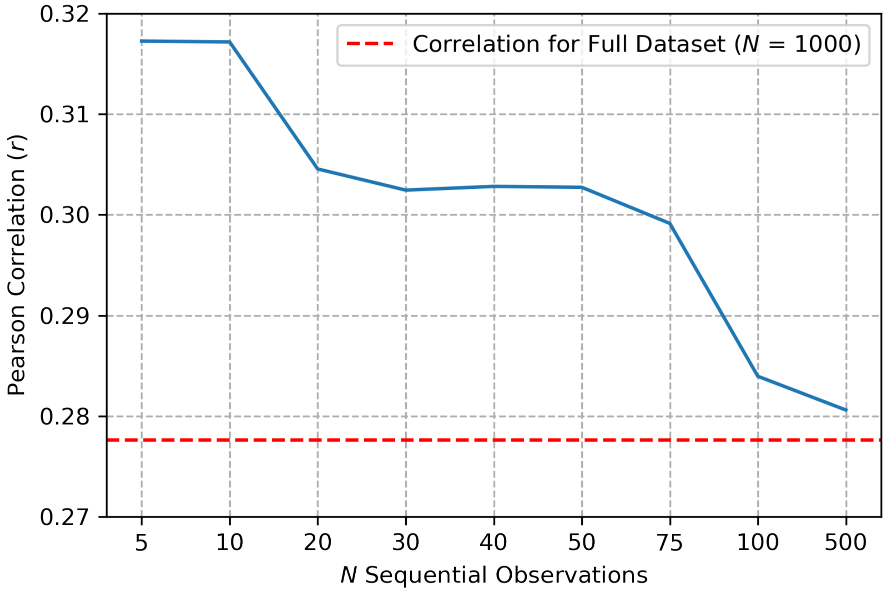


Figure 5.5. Pearson r correlation plot for actual cloud execution time (t^c) and input data size (d).

important than its size. How we collect the sample is also a key factor; if the data were sampled randomly, the correlation could have been lower. As we sequentially collected data during the experiment, the sample maintained a correlation comparable to larger datasets. Hence, we can conclude that smaller sequential datasets can be effective in predicting the execution times of robotics applications.

5.5.2 Residuals

Residuals (specifically, the difference between mean t and mean p) help us determine the error between actual and predicted execution times. Residual values for both cloud and local execution for models trained on different dataset sizes are listed in Table 5.3. With cloud computing, the residual was around 0.03681 for $N = 5$, which implies a 26 percent error rate (residual / actual t^c). Error rates decreased with increasing N, being about 13 percent for $N = 10$, 4 percent for $N = 20$ & 30, and less than 1 percent for all higher N values. A

Table 5.3. Running windows means for cloud execution (t^c and p^c) and local execution (t^l and p^l)

N	**Running Window Mean** t^c **(Seconds)**	**Running Window Mean** p^c **(Seconds)**	**Running Window Mean** t^l **(Seconds)**	**Running Window Mean** p^l **(Seconds)**
5	0.14016	0.17697	0.17527	0.19099
10	0.14029	0.15990	0.17537	0.17888
20	0.14005	0.14595	0.17546	0.17807
30	0.14005	0.14578	0.17546	0.18259
40	0.14016	0.14056	0.17552	0.18078
50	0.14025	0.14061	0.17554	0.18027
75	0.13991	0.13997	0.17519	0.17769
100	0.14048	0.13925	0.17564	0.17583
500	0.13481	0.14012	0.17113	0.17649
1000	0.14011	0.14796	0.17530	0.18151

less than 1 percent error rate implies an average difference between actual and predicted execution times of less than 0.39 milliseconds.

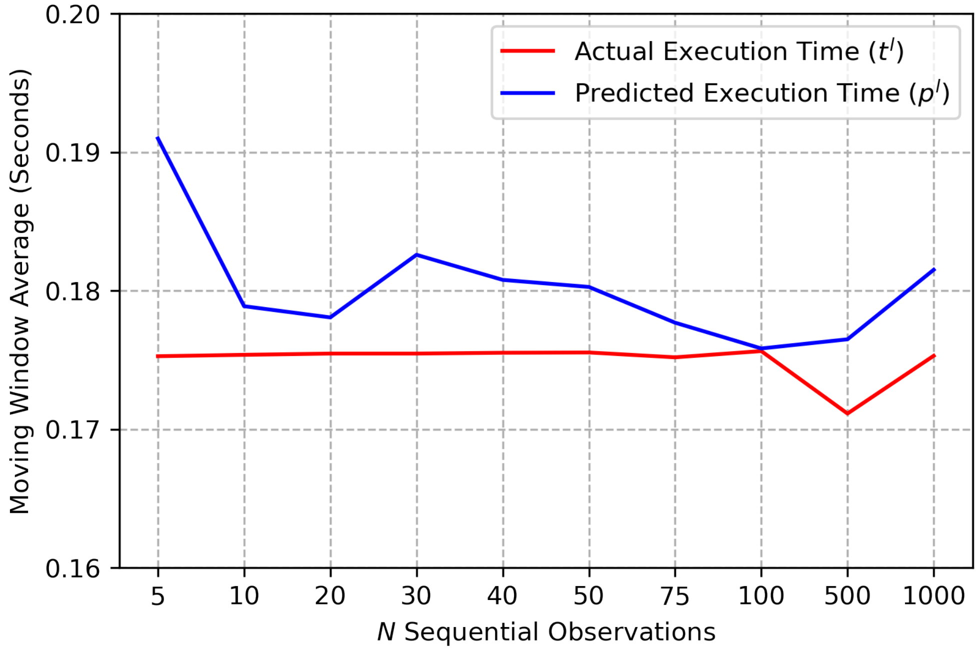


Figure 5.6. Running window mean for cloud actual execution time (t^l) and predicted time (p^l). Maximum prediction accuracy is achieved at around $N = 100$.

Meanwhile, error rate for predictions of local execution time for $N = 5$ featured a residual of 0.01572 and a resulting error rate of about 8 percent, while all higher N values had error rates of around 2 percent. Due to the high correlation between local execution time and input data size, there was better convergence and prediction accuracy at lower N. The 2 percent error rate implies an average difference of 3.5 milliseconds between actual and predicted execution times.

As seen in Fig. 5.6 and Fig. 5.7, one of the takeaways we want to highlight is that error percentages were high when $N = 5$, but less than 2 percent when $N > 40$. This implies that complex algorithms trained on large datasets are not needed to accurately predict offloading decision-making. A simple lightweight regression algorithm fitted on the previous N elements will be able to efficiently decide when to offload the application to the cloud. In this case, $N > 40$ was sufficient to yield an acceptable level of error rate of less than 2 percent.

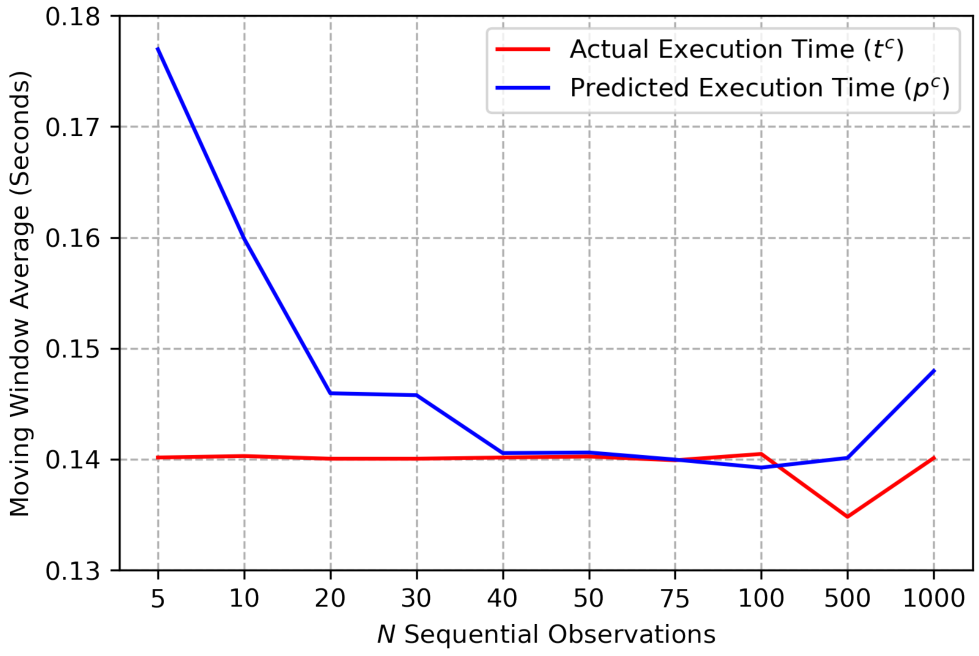


Figure 5.7. Running window mean for cloud actual execution time (t^c) and predicted time (p^c). Maximum prediction accuracy is achieved at around N = 40.

5.5.3 Decision Making Accuracy

To evaluate the accuracy of the algorithm, we compared the predicted action obtained from p^l and p^c (Eq. 5.1) with the correct action obtained from t^l and t^c. Table. 5.4, shows the prediction accuracy for various N values. The accuracy was around 60 percent for $N = 5$ and around 79 percent for $N = 50$. Above $N = 50$, only marginal increases in accuracy were obtained. We also trained the model on the entire dataset ($N = 1{,}000$) with 80:20 (training:testing) split, the result from which implies that the best possible accuracy for the linear regression will be around 83 percent for the given dataset.

We then further evaluated the proposed model by comparing it with other state-of-the-art machine learning models. Unlike in robotics, application offloading is well-studied in the context of mobile devices [128], [129], which literature has predominantly used Long Short-Term Memory (LSTM) for predicting execution time [109]. Hence, we implemented a LSTM model based on [110] (hidden layers = 4, batch size = 50) to predict application

Table 5.4. Prediction accuracy for various N

N	**Prediction Accuracy (%)**
5	60.02
10	64.95
20	70.60
30	73.38
40	76.32
50	78.80
75	79.74
100	79.62
500	81.90
1000	83.32

execution time from input data size. For training and testing, we used an 80:20 split of the entire dataset ($N = 1,000$). We then compared the predicted actions obtained from LSTM predicted execution times with the actual actions obtained from t^l and t^c. As seen in Fig. 5.8, the LSTM model achieved a final accuracy of 75.68 percent whereas the linear regression achieved an accuracy of 83.32 percent. Hence, we can conclude that our predictive algorithm was comparable in accuracy to LSTM for $N = 1,000$.

Observing the residual values and accuracy gives us reasonable confidence in the ability of the proposed algorithm to correctly predict execution times when trained on smaller datasets.

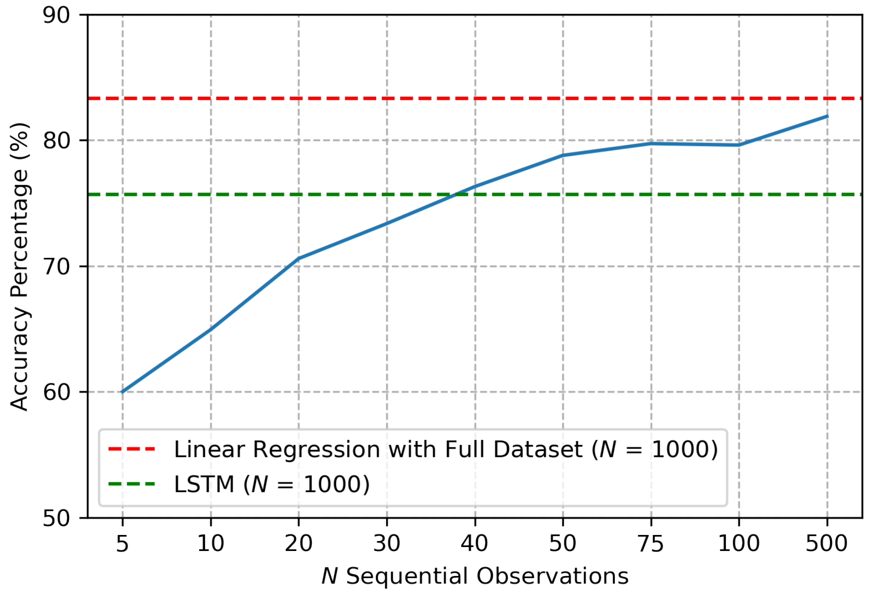


Figure 5.8. Prediction accuracy for various N. The red dotted line indicates the accuracy of linear regression on the full dataset. The green dotted line indicates the accuracy of an LSTM model on the full dataset. For full dataset 80:20 training:validation split was used.

5.5.4 Discussion

In this chapter, we propose a predictive algorithm for offloading decision-making, and further validate the algorithm using linear regression with a robot path planning application.

Several exogenous factors can directly or indirectly affect the execution time of an application. It might not be practically feasible to capture all such variables and train a machine learning algorithm to incorporate them into highly-accurate predictions of execution time. These influencing factors are often periodical, such as when a new application is launched alongside currently-running applications, requiring all of the applications share computational resources and thus extending execution time until one or another application terminates. Another example is network failure, which will extend the cloud execution time until network connectivity is restored. These exogenous factors are incredibly difficult to predict, but by considering N previous values we can capture some periodical factors and so predict execution times that are true to the current state of the system. Moreover, we observed a prediction accuracy of around 80 percent with only 2 percent difference between the actual and predicted values when we considered a training set of size $N = 40$. With such a small N value, the predictive algorithm will be quicker to train, will output its predictions faster, and will require much data for training compared with traditional machine learning algorithms.

One potential drawback of the proposed algorithm is that it only works effectively when both an independent variable and a significantly correlated dependent variable exist. Furthermore, the linear algorithm is usually sensitive to outliers, hence outliers must be appropriately addressed before fitting the data. Another drawback with a linear algorithm is that the data should have a linear relation. If the number of prediction variables is increased, issues with data linearity might occur. Finally, we also validated the proposed algorithm for subtasks that execute sequentially, but it remains to be seen how the algorithm performs for applications that execute in parallel. Perhaps parallel execution will need to be implemented in a distributed setup. Another drawback we want to highlight is that the accuracy of the prediction depends on the correlation between input data and predicted values; without a significant correlation, predicted values will be unreliable and can result in less-correct actions.

Finally, we do not claim that these models outperform state-of-the-art machine learning models. Machine learning models trained on large datasets with a variety of features will outperform these predictive models. However, it is not always possible to have a pre-trained machine learning model for a variety of applications and robot hardware. Our proposed predictive model can be used as an alternative in situations where prior knowledge is not available and offloading decisions need be made on the fly as quickly as possible.

5.6 Conclusion

In this chapter, we introduce a predictive algorithm to predict the execution time of an application for a given application data input size and then use that prediction for decision-making regarding application offloading. As the proposed algorithm starts learning after the application is initiated, minimizing training time is of highest priority; this is achieved by training the algorithm on a small number of previous data observations (N). We expect that this training approach will enable the capture of exogenous factors that are not directly incorporated in the model's design.

To validate the proposed predictive algorithm, we used linear regression and a Gazebo world. We experimented with varying N values to evaluate performance, and we found that the algorithm had an acceptable error and prediction accuracy when $N > 40$.

6. CONCLUSION AND FUTURE WORK

This dissertation presented our research on application offloading for cloud robotics. We proposed a novel cloud robotics architecture with first-of–its-kind features and validated the feasibility of using the architecture with full computational offloading. In Chapter 4 and 5, we proposed dynamic computational offloading algorithms based on application input data size. After the summary of previous chapters below, the direction of future research is suggested.

6.1 Conclusion

Cloud robotics is a relatively new area of research in which the biggest limitation is the decision of what, when, and how to distribute an application between on-board resources and cloud computing resources. With this in mind, we designed a novel cloud robotics architecture called *Smart Cloud*, which is the first of its kind to incorporate JavaScript-based libraries for running diverse robotic applications related to machine learning and more. The architecture is designed to cater to both heterogeneous and homogeneous multi-robot systems as well as single-robot systems. Using full offloading, we demonstrated the efficiency of the architecture by evaluating its CPU usage, latency, and security.

For robots with bare minimum computational capabilities, full application offloading will be an obvious choice. However, many robots currently produced are computationally capable, hence are capable of onboard execution of many applications; this leads to dynamic application offloading being more appropriate, with offloading decisions taking into account both the robot's computational capability and the application's computational requirements. Hence for Chapter 4, we proposed dynamic computational offloading solution based on DQN. Our problem formulation is based on acquiring maximum rewards and not the end-goal success criterion, our algorithm is designed as a continuous task problem with discrete action space, i.e., we apply a choice of action at each time step and use the corresponding outcome to train the DQN. The proposed algorithm was able to learn an optimal policy on when to offload an application based on state-space values. The state space was built on the assumption that the size of input data submitted to an application directly impacts its execution time, and we successfully validated this assumption using the size of path input data in a navigation application environment. Finally, even though we verified the algorithm for a robot navigation application, it can be generalized to a majority of robotic applications. This research also verifies that there is a relationship between input data size and the time needed for execution. Hence, if we model our state space to capture information concerning data size, the algorithm will converge and learn a policy for offloading decision-making.

The proposed DQN algorithm and other algorithms that we studied in our literature review, require large datasets and substantial time in training to achieve convergence and make accurate predictions. Also, there is no guarantee that the trained algorithm will work for other types of applications and robots. Hence, it is important to design a lightweight offloading algorithm that can learn after the application is initiated. Our proposed algorithm is mainly designed to being adaptable to diverse applications and not requiring any pretraining. As the proposed algorithm is not pre-trained, it is important for it to quickly learn optimal decision-making. In Chapter 5, we introduced a predictive algorithm to predict the execution time of an application for a given application data input size and then use that prediction for decision-making regarding application offloading. As the proposed algorithm starts learning after the application is initiated, minimizing training time is of highest priority; this is achieved by training the algorithm on a small number of previous data observations (N). We expect that this training approach will enable the capture of exogenous factors that are not directly incorporated in the model's design. To validate the algorithm, we trained the algorithm on the previous N observations, which include independent (input data size) and dependent (execution time) variables. To understand how algorithm performance varies in terms of correlation, prediction accuracy and error, we tested various N values using linear regression and a mobile robot path planning application. From our experiments and analysis, we determined the algorithm to have acceptable error and prediction accuracy when $N > 40$.

6.2 Future Research Direction

Building on the research foundation provided by this dissertation, future research can be directed on the following aspects.

For Chapter 3, the future work can focus on the development of tools and mechanisms to lower additional cost parameters such as latency and energy. The architecture can be further evaluated in terms of metrics such as latency, scalability, interoperability, availability, and security.

For Chapter 4, the additional work to accommodate a multi-robot scenario can be considered. In future work, we can also focus on adding additional cost parameters such as energy usage. We can explore various application prioritization mechanisms that can help prioritize mission-critical applications, and we can apply mechanism-specific cost parameters to the applications. For example, applications that are critical for robot functioning will have reducing latency as the first priority, while applications that are not mission-critical will prioritize reducing their own energy use. To expand the research even further, we can considers algorithms such as DDPG [56] for a multiple application scenario using a single state space.

For Chapter 5, we can consider additional cost parameters such as energy usage. We can also explore algorithms, such as ARIMA and SARIMA [122], that can help predict execution times for applications whose variable relationships are non-linear.

REFERENCES

[1] W. Wang and K. Siau, "Artificial intelligence, machine learning, automation, robotics, future of work and future of humanity: A review and research agenda," *Journal of Database Management*, vol. 30, pp. 61–79, Jan. 2019. DOI: 10.4018/JDM.2019010104.

[2] S. Alsamhi, O. Ma, and S. Ansari, "Convergence of machine learning and robotics communication in collaborative assembly: Mobility, connectivity and future perspectives," *Journal of Intelligent and Robotic Systems*, Oct. 2019. DOI: 10.1007/s10846-019-01079-x.

[3] *Cloud robotics market-size-analysis forecast (2018 - 2023)*, https://www.mordorintelligence.com/industry-reports/cloud-robotics-market.

[4] 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, Apr. 2015, ISSN: 1545-5955. DOI: 10.1109/TASE.2014.2376492.

[5] T. Dillon, C. Wu, and E. Chang, "Cloud computing: Issues and challenges," in *2010 24th IEEE International Conference on Advanced Information Networking and Applications*, 2010, pp. 27–33.

[6] A. Lenk, M. Klems, J. Nimis, S. Tai, and T. Sandholm, "What's inside the cloud? an architectural map of the cloud landscape," in *2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, May 2009, pp. 23–31. DOI: 10.1109/CLOUD.2009.5071529.

[7] J. KUFFNER, "Cloud-enabled humanoid robots," *Humanoid Robots (Humanoids), 2010 10th IEEE-RAS International Conference on, Nashville TN, United States, Dec.*, 2010. [Online]. Available: https://ci.nii.ac.jp/naid/10031099795/en/.

[8] H. Zhang and L. Zhang, "Cloud robotics architecture: Trends and challenges," in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 2019, pp. 362–3625.

[9] K. Siau and W. Wang, "Building trust in artificial intelligence, machine learning, and robotics," *Cutter Business Technology Journal*, vol. 31, no. 2, pp. 47–53, 2018.

[10] N. Sünderhauf, O. Brock, W. Scheirer, R. Hadsell, D. Fox, J. Leitner, B. Upcroft, P. Abbeel, W. Burgard, M. Milford, and P. Corke, "The limits and potentials of deep learning for robotics," *The International Journal of Robotics Research*, vol. 37, no. 4-5, pp. 405–420, 2018. DOI: 10.1177/0278364918770733. eprint: https://doi.org/10.1177/0278364918770733. [Online]. Available: https://doi.org/10.1177/0278364918770733.

[11] W. Wang and K. Siau, "Artificial intelligence, machine learning, automation, robotics, future of work and future of humanity: A review and research agenda," *J. Database Manag.*, vol. 30, pp. 61–79, 2019.

[12] P. C. Sen, M. Hajra, and M. Ghosh, "Supervised classification algorithms in machine learning: A survey and review," in *Emerging Technology in Modelling and Graphics*, J. K. Mandal and D. Bhattacharya, Eds., Singapore: Springer Singapore, 2020, pp. 99–111, ISBN: 978-981-13-7403-6.

[13] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, pp. 50–58, Apr. 2010. DOI: 10.1145/1721654.1721672.

[14] K. Goldberg, M. Mascha, S. Gentner, N. Rothenberg, C. Sutter, and J. Wiegley, "Desktop teleoperation via the world wide web," in *Proceedings of 1995 IEEE International Conference on Robotics and Automation*, vol. 1, May 1995, 654–659 vol.1. DOI: 10.1109/ROBOT.1995.525358.

[15] O. Saha and R. Dasgupta, "A comprehensive survey of recent trends in cloud robotics architectures and applications," *Robotics*, vol. 7, Aug. 2018. DOI: 10.3390/robotics7030047.

[16] *Getting started with matlab, simulink, and ros » racing lounge - matlab & simulink*, https://blogs.mathworks.com/racing-lounge/2017/11/08/matlab-simulink-ros/, (Accessed on 03/18/2021).

[17] *Willow garage*, http://www.willowgarage.com/, (Accessed on 11/28/2018).

[18] M. Quigley, B. P. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "Ros : An open-source robot operating system," 2009.

[19] A. Koubaa, *Robot Operating System (ROS): The Complete Reference (Volume 1)*, 1st. Springer Publishing Company, Incorporated, 2016, ISBN: 3319260529.

[20] C. Crick, G. Jay, S. Osentoski, B. Pitzer, and O. C. Jenkins, "Rosbridge: Ros for non-ros users," in *Robotics Research : The 15th International Symposium ISRR*, H. I. Christensen and O. Khatib, Eds. Cham: Springer International Publishing, 2017, pp. 493–504, ISBN: 978-3-319-29363-9. DOI: 10.1007/978-3-319-29363-9_28. [Online]. Available: https://doi.org/10.1007/978-3-319-29363-9-28.

[21] V. Wang, F. Salim, and P. Moskovits, "The websocket protocol," in *The Definitive Guide to HTML5 WebSocket*. Berkeley, CA: Apress, 2013, pp. 33–60, ISBN: 978-1-4302-4741-8. DOI: 10.1007/978-1-4302-4741-8_3. [Online]. Available: https://doi.org/10.1007/978-1-4302-4741-8-3.

[22] R. Buyya, "Market-oriented cloud computing: Vision, hype, and reality of delivering computing as the 5th utility," in *2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 2009, pp. 1–1. DOI: 10.1109/CCGRID.2009.97.

[23] *Dropbox*, https://www.dropbox.com/?landing=dbv2, (Accessed on 11/28/2018).

[24] *Salesforce.com: The customer success platform to grow your business*, https://www.salesforce.com/, (Accessed on 11/28/2018).

[25] *Robots as a service: A technology trend every business must consider*, https://www.forbes.com/sites/bernardmarr/2019/08/05/robots-as-a-service-a-technology-trend-every-business-must-consider/#6a59d1fe24ea, (Accessed on 03/27/2020).

[26] *Cloud service models (iaas, paas, saas) diagram | david chou*, https://dachou.github.io/2018/09/28/cloud-service-models.html, (Accessed on 03/18/2021).

[27] I. Parvez, A. Rahmati, I. Guvenc, A. I. Sarwat, and H. Dai, "A survey on low latency towards 5g: Ran, core network and caching solutions," *IEEE Communications Surveys Tutorials*, vol. 20, no. 4, pp. 3098–3130, 2018.

[28] M. Maier, M. Chowdhury, B. P. Rimal, and D. P. Van, "The tactile internet: Vision, recent progress, and open challenges," *IEEE Communications Magazine*, vol. 54, no. 5, pp. 138–145, May 2016, ISSN: 0163-6804. DOI: 10.1109/MCOM.2016.7470948.

[29] S. Nag, I. Heffan, A. Saenz-Otero, and M. Lydon, "Spheres zero robotics software development: Lessons on crowdsourcing and collaborative competition," in *2012 IEEE Aerospace Conference*, Mar. 2012, pp. 1–17. DOI: 10.1109/AERO.2012.6187452.

[30] M. Waibel, M. Beetz, J. Civera, R. D'Andrea, J. Elfring, D. Gálvez-López, K. Häussermann, R. Janssen, J. M. M. Montiel, A. Perzylo, B. Schießle, M. Tenorth, O. Zweigle, and R. V. D. Molengraft, "Roboearth," *IEEE Robotics Automation Magazine*, vol. 18, no. 2, pp. 69–82, Jun. 2011, ISSN: 1070-9932. DOI: 10.1109/MRA.2011.941632.

[31] S. A. Miratabzadeh, N. Gallardo, N. Gamez, K. Haradi, A. R. Puthussery, P. Rad, and M. Jamshidi, "Cloud robotics: A software architecture: For heterogeneous large-scale autonomous robots," in *2016 World Automation Congress (WAC)*, 2016, pp. 1–6.

[32] J. Quintas, P. Nunes, and J. M. M. Dias, "Cloud robotics : Towards context aware robotic networks," 2011.

[33] B. P. Rimal, E. Choi, and I. Lumb, "A taxonomy and survey of cloud computing systems," in *2009 Fifth International Joint Conference on INC, IMS and IDC*, Aug. 2009, pp. 44–51. DOI: 10.1109/NCM.2009.218.

[34] O. Saha and P. Dasgupta, "A comprehensive survey of recent trends in cloud robotics architectures and applications," *Robotics*, vol. 7, no. 3, 2018, ISSN: 2218-6581. DOI: 10.3390/robotics7030047. [Online]. Available: https://www.mdpi.com/2218-6581/7/3/47.

[35] C. Lai, H. Wang, H. Chao, and G. Nan, "A network and device aware qos approach for cloud-based mobile streaming," *IEEE Transactions on Multimedia*, vol. 15, no. 4, pp. 747–757, Jun. 2013, ISSN: 1520-9210. DOI: 10.1109/TMM.2013.2240270.

[36] M. Sato, K. Kamei, S. Nishio, and N. Hagita, "The ubiquitous network robot platform: Common platform for continuous daily robotic services," in *2011 IEEE/SICE International Symposium on System Integration (SII)*, Dec. 2011, pp. 318–323. DOI: 10.1109/SII.2011.6147467.

[37] B. Kehoe, A. Matsukawa, S. Candido, J. Kuffner, and K. Goldberg, "Cloud-based robot grasping with the google object recognition engine," *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 4263–4270, May 2013. DOI: 10.1109/ICRA.2013.6631180.

[38] J. Quintas, P. Menezes, and J. Dias, "Interoperability in cloud robotics — developing and matching knowledge information models for heterogenous multi-robot systems," in *2017 26th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*, 2017, pp. 1291–1296.

[39] K. Popovic and Z. Hocenski, "Cloud computing security issues and challenges," in *The 33rd International Convention MIPRO*, May 2010, pp. 344–349.

[40] S. Jain and R. Doriya, "Security issues and solutions in cloud robotics: A survey," in *Next Generation Computing Technologies on Computational Intelligence*, M. Prateek, D. Sharma, R. Tiwari, R. Sharma, K. Kumar, and N. Kumar, Eds., Singapore: Springer Singapore, 2019, pp. 64–76, ISBN: 978-981-15-1718-1.

[41] D. Flanagan and P. Ferguson, *JavaScript: The Definitive Guide*, 3rd. USA: O'Reilly & Associates, Inc., 1998, ISBN: 1565923928.

[42] M. Inaba, S. Kagami, F. Kanehiro, Y. Hoshino, and H. Inoue, "A platform for robotics research based on the remote-brained robot approach," *The International Journal of Robotics Research*, vol. 19, no. 10, pp. 933–954, 2000. DOI: 10.1177/02783640022067878. eprint: https://doi.org/10.1177/02783640022067878. [Online]. Available: https://doi.org/10.1177/02783640022067878.

[43] 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, Apr. 2015, ISSN: 1545-5955. DOI: 10.1109/TASE.2014.2329556.

[44] R. Arumugam, V. R. Enti, L. Bingbing, W. Xiaojun, K. Baskaran, F. F. Kong, A. S. Kumar, K. D. Meng, and G. W. Kit, "Davinci: A cloud computing framework for service robots," in *2010 IEEE International Conference on Robotics and Automation*, May 2010, pp. 3084–3089. DOI: 10.1109/ROBOT.2010.5509469.

[45] R. Doriya, P. Chakraborty, and G. C. Nandi, "'robot-cloud': A framework to assist heterogeneous low cost robots," in *2012 International Conference on Communication, Information Computing Technology (ICCICT)*, Oct. 2012, pp. 1–5. DOI: 10.1109/ICCICT.2012.6398208.

[46] Y. Li, H. Wang, B. Ding, and W. Zhou, "Robocloud: Augmenting robotic visions for open environment modeling using internet knowledge," *Science China Information Sciences*, vol. 61, no. 5, Apr. 2018, ISSN: 1869-1919. DOI: 10.1007/s11432-017-9380-5.

[47] L. Riazuelo, J. Civera, and J. Montiel, "C2tam: A cloud framework for cooperative tracking and mapping," *Robotics and Autonomous Systems*, vol. 62, no. 4, pp. 401–413, 2014, ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2013.11.007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0921889013002248.

[48] L. Muratore, B. Lennox, and N. Tsagarakis, "Xbotcloud: A scalable cloud computing infrastructure for xbot powered robots," Oct. 2018. DOI: 10.1109/IROS.2018.8593587.

[49] 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)*, May 2017, pp. 1615–1622. DOI: 10.1109/ICRA.2017.7989192.

[50] R. Toris, J. Kammerl, D. V. Lu, J. Lee, O. C. Jenkins, S. Osentoski, M. Wills, and S. Chernova, "Robot web tools: Efficient messaging for cloud robotics," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Sep. 2015, pp. 4530–4537. DOI: 10.1109/IROS.2015.7354021.

[51] S. Tilkov and S. Vinoski, "Node.js: Using javascript to build high-performance network programs," *IEEE Internet Computing*, vol. 14, no. 6, pp. 80–83, Nov. 2010, ISSN: 1089-7801. DOI: 10.1109/MIC.2010.145.

[52] B. Balaguer, S. Carpin, and S. Balakirsky, “Towards quantitative comparisons of robot algorithms: Experiences with slam in simulation and real world systems,” in *In: Workshop on Performance Evaluation and Benchmarking for Intelligent Robots and Systems at IEEE/RSJ IROS (2007)*.

[53] *Netdata - get control of your linux servers. simple. effective. awesome.* https://www.netdata.cloud/.

[54] H. Everett, *Sensors for mobile robots*. CRC Press, 1995.

[55] J. Abella, M. Padilla, J. D. Castillo, and F. J. Cazorla, “Measurement-based worst-case execution time estimation using the coefficient of variation,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 22, no. 4, Jun. 2017, ISSN: 1084-4309. DOI: 10.1145/3065924. [Online]. Available: https://doi.org/10.1145/3065924.

[56] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, *Continuous control with deep reinforcement learning*, 2019. arXiv: 1509.02971 [cs.LG].

[57] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, “Deep reinforcement learning: A brief survey,” *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017. DOI: 10.1109/MSP.2017.2743240.

[58] H. Jiang, H. Wang, W. .-Y. Yau, and K. .-W. Wan, “A brief survey: Deep reinforcement learning in mobile robot navigation,” in *2020 15th IEEE Conference on Industrial Electronics and Applications (ICIEA)*, 2020, pp. 592–597. DOI: 10.1109/ICIEA48937.2020.9248288.

[59] W. Zhao, J. P. Queralta, and T. Westerlund, “Sim-to-real transfer in deep reinforcement learning for robotics: A survey,” in *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, 2020, pp. 737–744. DOI: 10.1109/SSCI47803.2020.9308468.

[60] P. Mach and Z. Becvar, “Mobile edge computing: A survey on architecture and computation offloading,” *IEEE Communications Surveys Tutorials*, vol. 19, no. 3, pp. 1628–1656, 2017. DOI: 10.1109/COMST.2017.2682318.

[61] H. Wu, “Multi-objective decision-making for mobile cloud offloading: A survey,” *IEEE Access*, vol. 6, pp. 3962–3976, 2018. DOI: 10.1109/ACCESS.2018.2791504.

[62] C. Jiang, X. Cheng, H. Gao, X. Zhou, and J. Wan, “Toward computation offloading in edge computing: A survey,” *IEEE Access*, vol. 7, pp. 131 543–131 558, 2019. DOI: 10.1109/ACCESS.2019.2938660.

[63] M. Penmetcha, S. Sundar Kannan, and B.-C. Min, “Smart cloud: Scalable cloud robotic architecture for web-powered multi-robot applications,” in *2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2020, pp. 2397–2402. DOI: 10.1109/SMC42975.2020.9283148.

[64] L. Muratore, B. Lennox, and N. Tsagarakis, “Xbotcloud: A scalable cloud computing infrastructure for xbot powered robots,” Oct. 2018. DOI: 10.1109/IROS.2018.8593587.

[65] G. Mohanarajah, V. Usenko, M. Singh, R. D’Andrea, and M. Waibel, “Cloud-based collaborative 3d mapping in real-time with low-cost robots,” *IEEE Transactions on Automation Science and Engineering*, vol. 12, no. 2, pp. 423–431, 2015. DOI: 10.1109/TASE.2015.2408456.

[66] A. Rahman, J. Jin, A. Cricenti, A. Rahman, and M. Panda, “Motion and connectivity aware offloading in cloud robotics via genetic algorithm,” in *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, 2017, pp. 1–6. DOI: 10.1109/GLOCOM.2017.8255040.

[67] L. Wang, M. Liu, and M. Q. Meng, “A hierarchical auction-based mechanism for real-time resource allocation in cloud robotic systems,” *IEEE Transactions on Cybernetics*, vol. 47, no. 2, pp. 473–484, 2017. DOI: 10.1109/TCYB.2016.2519525.

[68] Z. Hong, H. Huang, S. Guo, W. Chen, and Z. Zheng, “Qos-aware cooperative computation offloading for robot swarms in cloud robotics,” *IEEE Transactions on Vehicular Technology*, vol. 68, no. 4, pp. 4027–4041, Apr. 2019, ISSN: 1939-9359. DOI: 10.1109/TVT.2019.2901761.

[69] D. Spatharakis, M. Avgeris, N. Athanasopoulos, D. Dechouniotis, and S. Papavassiliou, “A switching offloading mechanism for path planning and localization in robotic applications,” in *2020 International Conferences on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics)*, 2020, pp. 77–84. DOI: 10.1109/iThings-GreenCom-CPSCom-SmartData-Cybermatics50389.2020.00031.

[70] Q. Pham, F. Fang, V. N. Ha, M. J. Piran, M. Le, L. B. Le, W. Hwang, and Z. Ding, “A survey of multi-access edge computing in 5g and beyond: Fundamentals, technology integration, and state-of-the-art,” *IEEE Access*, vol. 8, pp. 116 974–117 017, 2020. DOI: 10.1109/ACCESS.2020.3001277.

[71] M. Min, L. Xiao, Y. Chen, P. Cheng, D. Wu, and W. Zhuang, “Learning-based computation offloading for iot devices with energy harvesting,” *IEEE Transactions on Vehicular Technology*, vol. 68, no. 2, pp. 1930–1941, 2019. DOI: 10.1109/TVT.2018.2890685.

[72] H. Guo, J. Liu, J. Zhang, W. Sun, and N. Kato, "Mobile-edge computation offloading for ultradense iot networks," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4977–4988, 2018. DOI: 10.1109/JIOT.2018.2838584.

[73] A. Yousefpour, G. Ishigaki, R. Gour, and J. P. Jue, "On reducing iot service delay via fog offloading," *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 998–1010, 2018. DOI: 10.1109/JIOT.2017.2788802.

[74] J. Shuja, K. Bilal, W. Alasmary, H. Sinky, and E. Alanazi, "Applying machine learning techniques for caching in next-generation edge networks: A comprehensive survey," *Journal of Network and Computer Applications*, vol. 181, p. 103 005, 2021, ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2021.103005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804521000321.

[75] E. Ahmed, A. Ahmed, I. Yaqoob, J. Shuja, A. Gani, M. Imran, and M. Shoaib, "Bringing computation closer toward the user network: Is edge computing the solution?" *IEEE Communications Magazine*, vol. 55, no. 11, pp. 138–144, 2017. DOI: 10.1109/MCOM.2017.1700120.

[76] X. Qiu, W. Zhang, W. Chen, and Z. Zheng, "Distributed and collective deep reinforcement learning for computation offloading: A practical perspective," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 5, pp. 1085–1101, 2021. DOI: 10.1109/TPDS.2020.3042599.

[77] X. Qiu, L. Liu, W. Chen, Z. Hong, and Z. Zheng, "Online deep reinforcement learning for computation offloading in blockchain-empowered mobile edge computing," *IEEE Transactions on Vehicular Technology*, vol. 68, no. 8, pp. 8050–8062, 2019. DOI: 10.1109/TVT.2019.2924015.

[78] M. Tang and V. S. Wong, "Deep reinforcement learning for task offloading in mobile edge computing systems," *IEEE Transactions on Mobile Computing*, no. 01, pp. 1–1, Nov. 5555, ISSN: 1558-0660. DOI: 10.1109/TMC.2020.3036871.

[79] J. Wang, J. Hu, G. Min, A. Y. Zomaya, and N. Georgalas, "Fast adaptive task offloading in edge computing based on meta reinforcement learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 1, pp. 242–253, 2021. DOI: 10.1109/TPDS.2020.3014896.

[80] Y. Dai, K. Zhang, S. Maharjan, and Y. Zhang, "Edge intelligence for energy-efficient computation offloading and resource allocation in 5g beyond," *IEEE Transactions on Vehicular Technology*, vol. 69, no. 10, pp. 12 175–12 186, 2020. DOI: 10.1109/TVT.2020.3013990.

[81] S. Chinchali, A. Sharma, J. Harrison, A. Elhafsi, D. Kang, E. Pergament, E. Cidon, S. Katti, and M. Pavone, "Network offloading policies for cloud robotics: A learning-based approach," *CoRR*, vol. abs/1902.05703, 2019. arXiv: 1902.05703. [Online]. Available: http://arxiv.org/abs/1902.05703.

[82] H. Liu, S. Liu, and K. Zheng, "A reinforcement learning-based resource allocation scheme for cloud robotics," *IEEE Access*, vol. 6, pp. 17 215–17 222, 2018. DOI: 10.1109/ACCESS.2018.2814606.

[83] Z. Peng, J. Lin, D. Cui, Q. Li, and J. He, "A multi-objective trade-off framework for cloud resource scheduling based on the deep q-network algorithm," *Cluster Computing*, pp. 1–15, 2020.

[84] A. Valmari, "The state explosion problem," in *Lectures on Petri Nets I: Basic Models: Advances in Petri Nets*, W. Reisig and G. Rozenberg, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 429–528, ISBN: 978-3-540-49442-3. DOI: 10.1007/3-540-65306-6_21. [Online]. Available: https://doi.org/10.1007/3-540-65306-6_21.

[85] D. S. Hochba, "Approximation algorithms for np-hard problems," *SIGACT News*, vol. 28, no. 2, pp. 40–52, Jun. 1997, ISSN: 0163-5700. DOI: 10.1145/261342.571216. [Online]. Available: https://doi.org/10.1145/261342.571216.

[86] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013. arXiv: 1312.5602. [Online]. Available: http://arxiv.org/abs/1312.5602.

[87] K. Cho, Y. Sung, and K. Um, "A production technique for a q-table with an influence map for speeding up q-learning," in *The 2007 International Conference on Intelligent Pervasive Computing (IPC 2007)*, 2007, pp. 72–75.

[88] H. Zhu, I. C. Paschalidis, and M. E. Hasselmo, "Feature extraction in q-learning using neural networks," in *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, 2017, pp. 3330–3335.

[89] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, Nov. 2017, ISSN: 1053-5888. DOI: 10.1109/msp.2017.2743240. [Online]. Available: http://dx.doi.org/10.1109/MSP.2017.2743240.

[90] S. Gu, T. Lillicrap, I. Sutskever, and S. Levine, "Continuous deep q-learning with model-based acceleration," in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ser. ICML'16, New York, NY, USA: JMLR.org, 2016, pp. 2829–2838.

[91] C. J. C. H. Watkins and P. Dayan, "Q-learning," in *Machine Learning*, 1992, pp. 279–292.

[92] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018, ISBN: 0262039249.

[93] C. Chicone, "Stability theory of ordinary differential equations," in *Mathematics of Complexity and Dynamical Systems*, R. A. Meyers, Ed. New York, NY: Springer New York, 2011, pp. 1653–1671, ISBN: 978-1-4614-1806-1. DOI: 10.1007/978-1-4614-1806-1_106. [Online]. Available: https://doi.org/10.1007/978-1-4614-1806-1_106.

[94] J. Li, H. Gao, T. Lv, and Y. Lu, "Deep reinforcement learning based computation offloading and resource allocation for mec," in *2018 IEEE Wireless Communications and Networking Conference (WCNC)*, 2018, pp. 1–6.

[95] *Gazebo*, http://gazebosim.org/, (Accessed on 01/07/2021).

[96] *Simulating jackal — jackal tutorials 0.5.4 documentation*, https://www.clearpathrobotics.com/assets/guides/kinetic/jackal/simulation.html, (Accessed on 01/05/2021).

[97] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[98] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968. DOI: 10.1109/tssc.1968.300136. [Online]. Available: https://doi.org/10.1109/tssc.1968.300136.

[99] P. Raja and S. Pugazhenthi, "Optimal path planning of mobile robots: A review," *International Journal of the Physical Sciences*, vol. 7, Feb. 2012. DOI: 10.5897/IJPS11.1745.

[100] A. V. Goldberg and R. E. Tarjan, "Expected performance of dijkstra's shortest path algorithm," *NEC Research Institute Report*, 1996.

[101] *Ros notes: Map resolution – new screwdriver*, https://newscrewdriver.com/2018/09/21/ros-notes-map-resolution/, (Accessed on 01/12/2021).

[102] *Zeromq*, https://zeromq.org/, (Accessed on 01/13/2021).

[103] L. K. Hansen and P. Salamon, "Neural network ensembles," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 12, no. 10, pp. 993–1001, 1990. DOI: 10.1109/34.58871.

[104] O. I. Abiodun, A. Jantan, A. E. Omolara, K. V. Dada, N. A. Mohamed, and H. Arshad, "State-of-the-art in artificial neural network applications: A survey," *Heliyon*, vol. 4, no. 11, e00938, 2018, ISSN: 2405-8440. DOI: https://doi.org/10.1016/j.heliyon.2018.e00938. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2405844018332067.

[105] A. D. Tijsma, M. M. Drugan, and M. A. Wiering, "Comparing exploration strategies for q-learning in random stochastic mazes," in *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, 2016, pp. 1–8. DOI: 10.1109/SSCI.2016.7849366.

[106] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[107] H. van Hasselt, A. Guez, and D. Silver, *Deep reinforcement learning with double q-learning*, 2015. arXiv: 1509.06461 [cs.LG].

[108] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, *Dueling network architectures for deep reinforcement learning*, 2016. arXiv: 1511.06581 [cs.LG].

[109] H. Lu, C. Gu, F. Luo, W. Ding, and X. Liu, "Optimization of lightweight task offloading strategy for mobile edge computing based on deep reinforcement learning," *Future Generation Computer Systems*, vol. 102, pp. 847–861, 2020, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2019.07.019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X19308209.

[110] Y. Miao, G. Wu, M. Li, A. Ghoneim, M. Al-Rakhami, and M. S. Hossain, "Intelligent task prediction and computation offloading based on mobile-edge cloud computing," *Future Generation Computer Systems*, vol. 102, pp. 925–931, 2020, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2019.09.035. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X19320862.

[111] *The fundamentals of the big-o notation | by ruben winastwan | towards data science*, https://towardsdatascience.com/the-fundamentals-of-the-big-o-notation-7fe14210b675, (Accessed on 02/15/2021).

[112] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv*, 2018.

[113] W. Chen, Y. Yaguchi, K. Naruse, Y. Watanobe, K. Nakamura, and J. Ogawa, "A study of robotic cooperation in cloud robotics: Architecture and challenges," *IEEE Access*, vol. 6, pp. 36 662–36 682, 2018. DOI: 10.1109/ACCESS.2018.2852295.

[114] 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. DOI: 10.1109/ACCESS.2016.2574979.

[115] A. Rahman, J. Jin, A. Rahman, A. Cricenti, M. Afrin, and Y.-n. Dong, "Energy-efficient optimal task offloading in cloud networked multi-robot systems," *Computer Networks*, vol. 160, pp. 11–32, 2019, ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2019.05.016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128619306371.

[116] A. A. Alli and M. M. Alam, "Secoff-fciot: Machine learning based secure offloading in fog-cloud of things for smart city applications," *Internet of Things*, vol. 7, p. 100 070, 2019, ISSN: 2542-6605. DOI: https://doi.org/10.1016/j.iot.2019.100070. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2542660518301938.

[117] X. Chen, L. Jiao, W. Li, and X. Fu, "Efficient multi-user computation offloading for mobile-edge cloud computing," *IEEE/ACM Transactions on Networking*, vol. 24, no. 5, pp. 2795–2808, 2016. DOI: 10.1109/TNET.2015.2487344.

[118] C. You, K. Huang, H. Chae, and B. Kim, "Energy-efficient resource allocation for mobile-edge computation offloading," *IEEE Transactions on Wireless Communications*, vol. 16, no. 3, pp. 1397–1411, 2017. DOI: 10.1109/TWC.2016.2633522.

[119] Y. Mao, J. Zhang, and K. B. Letaief, "Dynamic computation offloading for mobile-edge computing with energy harvesting devices," *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 12, pp. 3590–3605, 2016. DOI: 10.1109/JSAC.2016.2611964.

[120] D. C. Montgomery, E. A. Peck, and G. G. Vining, *Introduction to linear regression analysis*. John Wiley &amp; Sons, 2021.

[121] V. Svetnik, A. Liaw, C. Tong, J. C. Culberson, R. P. Sheridan, and B. P. Feuston, "Random forest: A classification and regression tool for compound classification and qsar modeling," *Journal of Chemical Information and Computer Sciences*, vol. 43, no. 6, pp. 1947–1958, 2003, PMID: 14632445. DOI: 10.1021/ci034160g. eprint: https://doi.org/10.1021/ci034160g. [Online]. Available: https://doi.org/10.1021/ci034160g.

[122] W. W. Wei, "Time series analysis," in *The Oxford Handbook of Quantitative Methods in Psychology: Vol. 2*, 2006.

[123] A. Likas, N. Vlassis, and J. J. Verbeek, "The global k-means clustering algorithm," *Pattern Recognition*, vol. 36, no. 2, pp. 451–461, 2003, Biometrics, ISSN: 0031-3203. DOI: https://doi.org/10.1016/S0031-3203(02)00060-2. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0031320302000602.

[124] E. Galceran and M. Carreras, "A survey on coverage path planning for robotics," *Robotics and Autonomous Systems*, vol. 61, no. 12, pp. 1258–1276, 2013, ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2013.09.004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S092188901300167X.

[125] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numer. Math.*, vol. 1, no. 1, pp. 269–271, Dec. 1959, ISSN: 0029-599X. DOI: 10.1007/BF01386390. [Online]. Available: https://doi.org/10.1007/BF01386390.

[126] *Introduction | ømq - the guide*, https://zguide.zeromq.org/, (Accessed on 02/26/2021).

[127] D. Freedman, R. Pisani, and R. Purves, "Statistics (international student edition)," *Pisani, R. Purves, 4th edn. WW Norton & Company, New York*, 2007.

[128] K. Kumar, J. Liu, Y.-H. Lu, and B. Bhargava, "A survey of computation offloading for mobile systems," *Mob. Netw. Appl.*, vol. 18, no. 1, pp. 129–140, Feb. 2013, ISSN: 1383-469X. DOI: 10.1007/s11036-012-0368-0. [Online]. Available: https://doi.org/10.1007/s11036-012-0368-0.

[129] A. Bhattacharya and P. De, "A survey of adaptation techniques in computation offloading," *Journal of Network and Computer Applications*, vol. 78, pp. 97–115, 2017, ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2016.10.023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804516302570.

A. APPENDIX

No	**Description**	**Link**
1	A video on full offloading solution proposed in Chapter 3.	https://youtu.be/zImysVWLlFs
2	A video on DRL algorithm for dynamic application offloading proposed in Chapter 4	https://youtu.be/JAwxaOH9BFk
3	A video based on predictive algorithm for dynamic application offloading proposed in Chapter 5	https://youtu.be/w3tniTpgjYY

VITA

Manoj Penmetcha received a B.S. degree in information technology from Osmania University, India in 2010, and a M.S. degree in computer and information technology from Purdue University, West Lafayette, IN, USA, in 2012, where he is currently pursuing a Ph.D. degree in technology.

His research interests include machine learning, multi-robot systems, cloud robotics, cloud computing, edge computing, and wireless networks.

Referred Publications

- **M. Penmetcha** and B.C. Min, "A Deep Reinforcement Learning-based Dynamic Computational Offloading Method for Cloud Robotics," in *IEEE Access*.
- **M. Penmetcha**, S. S. Kannan, and B.C. Min, "Smart cloud: Scalable cloud robotic architecture for web-powered multi-robot applications," in *2020 IEEE International Conference on Systems, Man and Cybernetics (SMC)*, Oct 2020, pp. 2397-2402.
- **M. Penmetcha**, S. Luo, A. Samantaray, J. E. Dietz, B. Yang, and B. Min, "Computer vision-based algae removal planner for multi-robot teams," in *2019 IEEE International Conference on Systems, Man and Cybernetics (SMC)*, Oct 2019, pp. 1575–1581.
- **M. Penmetcha**, A. Samantaray, and B.C. Min, "Smartresponse: Emergency and non-emergency response for smartphone based indoor localization applications," in *HCI International 2017 – Posters' Extended Abstracts*, C. Stephanidis, Ed. Cham: Springer International Publishing, 2017, pp. 398–404.
- C. Vieira, **M. Penmetcha**, A. Magana, and E. Matson, "Computational thinking as a practice of representation: A proposed learning and assessment framework," *Journal of Computational Science Education*, vol. 7, pp. 21–30, 2016.
- **M. Penmetcha**, "Exploring the effectiveness of robotics as a vehicle for computational thinking," Master's thesis, Purdue University, 2012.