FogROS: An Adaptive Framework for Automating Fog Robotics Deployment and Co-scheduling Feature Updates and Queries for Feature Stores

Yafei Liang
Joseph Gonzalez, Ed.
Joseph M. Hellerstein, Ed.

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-46
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-46.html

May 10, 2022

Acknowledgement

I would like to first thank the faculties who have helped me, in particular, Professor Joseph E. Gonzalez for advising me throughout the course of my work and for providing resources and guidance during my Master’s program; Professor Joseph M. Hellerstein for introducing me to the system research area and connecting me with other resourceful faculties. I am very grateful for the opportunity of working at RISE/SKY Lab and with other researchers at Berkeley. I have learned so much under your guidance.
I would also like to thank my friends, who have made my college life colorful. Lastly, I would not have ended up at where I am today without my parents, Bogui Liang and Tao Lu. Thank you for supporting me in every way and always being there for me.

FogROS: An Adaptive Framework for Automating Fog Robotics Deployment and Co-scheduling Feature Updates and Queries for Feature Stores

by

Yafei Liang

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Spring 2022

The dissertation of Yafei Liang, titled FogROS: An Adaptive Framework for Automating Fog Robotics Deployment and Co-scheduling Feature Updates and Queries for Feature Stores, is approved:

Chair	[illegible]	Date	April 28, 2022
	[illegible]	Date	April 28, 2022
		Date	

University of California, Berkeley

FogROS: An Adaptive Framework for Automating Fog Robotics Deployment and Co-scheduling Feature Updates and Queries for Feature Stores

Abstract

FogROS: An Adaptive Framework for Automating Fog Robotics Deployment and Co-scheduling Feature Updates and Queries for Feature Stores

by

Yafei Liang

Master of Science in Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Joseph E. Gonzalez, Chair

As many robot automation applications increasingly rely on multi-core processing or deep-learning models, cloud computing is becoming an attractive and economically viable resource for systems that do not contain high computing power onboard. Despite its immense computing capacity, it is often underused by the robotics and automation community due to lack of expertise in cloud computing and cloud-based infrastructure. Fog Robotics balances computing and data between cloud edge devices. We propose a software framework, FogROS, as an extension of the Robot Operating System (ROS), the de-facto standard for creating robot automation applications and components. It allows researchers to deploy components of their software to the cloud with minimal effort, and correspondingly gain access to additional computing resources and predeployed software made available by other researchers.

To accommodate the real-time update requirements for many Machine Learning models and robotics applications, Feature Stores are fast emerging as a new class of Machine Learning system that maintains intermediate statistics of live data streams used for model training and inference to improve accuracy and save prediction time. Our work is based on RALF, a feature store designed for streaming data and explicitly leverages downstream feedback. Our project explores the impact of *lazy evaluation*, which postpones feature updates in a feature store, on the three most important aspects of feature stores (i.e., staleness, latency, and costs) and builds an *SLO-aware featurization scheduler* that reduces the staleness of the queried features by co-scheduling feature updates and query responses.

To my parents, Bogui Liang and Tao Lu.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to first thank the faculties who have helped me, in particular, Professor Joseph E. Gonzalez for advising me throughout the course of my work and for providing resources and guidance during my Master's program; Professor Joseph M. Hellerstein for introducing me to the system research area and connecting me with other resourceful faculties. I am very grateful for the opportunity of working at RISE/SKY Lab and with other researchers at Berkeley. I have learned so much under your guidance.

I would also like to thank my friends, who have made my college life colorful. Lastly, I would not have ended up at where I am today without my parents, Bogui Liang and Tao Lu. Thank you for supporting me in every way and always being there for me.

Chapter 1

FogROS: An Adaptive Framework for Automating Fog Robotics Deployment

1.1 Acknowledgements

This research was performed at the AUTOLAB at UC Berkeley in affiliation with the Berkeley AI Research (BAIR) Lab, and the CITRIS "People and Robots" (CPAR) Initiative. The authors were supported in part by donations from Google, Siemens, Toyota Research Institute, Autodesk, Honda, Intel, Hewlett-Packard, and VMWare. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE 1752814 and NSF/VMware Partnership on Edge Computing Data Infrastructure (ECDI), NSF award 1838833. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

I thank my colleagues who helped immensely in the course of this work, in particular:

- Kaiyuan (Eric) Chen (co-author of this paper), for his immense help with the implementation of the FogROS system.
- Nikhil Jha (co-author of this paper), for helping us with the Virtual Private Cloud option for FogROS networking.
- Jeffrey Ichnowski (co-author of this paper), for providing the motion planning workload to the FogROS project.
- Michael Danielczuk (co-author of this paper), for providing the grasp planning workload to the FogROS project.
- Joseph Gonzalez (co-author of this paper), for commenting on our paper draft, providing valuable feedbacks, and attending our project meetings.

- John Kubiatowicz (co-author of this paper), for providing system insights to this project.
- Ken Goldberg (co-author of this paper), for providing revision advice to the this project and connecting us with experienced and helpful researchers.
- Joseph Hellerstein, for providing advice during the FogROS project presentation from a database researcher's perspective.
- Alvin Cheung, for providing advice during the FogROS project presentation from a database researcher's perspective.

1.2 Introduction

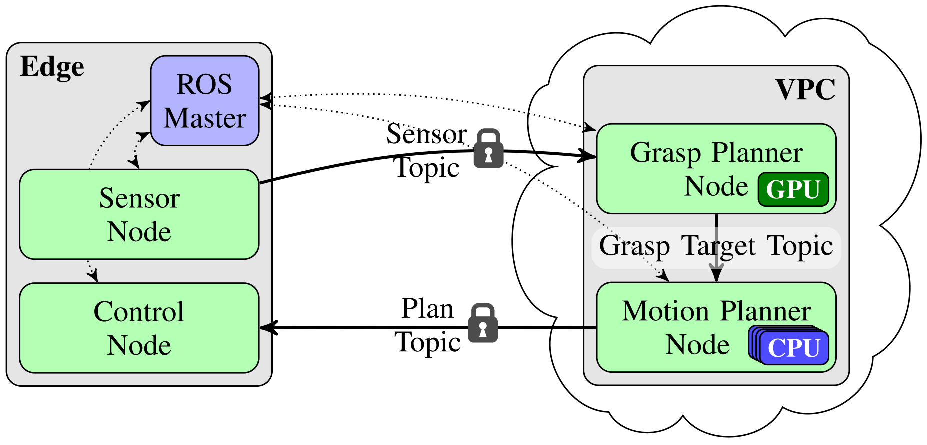
Power, weight, and cost considerations often mean robots do not include computing capabilities capable of running large-scale multi-core CPU-based, graphics processing unit (GPU)-based, field-programmable gate array (FPGA)-based, or tensor processing unit (TPU)-based algorithms. For example, a light-weight drone with an attached gripper that uses a GPU-based grasp-planning module to compute grasp points for picking up objects [4] or perching [48], requires access to a GPU that the drone would not have onboard. While nearby computers can provide the necessary computing capabilities, this practice can be complex to set up, scale, and is prone to over-provisioning. Instead, we propose a framework based on the *Fog Robotics* [24, 40, 60] idea of balancing between the compute available at the edge and in the cloud. This framework, *FogROS*, is an extension of the Robot Operating System (ROS) [47] that, with minimal effort, allows researchers to deploy components of their software to the cloud, and correspondingly gain access to additional computing cores, GPUs, FPGAs, and TPUs, as well as predeployed software made available by other researchers.

ROS, at its core, is a platform in which software components (*nodes*) communicate with each other via a publication/subscription (pub/sub) system. Individual nodes can publish messages to named *topics* and subscribe to other named topics to get messages published by other nodes. In practice, these nodes all run on the robot and perhaps a nearby computer. For example, on a robot, a sensor node publishes to a sensor topic, a planning node subscribes to the sensor topic and compute a plan based on the sensor messages, and then publishes messages that another node uses to execute the plan (Fig. 1.1).

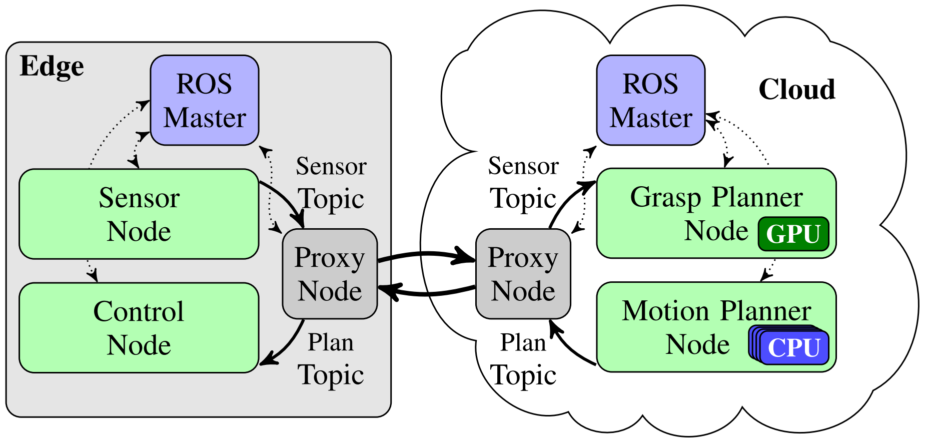
With FogROS, a researcher can use the same code, and make a small change to a configuration file to select components of the edge computer software to deploy to cloud-based computers. On launch, FogROS provisions a cloud-based computer, deploys the nodes to it, and then transparently passes the pub/sub communication between the edge computer and the cloud. The only observable differences are: (1) the pub/sub latency increases, and (2) the cloud-deployed components can compute faster given the additional computing resources. The increased latency means that not all components will benefit from being deployed to the cloud, in particular, any component with real-time requirements (e.g., a motor controller) or any component that requires little computing power, should not be deployed to the cloud. On the other hand, for many applications, the increased computation speed may enable new robot capabilities, speed up tasks, and allow for higher accuracy in tasks such as object detection or segmentation due to the use of larger models.

FogROS also supports launching pre-built automation container images. These container images contain all the software and dependencies required to run a program. To date, many academic and industrial open-source communities leverage container services, such as Docker [13], to distribute their applications. FogROS can deploy robot automation containers to the cloud without explicitly configuring the environment and hardware, facilitating ease of containerized software reuse.

This paper makes three contributions: (1) FogROS, an open-source extension to ROS that allows user-friendly and adaptive deployment of software components to cloud-based comput-



(a) FogROS Application on VPC



(b) FogROS Application with Proxy

Figure 1.1: Synthetic Workflow with a server and a client

ers; (2) a method to pre-deploy containerized FogROS software that allows commonly-used software to be quickly integrated into applications; and (3) application examples evaluating the performance of FogROS deployment.

Design Principles

FogROS aims to adhere to the following design principles:

Transparent to software FogROS should preserve ROS abstractions and interfaces. Applications should notice no difference between cloud-deployed and on-board nodes (other than the latency of message processing).

Flexible computing resources Different nodes require different computing capabilities. Some nodes benefit from additional computing cores, while others benefit from access to a GPU. FogROS should make selecting the appropriate configuration simple.

Minimal configuration required Running software nodes on a cloud-based computer should be as easy as running them on the edge computer.

Pre-deployed nodes Some useful nodes require extensive setup, installation of a dependency structure, and may have conflicting dependency versions. FogROS should make it possible to use pre-deployed containerized software through configuration.

Flexible Networking Different networking options may have different availability, performance, setup time, and costs associated. FogROS should allow the user to select the networking options best suited to their application.

Security and Isolation The communication between cloud and on-board nodes should be secure, and FogROS should close ports that expose software to compromise.

1.3 Related Work

Cloud computing has emerged as an attractive and economically viable [27] resource to offload computation for robot automation systems with minimal onboard computing power. Kehoe *et al.* [29] survey the capabilities, research potential, and challenges of cloud robotics, as well as applications such as grasp planning, motion planning, and collective robot learning, that might benefit from the computational power of the cloud. Grasp planning and motion planning have both shown to be amenable to cloud computation. Ben *et al.* [7], Nan *et al.* [40], and Li *et al.* [32] generate robot grasp poses in the cloud by implementing parallelizable Monte-Carlo sampling of grasp perturbations [28, 6, 30] while Mahler *et al.* [36] explore cloud grasp pose computation that maintains privacy of proprietary geometries. In motion planning, Lam *et al.* [31] introduce path planning as a service (PPaaS) for on-demand path planning in the cloud and use Rapyuta to share plans among robots. Bekris *et al.* [5] and Ichnowski *et al.* [23] both devise methods for splitting motion planning computation between the cloud and the edge computer [24, 1]. In addition to providing computing resources, the cloud can also facilitate sharing and benchmarking of algorithms and models between edge computers [61] for grasping, motion planning, or computer vision.

To leverage cloud resources, many in academia and industry endeavor to connect edge computers to the cloud. Example approaches include using SSH port forwarding [19] or VPN-based proxying [33] to support unmodified ROS applications to share a single ROS master. FogROS builds on these approaches, and adds automation of ROS node deployment to the cloud and a virtual private cloud (VPC), saving time over prior approaches that require manual configuration of network access rules and IP addresses. For example, setting up a VPN-based proxying requires more than 12 steps for configuration and 37 steps for verification [19]. The complex manual configurations scale poorly and are error-prone. ROSRemote [46] and MSA [66] replace the ROS communication stack with custom Pub/Sub designs. Although edge computers can communicate with nodes owned by other ROS masters, these systems require heavy code changes to ROS applications. FogROS, as an option, leverages rosbridge [12], an open-source webserver that enables an edge computer to interact with another ROS environment with JSON queries. Given diverse attempts to connect edge computers to the cloud, Wan *et al.* [64] and Saha *et al.* [53] call for a unified and standardized framework to handle cloud-robot data interactions. FogROS aims to be a painless solution to this open issue by allowing unmodified ROS applications to be launched on the cloud with minimal additional configurations.

Sharing a similar vision as FogROS, RoboEarth [63] is a successful example where edge computers share information on the cloud. However, in their use cases, edge computers mainly use the shared database on the cloud, and do not benefit from the powerful cloud computing resources. Rapyuta [37] and AWS Greengrass [2] provide pipelines to deploy pre-built ROS nodes to edge computers or robots. Both platforms build ROS nodes or Docker images on the cloud, and push the built images to robots that are registered with their platforms. FogROS considers the reversed direction of Rapyuta and AWS Greengrass. Instead of pushing the computation from cloud to robots, FogROS is a lightweight platform that allows developers to rapidly prototype applications and gain quick access to extensive computing resources, without conforming to an additional framework.

1.4 Background

In this section, we provide a brief background on the building blocks of FogROS, including (A) cloud-based computing, (B) ROS and its pub/sub system, and (C) how ROS-based robotic systems are configured and launched.

Cloud Computing

Cloud-based computing services, such as Amazon Web Services (AWS), Google Cloud, and Microsoft Azure, offer network accessible computers of various specifications to be rented on a per-time-unit basis. Setting up a service typically requires a one-time registration and a credit card. Registered users can setup, reconfigure, turn on, turn off, and tear down virtual computers in the cloud. This can be done either through a web-browser interface, or programmatically through a network-based application programming interface (API). Computer configuration options include: amount of memory, amount of processing cores, type and amount of GPUs, and inclusion of custom processing hardware such as field-programmable gate arrays (FPGAs) and tensor processing units (TPUs). FogROS uses the AWS cloud service API to setup a cloud-based computer, deploy ROS and the code, secure network communications, and then run the node.

ROS and Pub/Sub

In ROS, *nodes* (software components) communicate with each other using a pub/sub (publication and subscription) system. Nodes register as publishers and/or subscribers to named communication channels called *topics*. Each topic has message type that determines what data is sent over the channel. For example, a ROS node that monitors the joint state (e.g., angles) through sensors, would publish messages of type JointState on an appropriately named topic, and that topic would only contain JointState messages. When a node publishes a sequence of messages to a topic, all registered subscribers will receive the message in the same sequence they were published.

Coordination of the publishers and subscribers to topics is maintained by the ROS *Master* [52]. The ROS Master exposes network API that allows nodes to connect over a network and register/unregister themselves as publishers and subscribers to topics. During the registration process, publishers get the current list of subscribers, and subscribers get the current list of publishers. Publishers may then connect directly to already-registered subscribers, and subscribers may connect directly to already-registered publishers.

Once a publishing and subscribing nodes are directly connected to each other[1], publishing nodes serialize message-specific data structure to a sequence of bytes and sends the bytes over the connection. When subscribing nodes receives the sequence of bytes, they deserializes the bytes to the message-specific data structure and process the message.

[1]As an implementation optimization, ROS nodes on the same machine can communicate via a shared-memory queue, instead of using a network.

However, existing ROS pub/sub communication has several limitations: (1) all the nodes have to share the same master to communicate (inter-master communication is not supported by ROS pub/sub protocol stack, and one has to use out-of-band protocols for communicating across masters); (2) although it is possible to join nodes from multiple machines to share a single master, the communication for ROS is not secured, and users must configure security protocols.

ROS Launch Scripts

Robot systems can be comprised of a complex graph of nodes communicating with each other via pub/sub. To consolidate an automation system deployment into a single file, ROS supports a launch configuration file. This file specifies which nodes are to be launched by code entry point, and allows for optional remapping of topic names (e.g., so that code written to process a standard message type can produce it from a topic with a name not known/specified when the code was written). Fig. 1.2 is an example launch script that launches a client node and a server node from the mpt_ros package locally.

```
<launch>
    <!-- Run client node -->
    <node name="client" pkg="mpt_ros" type="client"
         output="screen" />

    <!-- Run server node-->
    <node name="server" pkg="mpt_ros" type="server"
         output="screen" />
</launch>
```

Figure 1.2: ROS Launch Script Example

FogROS extends the launch script capabilities to allow the specification of which nodes to deploy in the cloud and on what machine type.

1.5 Approach

To meet the design principles, FogROS (1) extends ROS launch scripts to include an option of where to deploy and run a ROS node, the only place that requires user configurations; (2) provisions cloud-based computers, securely pushes the code or containers to them, and runs the code; (3) sets up one of two networking options (VPC or Proxy) to transparently and automatically proxy the pub/sub communication between the edge computer and the cloud; (4) provides introspection infrastructure for monitoring network conditions; and (5) supports launching containerized FogROS nodes from pre-built Docker images.

Launch Script Extensions

FogROS uses standard ROS launch scripts as the user interface. Users specify which nodes are to be deployed and what type of cloud computing instance is used in the same launch file as the nodes that users want to deploy locally. They can push multiple nodes to the cloud at the same time by providing the path to a separate launch script. FogROS parses the launch script, finds and collects all the packages in the script, and pushes them to the cloud computer. As part of the configuration process, users can optionally specify a bash script that installs dependencies outside of the FogROS launch process (e.g., mirroring the steps to install dependencies on the edge computer).

Fig. 1.3 provides an example of a FogROS launch script that serves the same functionality as Fig. 1.2, but with the server node running on a cloud computer. Local ROS nodes, such as client, are launched as before. With FogROS, the user specifies the launch file that contains the server node (server.launch), the type of cloud computer (c5.24xlarge), and optionally, a setup script (init.bash).

Cloud-Computer ROS Nodes

When FogROS launches cloud-based nodes, it performs the following sequence of steps that result in ROS nodes running on a cloud computer with messages being transparently proxied between the edge computer and the cloud computer:

1. Provision and start a cloud computer with the capabilities from the launch file and pre-loaded with ROS
2. Push code for ROS nodes to the cloud computer
3. Run the environment setup script
4. Set up secure networking (via Proxy or VPC)
5. Launch the pushed code

```
<launch>
    <!-- Run client node as before -->
    <node name="client" pkg="mpt_ros" type="client"
         output="screen" />

    <!-- Run server node w/ FogROS-->
    <node name="server" pkg="fogros" type="
        fogros_launch.py" >
        <rosparam>
            instance_type: c5.24xlarge
            launch_file: server.launch
            env_script: init.bash
        </rosparam>
    </node>
</launch>
```

Figure 1.3: FogROS Launch Script Example

Before FogROS provisions a cloud computer, it uses the cloud service provider API to create security rules to set up a secure computing infrastructure suitable for ROS application configuration. It closes network ports not needed for communication between nodes. Then it provisions the cloud computer with a specified location and type. To speed up the launching process, FogROS specifies an image pre-loaded with the core ROS libraries to run on the cloud computer. As part of the launch process, FogROS generates and installs secure credentials on the cloud computer, and gets its public internet protocol (IP) address.

Once the computer is started, using the IP address and secure credentials, FogROS recursively copies the ROS code to the cloud computer securely over a secure shell (SSH) [67] connection, optionally runs the user-specified setup script, and builds the code on the cloud. With the code ready to run, FogROS then starts the configured secure networking components for VPC (Sec. 1.5) or proxying (Sec. 1.5), and runs the ROS nodes in the cloud.

Networking: Virtual Private Cloud

To allow the edge computer and cloud-based computers to communicate securely with each other, FogROS automates the setup of a Virtual Private Cloud (VPC). A VPC secures point-to-point communication between cloud computers by assigning private IPs that are only accessible for other nodes within the VPC. FogROS creates a Virtual Private Network (VPN) between the edge computer and the VPC. A VPN is a secure network communication channel provided by the operating system. With this setup, from the perspective of a ROS node, all nodes appear as though they are on the same private network.

FogROS automates the setup of the VPC and the VPN when it provisions the cloud computers to run the ROS nodes, by using the cloud service providers API to: (1) create a

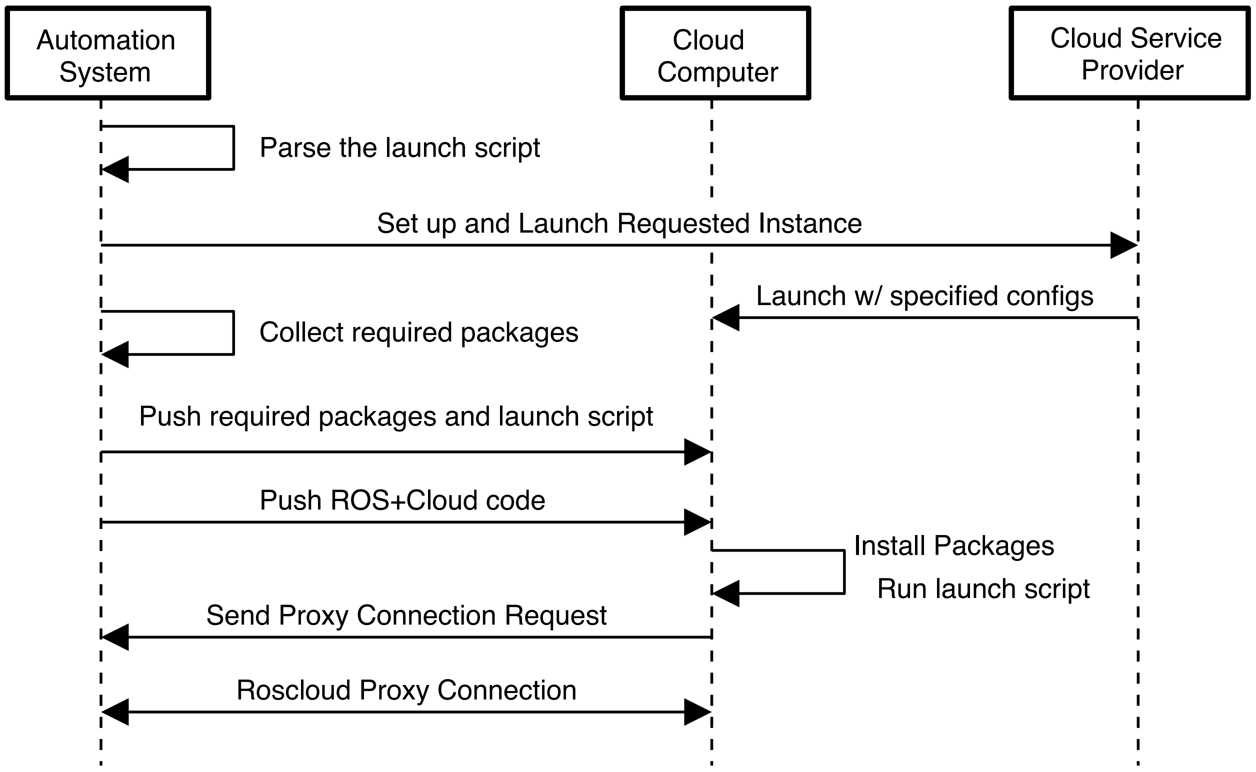

Automation System
Cloud Computer
Cloud Service Provider
Parse the launch script
Set up and Launch Requested Instance
Launch w/ specified configs
Collect required packages
Push required packages and launch script
Push ROS+Cloud code
Install Packages
Run launch script
Send Proxy Connection Request
Roscloud Proxy Connection

Figure 1.4: Sequence Diagram of FogROS Deployment Process. Users only need to input the launch file, and FogROS automates the provisioning, deployment, code execution, and network setup sequence.

VPC instance and a security group for it, (2) establish credentials for the cloud-computers that will participate in the VPC, (3) configure the cloud computers to use the VPC for cloud-to-cloud communication, and (4) set up a VPN endpoint to which the edge computer will connect. Once set up, the cloud service provider manages the VPC, while FogROS manages the VPN. As part of the setup process, FogROS sets a unique private IP address for each of the computers participating, so that the ROS nodes can establish connections between computers.

Networking: Pub/Sub Proxying

In addition to VPC networking, FogROS also supports a proxied-network option that enable communication between the edge computer and the cloud. This option is available for cases where the VPC solution may be unavailable due to service provider restrictions or costs, or when an additional level of isolation between the edge computer and the cloud is desired. There are also performance differences (see Section 2.6) when considering the

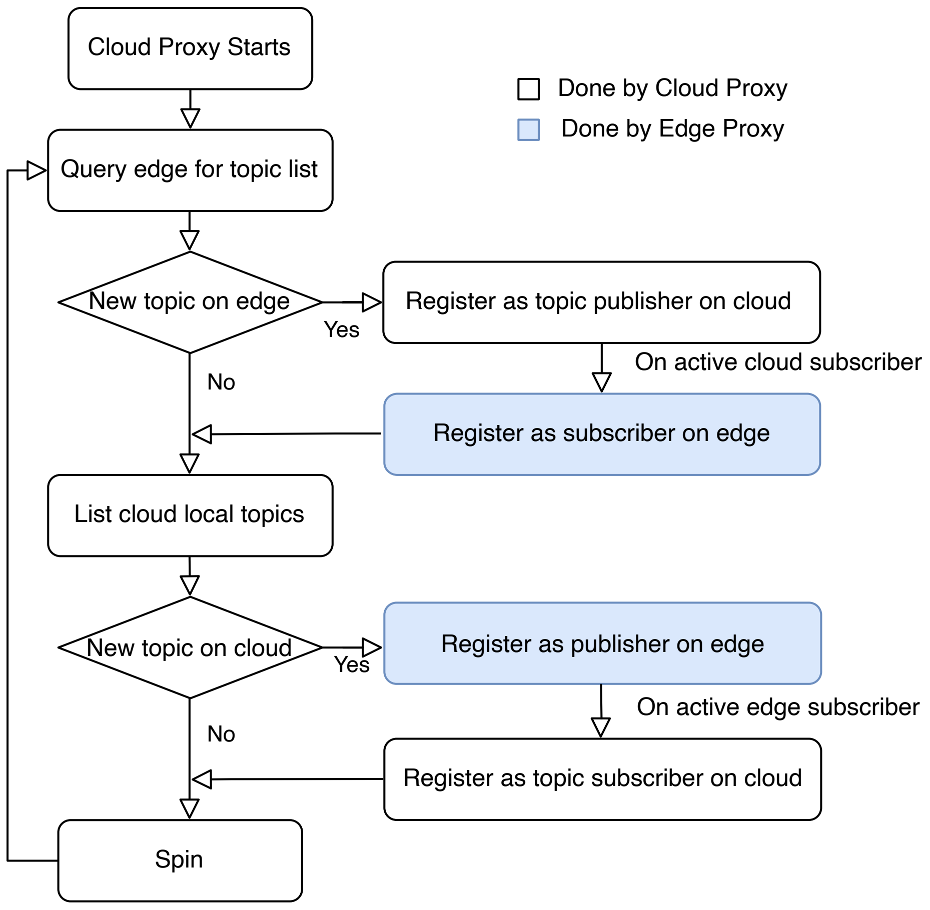

Cloud Proxy Starts
Done by Cloud Proxy
Done by Edge Proxy
Query edge for topic list
New topic on edge
Yes
Register as topic publisher on cloud
On active cloud subscriber
No
Register as subscriber on edge
List cloud local topics
New topic on cloud
Yes
Register as publisher on edge
On active edge subscriber
No
Register as topic subscriber on cloud
Spin

Figure 1.5: Automatic Pub/Sub Proxy. The proxy tunnels the traffic only if there is an active subscriber.

network options, and a user of FogROS may wish to measure performance in their application before choosing a suitable option.

In FogROS, a proxy consists of two ROS nodes, one running in the edge computer and one running on the cloud. These nodes connect directly to each other via a secure network connection, and register as publishers and subscribers to topics on the ROS Master running on each computer. When a proxy node receives a message from a subscription, it sends it to the other proxy node, which then publishes it to the subscribers registered on its ROS Master.

There are two options for FogROS to identify topics to proxy: (1) user-specified in the configuration file, or (2) automated. If topics are specified by the user in the configuration file, FogROS subscribes and publishes to the topics specified. If the user does not specify topics, FogROS communicates with the ROS Master on each end and identifies which topics have registered subscribers and publishers. When a topic has a publisher on one end, and a subscriber on the other, the ROS proxy nodes coordinate with each other to proxy the associated topic (see Fig. 1.5). While the automated process is simpler to setup for the developer, it may result in increased setup time as the proxy nodes coordinate the setup of proxied topics, or wasted bandwidth on topics that do not need proxying.

Network Monitoring

With the proxying network option, FogROS also provides interfaces to monitor network conditions via ROS topics /fogros/latency and /fogros/throughput on both the edge computer and the cloud. These interfaces do not introduce additional overhead unless an active subscriber subscribes to them. Users can also inspect and interact with ROS topics with standard ROS tools such as rostopic. In addition, FogROS provides the same fault tolerance as ROS running locally, where ROS nodes can re-join the pub/sub communication after network interruption.

Pre-Built ROS Nodes

FogROS also supports launching containerized ROS nodes with a similar interface to the FogROS launch script extension described in Section 1.5. While an increasing number of ROS developers are using pre-built docker images to host ROS nodes, this functionality is not natively supported by ROS. With FogROS, users can specify the name of a publicly-available image on DockerHub as well as the destination machine on which they want to launch it. FogROS then uses a template environment setup script to pull and run the image on the specified machine. It analyzes the machine type and configures the docker run command to match the hardware (e.g., GPU) available on the computer.

Fig. 1.6 shows an example launch script for a Dex-Net grasp planning node in a docker image. FogROS provisions and starts a cloud computer (g4dn.xlarge) with a GPU, pulls the

```
<launch>
    <!-- Dex-Net Docker Image w/ FogROS -->
    <node name="dexnet" pkg="fogros" type="
        fogros_docker.py" output="screen">
        <rosparam>
            docker_image: keplerc/dexnet:gpu
            machine_type: g4dn.xlarge
        </rosparam>
    </node>
</launch>
```

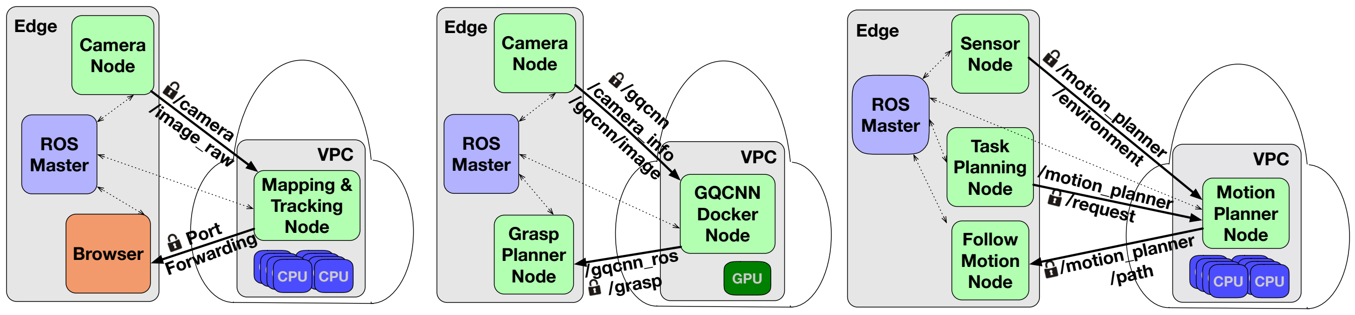
Figure 1.6: FogROS Docker Container Example

dexnet:gpu image from DockerHub [13], and attaches the docker container to the GPU, and runs it.

1.6 Evaluation

Here we present three example applications on FogROS: (A) visual SLAM, (B) Dex-Net grasp planning and (C) multi-core motion planning. The nodes, topics, and split between a single-core edge computer with 2GB RAM and the cloud are shown in Fig. 1.7. In addition to showing the network latency and performance with FogROS, we highlight the simplicity and minimal configuration of deploying these applications.

Visual SLAM Service



(a) VSLAM (b) Grasp Planning (c) Motion Planning

Figure 1.7: FogROS Docker Container Example

	Edge		Cloud —	FogROS —	Network	
Scenario	FPS	Create (s)	FPS	Create (s)	VPC (s)	Proxy (s)
fr1/xyz	16.6	2.9	**25.2**	**1.2**	0.4	0.6
fr2/xyz	14.7	2.1	**30.2**	**0.8**	0.4	0.6
fr1/desk	15.6	2.8	**23.8**	**2.0**	0.4	0.6

Table 1.1: SLAM with FogROS. We benchmark FogROS on 3 different visual SLAM scenarios, and record the frame-per-second (FPS) and the latency in creating a new map (Create) in seconds on the local edge computer (using one-core CPU) and a cloud-computer with 36-core CPU. We also record the average network time in seconds for transmitting raw video frames to the cloud computer. FogROS demonstrates up to 2.0x improvement on FPS and 2.6x improvement on new map creation time than using only edge computer.

ORB-SLAM2 [39] is a visual simultaneous localization and mapping system that uses monocular video input. In this experiment, a Camera Node publishes a 640×480 resolution video with each frame 48 KiB on average to the cloud (Fig. 1.7a). On the cloud an ORB-SLAM2 node subscribes to the video feed [57] and computes a pointcloud map along with the current estimated location within the map, which are sent back to the robot. For more details on the ORB-SLAM2 algorithm, we refer readers to the paper and open-source code available from Mur-Artal *et al.* [39].

To configure FogROS to work with ORB-SLAM2, we build ROS docker images and push them to Dockerhub [13]. We wrote a bash script to pull and run the docker image and include its path as in Fig 1.3; FogROS then runs the script when configuring the environment. After initialization, FogROS sets up and secures communication between the robot and the cloud SLAM server.

To evaluate the performance of FogROS when deploying ORB-SLAM2, we compare the cloud-deployed performance to an edge-computer-only implementation. We select a 36-core cloud-computer (AWS c4.8xlarge) for the ORB-SLAM2 node, and compare it with ORB-SLAM2 running on a one-core edge computer We report frames-per-second (FPS) and latency that creates the first map (in seconds) [41]. Table 1.1 suggests that cloud-based SLAM can achieve higher FPS, meaning that it can aggregate more data and produce higher quality maps in a real-time setting. Cloud-based SLAM also has less latency in generating a new map because the time to create a new map on the cloud is less than the time to create a new map on the edge.

Dex-Net Grasping Service

Grasp analysis computes the contact point(s) for a robot gripper that maximize grasp reliability—the likelihood of successfully lifting the object given those contact points. To plan grasps on rigid objects in industrial bins using an overhead depth camera, we use an open-source implementation of the fully-convolutional grasp-quality convolutional neural network (FC-GQ-CNN) [54, 35] from Dex-Net[34]. We wrap FC-GQ-CNN in a ROS node and deploy it to the cloud along with pretrained neural-network weights as a Docker image. We refer the reader to Satish *et al.* [54] and Mahler *et al.* [35] for details and code for the neural network and grasping environment.

This node subscribes to 3 input topics containing a scene depth image and mask for objects to be grasped, and a message of type sensor_msgs/CameraInfo containing camera intrinsics. Internally, the node feeds this to FC-GQ-CNN, which outputs a grasp pose and associated estimate of grasp quality. The node wraps these outputs, along with the gripper type and coordinates in image space, into a gqcnn_ros/GQCNNGrasp message, and publishes it.

While the node can be run both locally or in the cloud, using cloud GPU instances as opposed to a CPU for neural-network inference can greatly reduce computation time. In either case, the node is wrapped inside of a Docker container, reducing the need for resolving dependency issues between deep-learning libraries, CUDA, OS, and ROS versions. The pretrained models in the image is intended for a setup similar to that shown in Figure 1.8;

	Edge	Cloud	FogROS VPC		FogROS Proxy	
Scenario	Only (s)	Compute (s)	Network (s)	Total (s)	Network (s)	Total (s)
Compressed	7.3	0.6	0.6	**1.2**	0.8	**1.4**
Uncompressed	7.5	0.6	0.7	**1.3**	0.9	**1.5**

Table 1.2: Dex-Net Grasp Planning with FogROS. We benchmark Dex-Net on 10 trials, and record the compute time in seconds on the local edge computer with CPU only (1st column), and the total (4th&6th column, respectively) = compute (2nd column) + network (3rd&5th column, respectively) time for using a 4-core cloud computer with a single Nvidia T4 GPU. FogROS demonstrates up to 6.0x improvement with VPC and 5.7x improvement with proxy than using only edge server.

variations in camera pose, camera intrinsics, or gripper type may require retraining the underlying model for accurate predictions.

We run FogROS with the Dex-Net docker image using the launch file in Fig 1.6. We compare grasp planning times across 10 trials using both the CPU onboard the edge computer and FogROS with the Docker images on the cloud. We also show compute times when using a compressed depth image format to transfer images instead of transferring raw images to the cloud directly. For the latter case, images are compressed and decompressed using the `republish` node from the image_transport ROS package [25]. Table 1.2 shows the results for both compressed and uncompressed image transport between the nodes.

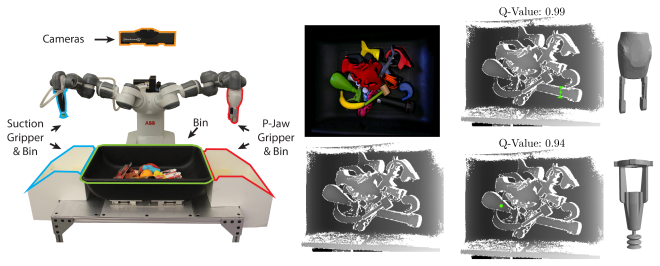


Figure 1.8: FogROS Docker Container Example

Multi-Core Motion Planning

Motion planning computes a collision-free motion for a robot to get from one configuration to another. Sampling-based motion planners randomly sample configurations and connect them together into a graph, rejecting samples and motions that are in collision. These planners can be scaled with additional computing cores.

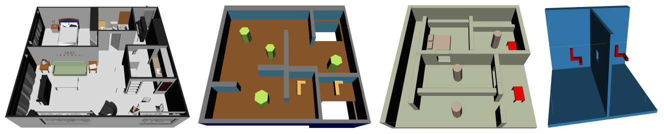


Figure 1.9: FogROS Docker Container Example

	Edge	Cloud	FogROS VPC		FogROS Proxy	
Scenario	Only (s)	Compute (s)	Network (s)	Total (s)	Network (s)	Total (s)
Apartment	157.6	4.2	0.4	**4.6**	0.7	**5.0**
Cubicles	35.8	1.4	0.3	**1.7**	0.6	**2.1**
Home	161.8	6.2	0.3	**6.5**	0.6	**6.8**
TwistyCool	167.9	5.1	0.4	**5.5**	0.6	**5.7**

Table 1.3: Multi-core Motion Planning with FogROS. We benchmark FogROS on 6 different motion planning scenarios using the same multi-core motion planner, and record the compute time in seconds on the local edge computer (1st column), and the total (4th&6th column, respectively) = compute (2nd column) + network (3rd&5th column, respectively) time for using a 96-core computer in the cloud. FogROS demonstrates up to 34.2x improvement with VPC and 31.52x improve with proxy than using only edge server.

Using FogROS, we deploy a multi-core sampling-based motion planner [22, 21] to a 96-core computer in the cloud to solve motion planning problems from the Open Motion Planning Library (OMPL) [58] (see Fig. 1.9). This planner node subscribes to topics for the collision model of the environment and motion plan requests (Fig. 1.7c). When the planner node receives a message on any of these topics, it computes a motion plan, and then publishes it to a separate topic. For more details on the multi-core motion planner, we refer the reader to the paper and the open-source code by Ichnowski *et al.* [21]. To configure FogROS to work with multi-core motion planner, we record the steps we use to setup the dependencies (e.g., FCL [44] and Nigh [20]) in a script. By providing the script, we configure FogROS similar to Fig 1.3.

We compare the planning time as the difference between publishing a motion plan request message, and receiving the plan result message, and show the results in Table 1.3. The same motion planning problem is solved in a fraction of the time on the cloud when compared to using the edge computer. However, for simpler planning problems, when the network latency (between 0.3 s and 0.6 s) is longer than the motion planning computing time, there may be little to no benefit to use a cloud deployment. Intuitively, if it takes more time to upload data onto the cloud than computing it locally, there is no point in uploading it. If the motion planner is asymptotically-optimal (finds shorter/better plans the longer it runs and with more CPU cores), then one could potentially run the motion planner for the same amount of time but get a better path using the cloud. Anand *et al.* [1] explored and shown the benefit of using the tradeoff between more cores and the resulting motion plan optimality.

1.7 Conclusion

We present FogROS, a user-friendly and adaptive extension to ROS that allows developers to rapidly deploy portions of their ROS system to computers in the cloud. FogROS sets up a secure network channel transparent to the program code, allowing applications to be split between edge and the cloud with little to no modification. In experiments, we show that the added latency associated with pushing software components to the cloud is can be small when compared to the time gained from using high-end computers with many cores and GPUs in the cloud. However, in some simple tasks, using high-end cloud servers may lead to marginal benefits and can be considered as an overkill.

In future work, we will address the interactions of multiple hardware systems with different ROS masters, and handle the decentralized communication efficiently and securely. We will also support real-time compression on the proxy connection between edge computer and cloud to help reduce latency especially on low-bandwidth connections.

Chapter 2

Co-scheduling Feature Updates and Queries for Feature Stores

2.1 Acknowledgements

I thank my colleagues who helped immensely in the course of this work, in particular:

- Woosuk Kwon (co-author of this class project), for implementing multiple co-scheduling algorithm options for feature updates and queries and contributing to a significant part of this work.
- Jimmy Xu (co-author of this class project), for running trade-off experiments for co-scheduling feature updates and queries.
- Sarah Wooders, for advising us in the co-scheduling design decisions for the RALF feature store.
- John Kubiatowicz, for providing system insights to this project.

2.2 Introduction

In real-world machine learning applications, it is a common practice to pre-process data and get features to reduce latency and improve accuracy. Features are raw and derived data and can have different representations and encodings. The process of converting the data into features is called featurization. Machine learning models need to featurize data from the data source to do training or prediction. For example, in the scenario of credit card fraud detection, the application uses credit cards' activity patterns to identify anomalies, while the user interface of a bank can provide only raw data such as credit card number, individual transaction record time, and transaction amount. For the machine learning model to detect fraud, some pre-computation needs to happen to produce the probability of each transaction occurrence, so that the downstream model can infer if the transaction is a fraud.

Featurization usually takes a significant amount of computing power and time. It is desirable for online models to respond in a timely manner to achieve its goal (e.g. stop fraudulent transactions in time). Thus, a feature store that can pre-process raw data and respond to queries from models is ideal to hide the latency associated with featurization.

Feature stores usually store and maintain a feature table that maps from a feature key (e.g. a user ID) to featurized data (e.g. the transaction history of a user). Ideally, the table should be updated upon receiving new raw data; however, this will require a massive amount of computation and may result in higher latency. Thus, a feature store needs to decide on the frequency of update and balance the tradeoffs between latency and staleness of the features.

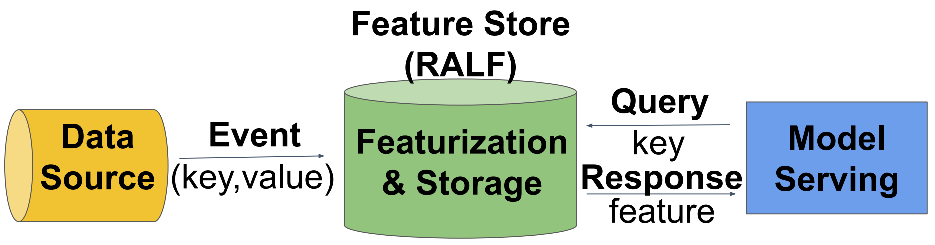


Figure 2.1: Overview of a Machine Learning pipeline with a Feature store: A data update creates an event with a key and a value. The feature store performs featurization and stores the feature. The Downstream model can issue a query with the desired key and the feature store will respond with the feature value associated with that key.

Among the many exploration of feature stores, RALF [65] is designed for streaming data, and it explicitly leverages downstream feedback to reduce costs with minimal downstream accuracy degradation. Currently, RALF only computes updates eagerly. As Figure 2.2 and Figure 2.2 shows, the latency and staleness of eager computation increase exponentially when the rate of incoming data, or update rate, is around or above 1000 per second. Our project adds the support for lazy computation to RALF to mitigate this issue while exploring the tradeoffs associated with lazy computation. With lazy evaluation, RALF will not featurize newly received raw data until the client sends a query to get that feature. Lazy evaluation will schedule a featurization if the feature for the queried key doesn't exist or return an old feature if featurization has ever happened for that key. We use the metrics defined in the following section to evaluate the effectiveness of our implementation.

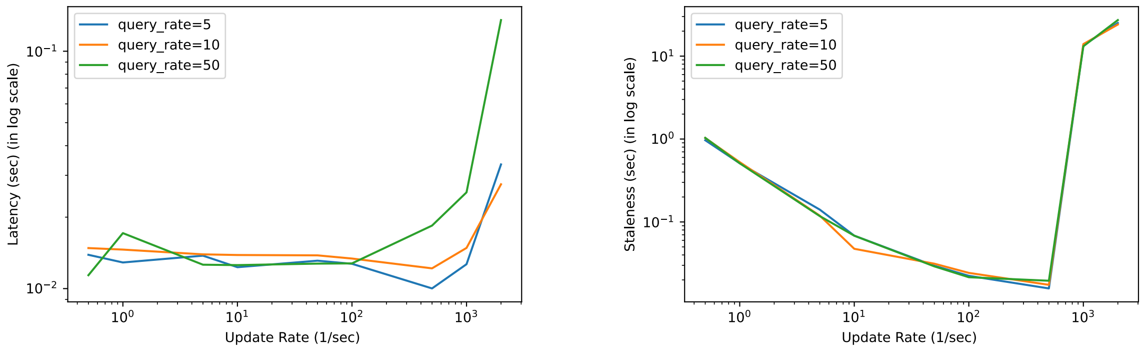


Figure 2.2: Latency and staleness of eager evaluation in log scale. Both latency and staleness of the system grows exponentially when update rate approaches 1000 times per second for the three different query rates tested.

Another issue is that RALF does not consider query service-level objectives (SLOs) for scheduling queries and features updates. As a result, they lose the opportunity to return more up-to-date features computed from the freshest data updates. In our project, we explore the query SLOs between 10ms to 50 ms to see how different scheduling policies and latency requirements can affect the accuracy of the downstream model. With the co-scheduling algorithm, features are computed when queried to meet some specified deadline while increasing the accuracy.

The rest of this paper is structured as follows. In section 2.3, we discuss the success metrics that our system considers. Then, we describe our exploration of metrics tradeoff with synthetic data in section 2.4 and our scheduler implementation in section 2.5. In section 2.6, we present the experiment results of our scheduler implementation. We then have a little discussion of the experiment results in section 2.8, compare our scheduler to related work in section 2.7, and conclude in section 2.9.

2.3 Success Metrics

When training machine learning models with a feature store, developers care the most about three metrics.

Staleness Different workloads might have different definitions of staleness but in general, the more stale the features are, the less fresh they are, and the lower accuracy the model output can have. Given the fact that the environment might change a lot when time goes by, fresher features tend to reflect the real environment more accurately. Downstream model accuracy is a clean and effective way to measure the impact of feature store staleness. In our work, we explore different ways to reduce staleness of our features and increase accuracy for two different workloads.

Latency Latency is defined as the time difference between the time a query is issued and the time a response is received. To achieve higher pipeline efficiency and also a better user experience, we pursue a lower latency when possible. In this project, we either measure the latency directly or give an explicit SLO constraint to the feature store when issuing a query.

Computation cost Featurization in feature stores can be very expensive. For example, in our embedding workload, a featurization can take a few milliseconds and there can be a few hundred data updates per second. In real world, this can be translated to AWS lambda [3] rental costs. The more computation that happens, the more developers need to pay for a serverless compute service. We aim to reduce the number of times calling the featurization methods.

In the following sections, we explore the trade-offs between these three metrics.

2.4 Exploring Trade Offs in Lazy Evaluation

We implement lazy evaluation and a load shedding policy. We then create a synthetic workload to understand the behavior of the current system as well as exploring various trade offs between lazy evaluation and eager evaluation.

Lazy Evaluation Implementation

In the current implementation of RALF, it always does featurization upon receiving new data (i.e. eager evaluation). We implement lazy evaluation with which the system will not featurize the updated raw data unless the client queries for it. Upon receiving a query, if the current table has the feature, the system will return the stale feature immediately and schedule a featurization. If the current table does not have the corresponding feature but the key exists in an upstream table, it will wait for the featurization to complete. Otherwise, a key error will be returned indicating the key does not exist in the system.

As illustrated in Figure 2.3, eager evaluation will perform featurization for updates of (key_1,value_1), (key_1,value_2), and (key_1,value_3), respectively. Lazy evaluation will not do anything until the first query with key_1 comes in. Lazy evaluation doesn't have a feature value for key_1, so it checks its upstream table and see that an update with (key_1, value_3) is the latest update with key_1. Lazy evaluation performs a featurization and returns a feature associated with value_3. Eager evaluation returns the feature associated with value_3 that it has computed. When update (key_1, value_4) arrives, eager evaluation will perform another featurization while lazy evaluation does nothing. When query with key_1 comes in again, eager evaluation will return the feature associated with value_4 that it has computed. Lazy evaluation sees that it has a feature for key_1 that was computed from value_3, so it will return value_3 to the query first, and schedule a featurization so that it can respond with the feature associated with value_4 when the next query comes in.

Load Shedding

A disadvantage of laziness is when the query rate is high, it may schedule unnecessary updates. For example, if a client monitors a value and sends heartbeat data to a feature store but the value does not change frequently, the feature store will receive multiple duplicate feature updates with the same key and value. To avoid redundant computations, our system provides an option to group N queries and check if there is duplicate update requests for the same key. The system will perform computation only once for duplicate update requests. The result of this policy is illustrated in the subsection below.

Synthetic Workload

As shown in Figure 2.4, the server simulates the featurization pipeline. The client continuously generates raw data and sends HTTP requests to the server. The server has a

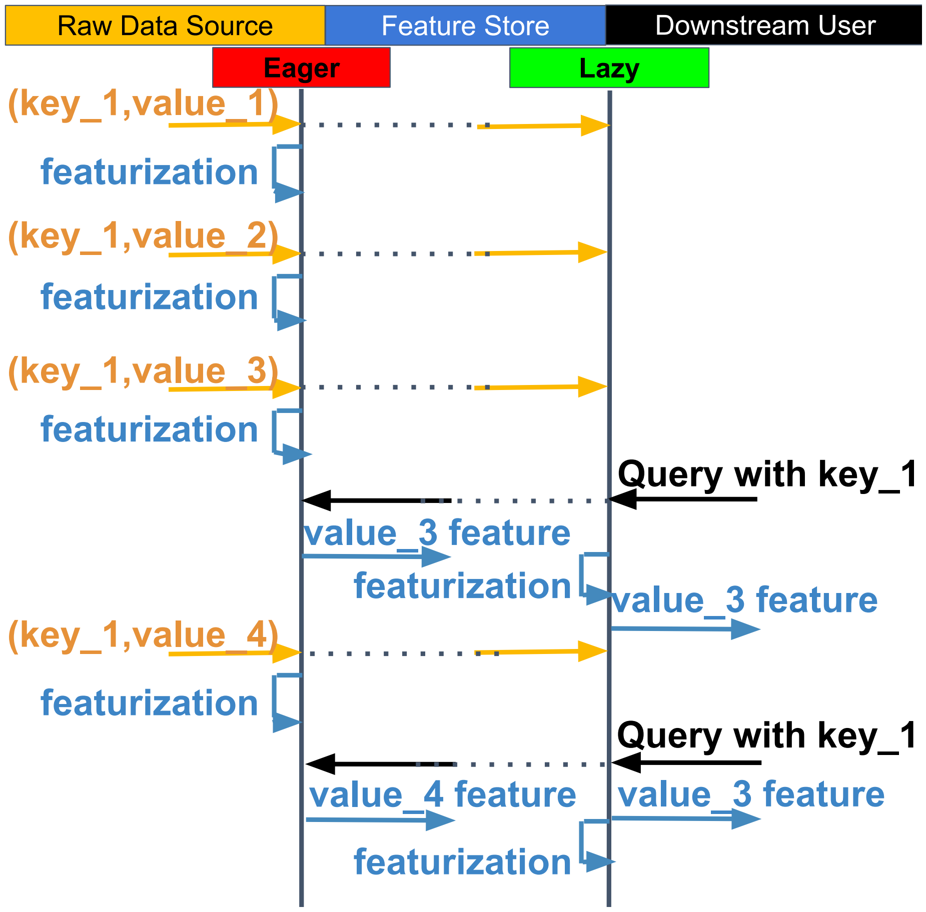


Figure 2.3: Lazy evaluation workflow: the lazy computation only does featurization when receiving a query.

source operator that receives streaming data through a Redis [50] data structure store. The server then performs featurization with a featurization operator. The client also simulates a downstream application that gets desired features by querying the table in the featurization operator. To avoid the impact of network connections, we assume that the clients, the inference server, and the feature store are running on the same local machine with varying parameters.

The implementation details are as follows. The client has `num_keys` of keys. It will count from 0 to `update_up_to` and assign the number (i.e. raw data) to key `num % num_keys`. Each data will be generated every `1/update_rate` seconds and sent to the source operator on the server. The featurization operator performs an identity operation that is set to take `processing_time` seconds, simulating an expensive featurization operation. The client sends a query every `1/query_rate` seconds.

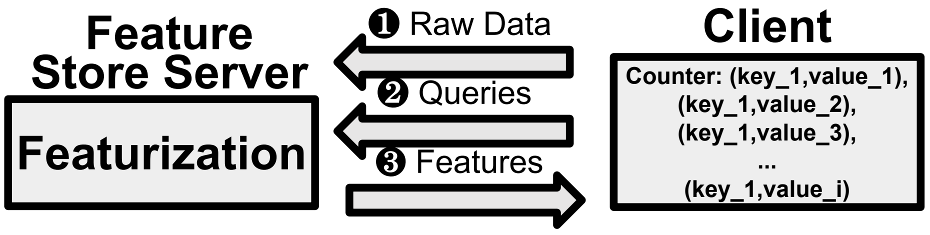

Feature Store Server
Featurization
❶ Raw Data
❷ Queries
❸ Features
Client
Counter: (key_1,value_1), (key_1,value_2), (key_1,value_3), ... (key_1,value_i)

Figure 2.4: Synthetic Workflow with a server and a client

Experiments

We conduct a number of experiments on the synthetic workload to explore: All experiments are done with the following parameters:

- There only exists one key.
- The featurization time is set to be 0.01 seconds.
- Unless specified, load shedding policies are not enabled.
- The query requests and update requests are in the same queue and are processed using FIFO policy.
- There are 4 worker threads to process queries and updates.
- The client continuously queries the server for 5 seconds, three times (15 seconds in total). The latency and staleness is the average of the last two runs, which excludes potential initialization time.

Trade-Offs We expect a higher staleness for lazy evaluation because when there is an existing value for a certain key being queried, the value will be returned immediately. Since featurization only happens when there is a query to the key, if the last time this key got queried was a long time ago, the value returned would not be very fresh (i.e., the staleness of the data is very high).

We expect a comparable latency for lazy evaluation if the update rate and query rate are within a normal range, because for a key that shows up more than once, both lazy and eager evaluation will return an old value that they computed from the last query or data update, respectively. However, if updates and queries come in from the same data stream and each query result needs to wait for the corresponding update (e.g., real-time click-recommendation on commercial websites), lazy evaluation may have a lower latency because it doesn't need to wait for the featurization to complete.

Since featurization is the most expensive process, we expect lazy evaluation to have a higher computation cost when the query rate is high because it does featurization whenever a query comes in. For the same reason, we expect eager evaluation to have a higher computation cost when the update rate is high because it does featurization for every data update.

To confirm our hypothesis about metrics changes, we vary the update rate and query rate for both lazy evaluation and eager evaluation, and measure the latency, staleness, and computation cost.

Load shedding In both eager and lazy evaluation, our load shedding policy can group N update requests for the same key so that only one update is performed, as opposed to N. This decreases the number of updates and increases the limit of concurrent updates and queries.

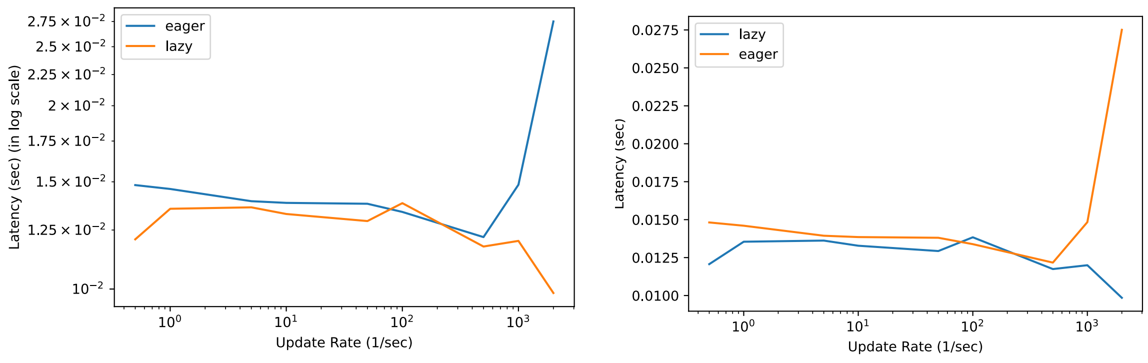


Figure 2.5: Latency of eager and lazy evaluation with fixed query rate of 10. The figure on the left is in log scale. The figure on the right is the zoomed-in version in linear scale. Latency of the eager system grows exponentially when the update rate approaches 1000 times per second. Latency of the lazy system stays low when the update rate grows.

Observations

Varying the update rate Figure 2.5 and Figure 2.6 shows that when the query rate is fixed, the latency and staleness for the feature store with lazy updates and eager updates are around the same when the update rate is below 1000 times per second. For a higher update rate, the latency of the feature store with eager computation grows exponentially.

Varying the query rate Figure 2.7 and Figure 2.8 shows that when update rate goes high, the eager evaluation will encounter a congestion problem because the limited computing

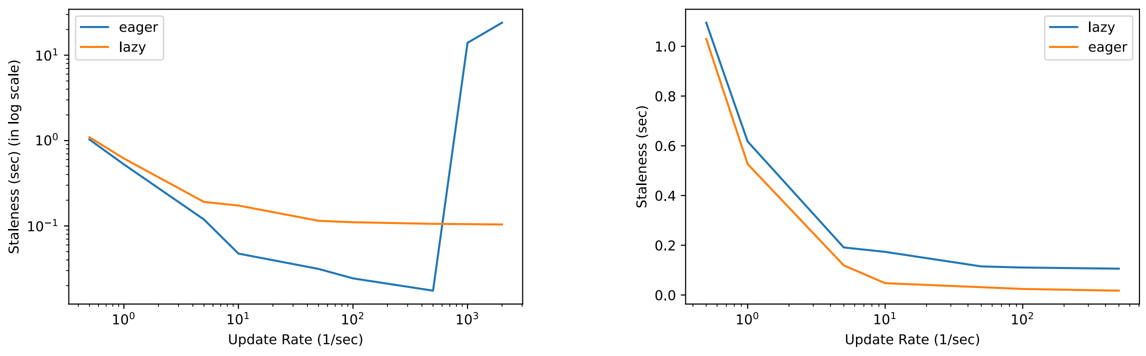


Figure 2.6: Staleness of eager and lazy evaluation with fixed query rate of 10. The figure on the left is in log scale. The figure on the right is the zoomed-in version in linear scale. Staleness of the eager system grows exponentially when the update rate approaches 1000 times per second. Staleness of the lazy system stays low when the update rate grows.

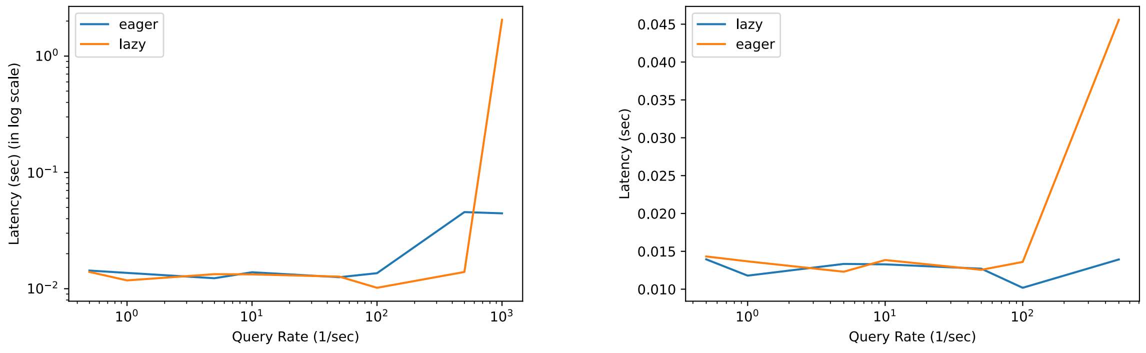


Figure 2.7: Latency of eager and lazy evaluation with fixed update rate of 10. The figure on the left is in log scale. The figure on the right is the zoomed-in version in linear scale. Latency of the lazy system grows exponentially when the query rate approaches 1000 times per second. Latency of the eager system stays low when the query rate grows.

resources cannot process all the updates. This leads to a rise in the latency and staleness of records. Lazy evaluation mitigates this problem by postponing processing new updates until receiving queries. For the synthetic data, only the latest value associated with the same key matters so as long as the query rate stays the same, the lazy evaluation only performs certain amount of computation. In addition, since featurization only happens when receiving a query in lazy evaluation, if the query rate is too low (¡ 10 queries / second), the staleness can be high, as shown in Figure 2.8. In this case, too many updates are discarded because

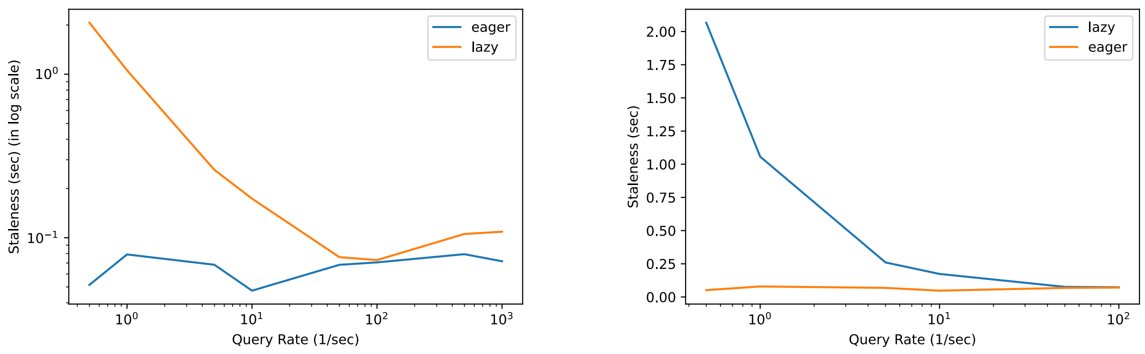


Figure 2.8: Staleness of eager and lazy evaluation with fixed update rate of 10. The figure on the left is in log scale. The figure on the right is the zoomed-in version in linear scale. Staleness of the lazy system is higher than the eager system when the query rate is smaller than 10 queries per second. Staleness of the eager system stays low when the update rate grows.

of the infrequent queries. As query rate grows, the staleness of data for a feature store with lazy evaluation and eager evaluation converges.

Computation costs With similar reasons as above, more computation leads to more cost. When update rate goes high, the cost for a feature store with eager evaluation goes high. When query rate goes high, the cost for a feature store with lazy evaluation goes high. The plateaus in Figure 2.9 indicates the system cannot process more updates concurrently.

Load shedding As expected, by grouping duplicate update requests, our load shedding policy decreases the number of redundant featurization and prevent the computation cost from growing too fast. The results for lazy computation with and without load shedding policy are shown in Figure 2.10. With load shedding policy, the computation cost grows slower than the system without load shedding policy as the query rate increases.

The current implementation of RALF treats update requests in the same way as query requests. It only has one queue to process both events, which may result in unnecessary waiting time for time-sensitive queries. Prioritizing queries is discussed in later sections.

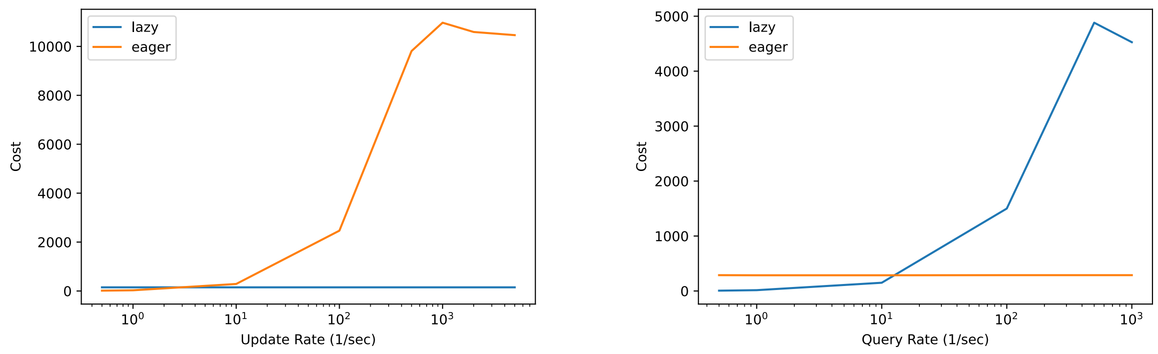


Figure 2.9: Computation costs of eager and lazy evaluation with fixed query rate of 10 (left) and with fixed update rate of 10 (right). Cost of the eager system grows exponentially when the update rate approaches 100 times per second. Cost of the lazy system stays low when the update rate grows. Cost of the lazy system grows exponentially when the query rate approaches 100 times per second. Cost of the eager system stays low when the query rate grows.

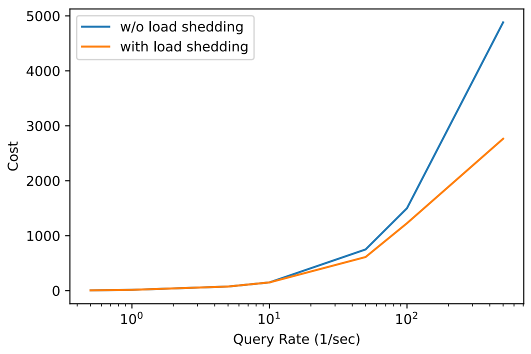


Figure 2.10: Cost between lazy evaluation with and without load shedding with fixed update rate of 10. Cost of the lazy system without load shedding policy grows faster than with load shedding policy.

2.5 SLO-Aware Featurization Scheduler

In this section, we design and implement an SLO-aware featurization scheduler based on the explored trade offs in Section 2.4. Motivated by the fact that the ultimate goal of real-time ML serving systems is to provide high-quality predictions while meeting SLOs, we argue that the goal of feature store should be to provide high-quality features within the query

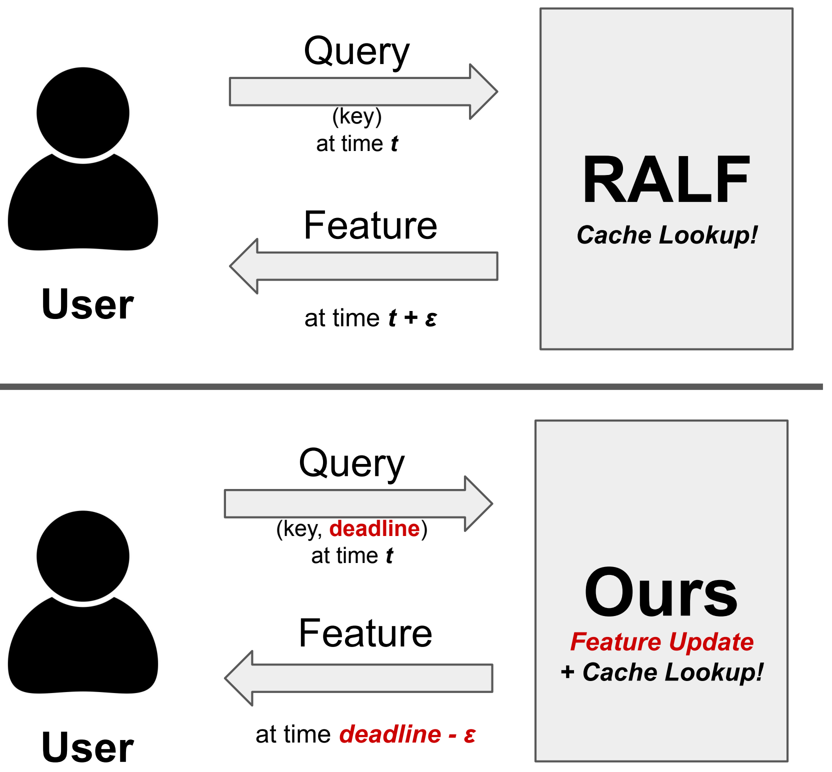


Figure 2.11: The high-level comparison between RALF and our system. While RALF is designed to immediately respond to queries by a simple cache lookup, our system provides a way for users to specify their query deadlines and considers the deadline information in scheduling.

deadlines. We show that by considering SLOs our scheduler can provide the most up-to-date features to queries and thus significantly enhance the downstream model accuracy.

Figure 2.11 illustrates the key difference between RALF [65], the existing feature store system, and our system. In RALF, the task of responding to queries and the task of updating features are done *asynchronously*. Specifically, RALF treats a feature query as a lookup to its cache store which is continuously updated by the incoming stream of events. As such, because individual queries are responded without any waiting time by RALF, their latencies are extremely low (i.e., < 5 ms) and stable. Nevertheless, RALF loses the opportunity to return more up-to-date features by co-scheduling query responses and feature updates. For example, when a query and an event to the same feature arrive simultaneously, RALF cannot reflect the latest event in the query response. For many real-time applications sensitive to feature staleness, this can lead to significant accuracy loss. Our scheduler addresses this limitation by accepting the deadline of each query and using it to intelligently schedule the

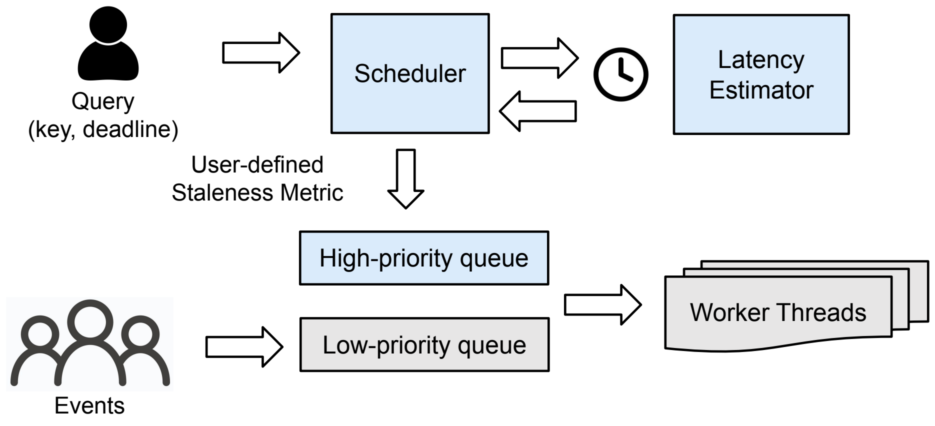


Figure 2.12: Overview of our system. The blue boxes represent the system components we added on top of RALF. Our system accepts deadline-specified queries, and co-schedules query responses and feature updates, using its latency estimator and a staleness metric defined by the user.

featurization pipelines.

System Overview

The fundamental idea of our scheduler is to prioritize the updates of the features being queried over those not being queried. That is, our system processes the latest events of the features upon requests and returns the up-to-date features to the queries within their deadlines. The intuition behind this is that the system can process the deferred updates (i.e., updates for unqueried features) when its resources are underutilized, as the distribution of the query arrival time is naturally skewed.

However, when the query rate is high and the system has a fixed processing capacity, the scheduler cannot process the updates of all the queried features while meeting the SLOs. Thus, among the features being queried, the scheduler should select which features to update before the deadlines and which features to not. To this end, our scheduler takes a *staleness-based policy* that gives higher priority to the processing of staler features over the updates for fresher features. That is, when the query rate is beyond the system capacity, our scheduler selects the queries to the stalest features and returns the features after updating them. For the queries that are not selected by the scheduler, features cached in the system are simply returned without updates, as in the current version of RALF.

Figure 2.12 illustrates our overall system architecture. Our system augments the RALF's query API so that users can specify the deadlines of their queries. Our scheduler then makes use of the SLO information and the estimated latency of feature updates to calculate the maximum number of queries whose feature updates can be processed before the deadlines.

When the number of pending queries exceeds this number, queries to the stalest features are selected based on a *user-defined staleness metric*. Updated features are served for the selected queries, while cached (possibly stale) features are return for those not selected. When the query rate becomes low and the system resources are underutilized, the deferred updates for queried/unqueried features are processed.

Scheduling Policy

Formally, the scheduling granularity of our system is an individual event, a collection of the tasks to update a certain feature with new data. While the ways to update features may vary, we only focus on such a case that an ML algorithm is involved in the featurization pipeline, and thus each event takes non-negligible time (e.g., > 1 ms) to be processed. For example, in a recommender system, user features are often computed by a computationally heavy ML model and used for making predictions. And the user features are continuously recomputed as more user data (e.g., the user's page views) are collected. In this example, updating a user feature by running the ML model with new data is regarded as a (processing of) event. As such events stream into the system, the system scheduler should decide which events to process first and which ones to process later. The decision of the scheduler can largely affect the downstream model accuracy, as providing stale features to downstream applications may lead to bad predictions (e.g., recommending items irrelevant to the user's latest page views).

Two-level Priority. The core of our scheduling policy is assigning two-level priorities to events. Specifically, the higher priority is given to the events whose target features are being queried and the lower priority is given to those whose target features are not being queried. The priority is strict; events with the lower priority are never processed as long as there exists any higher-priority event. Also, the scheduling algorithm is non-preemptive; because each event usually takes a few milliseconds, the system simply awaits when a low-priority event blocks a high-priority one.

Staleness-based Prioritization. When the query rate is high, the scheduler cannot process the events of all queried features while meeting the SLOs. The selection of events to process before deadlines is based on the degree of staleness of the queried features. Specifically, the scheduler prioritizes processing events that recompute stale features over the those for fresh features. Our intuition is that, if the computation costs are the same, updating a stale feature is more likely to improve the overall downstream accuracy than updating a fresh feature. The system relies on users to define their staleness metrics. This allows users to leverage their domain knowledge to further enhance scheduling.

Note that the priority of the unselected events is de-escalated to the low priority. This ensures an important invariant that at every moment the number of the high-priority events in the system do not exceed the maximum number of events that the system can process within the deadlines. We find this invariant extremely helpful in bounding the query latency and efficiently managing the queue of the high-priority events.

Fairness and Starvation. Arguably, our scheduling policy maintains *fairness in terms of feature staleness* in that it essentially aims to equalize the staleness of the stored features. Moreover, we can guarantee that our scheduling policy has no starvation problem. One might point out that the events for unqueried features will never get processed if the query rate is continuously high and the features are never queried. However, this is actually desirable, as the system is saving the computation cost for updating the features that will not be queried afterwards. When such a feature is queried, the scheduler will likely find it much staler than others and prioritize the processing of its events.

Scheduling Mechanism

As in Figure 2.12, we use multi-level queues for the two-level priority scheduling. The both queues use first-in-first-out (FIFO) processing policy. The detailed scheduling mechanism is as follows.

When a query arrives, the system first computes the staleness of the queried feature using the user-defined staleness metric function. The function takes the cached feature and its latest event as the inputs, and express the feature staleness in a non-negative real value. The 0 staleness indicates that the feature is in its freshest state while larger values mean that the feature is staler. Here, queries to the features that are not yet created by pending events get the maximum level of staleness.

Usually, the system has unprocessed events in the low-priority queue for the queried feature. The scheduler predicts whether these events can be processed before the query deadline. The prediction is made by considering the number of events in the high-priority queue that should be processed for preceding queries and the processing time of an event estimated by the *latency estimator*. The latency estimator periodically measures the processing time of the individual events and provides the exponential moving average of the processing time to the scheduler.

The events for the queried feature are simply put into the high-priority queue if they can be processed before the query deadline. If not, the scheduler tries to reduce the processing time of the query by removing some of the unprocessed events in the high-priority queue. Specifically, if the high-priority queue contains an event to update a feature less stale than the queried one, the scheduler removes that event from the queue. This process is repeated until the estimated processing time of the query becomes before the query deadline or no event can be further removed from the queue by the feature staleness comparison.

Lastly, the updated feature is stored in the cache and served to the querying user. Importantly, we design our system to *asynchronously* perform the task of recomputing features and the task of responding to queries (just like RALF). Regardless of the scheduling state, the system retrieves queried features from its cache at the moment right before the query deadlines. For example, if a query latency budget is 100 ms, the system will simply return the feature stored at the moment when 99 ms has been elapsed from the query arrival. Although such a trick can lead to higher average latency because of the unnecessary delay in query responses, we find it necessary for simplifying the system design and strictly bounding the latency even when the latency estimation is inaccurate.

Implementation

Our system is implemented on top of RALF, which is in turn built on top of Ray Serve [38, 49], a scalable model serving library. Our scheduler is implemented in ˜300 lines of Python code. We also provide default staleness metric functions that use either the elapsed time from when the feature is lastly computed or simply a random value.

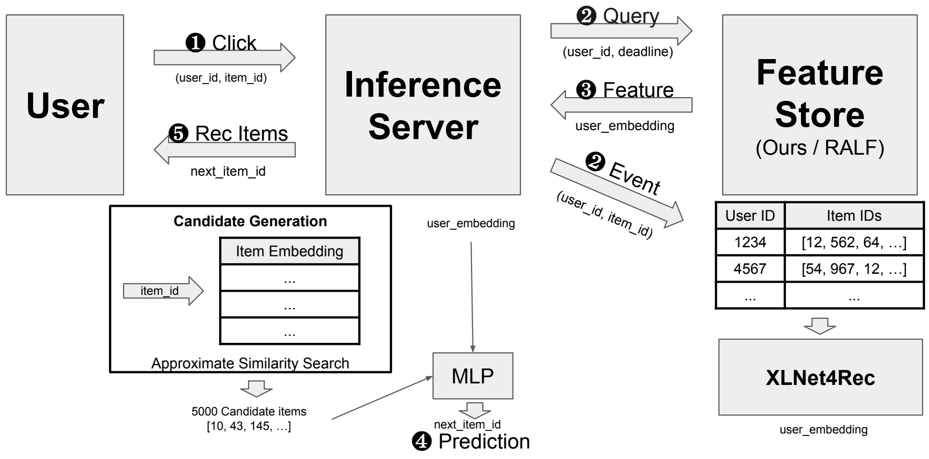


Figure 2.13: End-to-end recommendation serving pipeline used for our experiments.

2.6 Evaluation

We showcase the effectiveness of our system on an online recommendation workload. Specifically, we evaluate the system on a next item prediction task, a core recommendation problem whose goal is to predict the next item that the user is most likely to interact with given the sequence of the user's previous items. Obviously, in order to achieve high accuracy on the next item prediction task, it is important to consider the users' latest interaction with items to infer their current interest. In this section, we show that our system enables this by co-scheduling query responses and feature updates whereas RALF produces stale features and results in significantly low accuracy.

End-to-End Workflow

Figure 2.13 illustrates the end-to-end recommendation system pipeline developed for our experiments. Imagine that you are shopping on an online store such as Amazon. When you click into an item page, the site will also show other items you may be interested in. For example, if you are browsing smartphones, the site will probably display ones from different vendors or some accessories for the phones. In personalized recommendation systems, the recommended items are not only relevant to the one you are currently viewing but also have connections with your recent browsing history, which better represents your current interest. If the recommendation is successful, you will click through one of the items. The end goal of the system is to increase the likelihood that you click one of the recommended items at every browsing step.

The process of making recommendations is collaboration of two separate systems, an inference server and a feature store (i.e., either our system or RALF). The inference server is responsible for accepting user inputs, making final predictions, and sending them back to the users. The predictions are made using the features provided by the feature store, which are in this case the *user embeddings* produced by an ML model called XLNet4Rec [56].

The system workflow can be described as follows. 182 Whenever a user clicks an item, 183 the inference server queries the user embedding to the feature store. The key of the query is the user ID, and the deadline of the query is given by the inference server. Here, at the same time the user embedding is queried, 183 the inference server also creates an event for updating the user embedding with the new item ID. That is, a user click works as both a query and an event, since a click is a request to a new recommendation but is also used to recompute the user embedding. 184 After the feature store returns the queried feature, 185 the inference server makes predictions via similarity search between the user embedding and the item embeddings. 186 Finally, the inference server serves the list of recommended items to the user.

Workload Generation

Base Benchmark. Our workload is generated from the REES46 E-commerce dataset [1], which contains real customer behavior data from a large online store. The dataset includes the log of user-item interactions, such as page view, add-to-cart, and purchase, with the timestamps and item metadata. For simplicity, we do not distinguish the types of interactions and regard them equally as a click. Additionally, we ignore the item metadata and only use the tuples of (user_id, item_id, timestamp) in training and evaluating our recommendation model. Lastly, out of the seven-month data in the dataset, we only use the events from the month of October 2019 for our experiments, as in [18].

Recommender Model. Following the recent findings that Transformer models [59, 8, 56] achieve state-of-the-art performance in the next item prediction tasks, we use XLNet4Rec [56] as our recommender model. Similar to BERT4Rec [59], XLNet4Rec consists of an item embedding table and 2 Transformer blocks, each of which has 4 attention heads and hidden dimension of 448. The model takes as the input a sequence of the item IDs that a user has interacted with, and encodes it to a feature that densely represents the user interests, which we call a *user embedding.* Then the items whose embeddings are most similar to the user embedding are recommended to the user. The model is incrementally trained on the data from October 1st to October 30th, and is tested on the October 31st data.

In the test data, we find that the peak number of clicks per second is only 25, which is insufficient to show the impact of our scheduler. To evaluate our system on a more intensive workload, we synthesize a click pattern by adjusting the initial timestamps when users starts their clicks. This way, we get a click pattern where the time interval between the user's subsequent clicks is preserved while more clicks arrive simultaneously to the system. From the synthesized workload, we extract the peak 10 minutes data, which has at most 250 clicks per second.

[1]https://rees46.com/en/datasets/

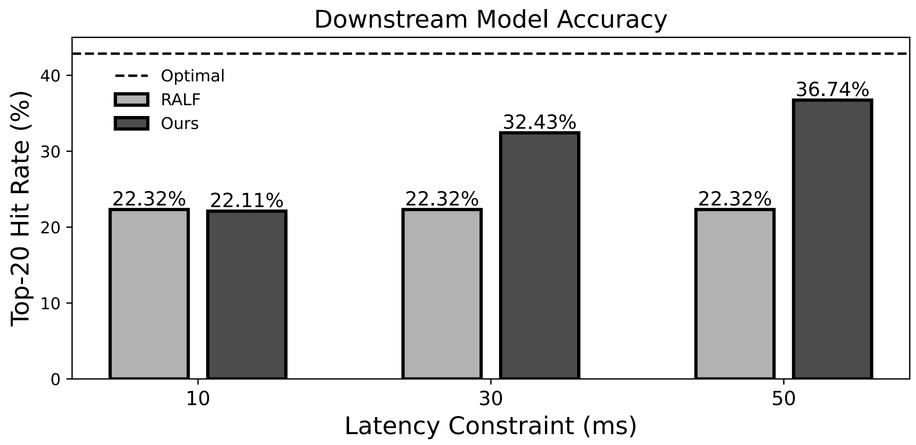


Figure 2.14: Recommendation accuracy of RALF and our system with varying latency constraints.

Experimental Setup

All of our experiments in this section are conducted on an AWS c5d.9xlarge instance [2], which has 48 virtual CPUs. Note that, following the common practice in recommendation systems [18], all the computations including the inference of XLNet4Rec are processed on CPUs, not GPUs.

The code of XLNet4Rec is brought from the NVIDIA Transformers4Rec repository [42]. We train the model with a standard training recipe. At inference time, the model is executed on PyTorch [45] which is powered by the Intel MKL-DNN library [26].

The inference server is implemented using Ray Serve. For the approximate similarity search, we use the CPU version of the Faiss library [14]. In querying to the feature store, the inference server sends HTTP requests. For sending new data to the feature store, the inference server uses Redis Stream [50]. To avoid the impact of network connections, we assume that the clients, the inference server, and the feature store are running on the same machine.

Downstream Accuracy

Figure 2.14 shows the downstream model accuracies of RALF and our system. We use top-20 hit rate as our accuracy metric, which is the probability that the true next item belongs to the top-20 recommendations. In the optimal case where the recommendation is always made on the most up-to-date features, the accuracy is 42%. This is the upper bound of the accuracy that we can obtain from our XLNet4Rec.

[2]https://aws.amazon.com/ec2/instance-types/c5/

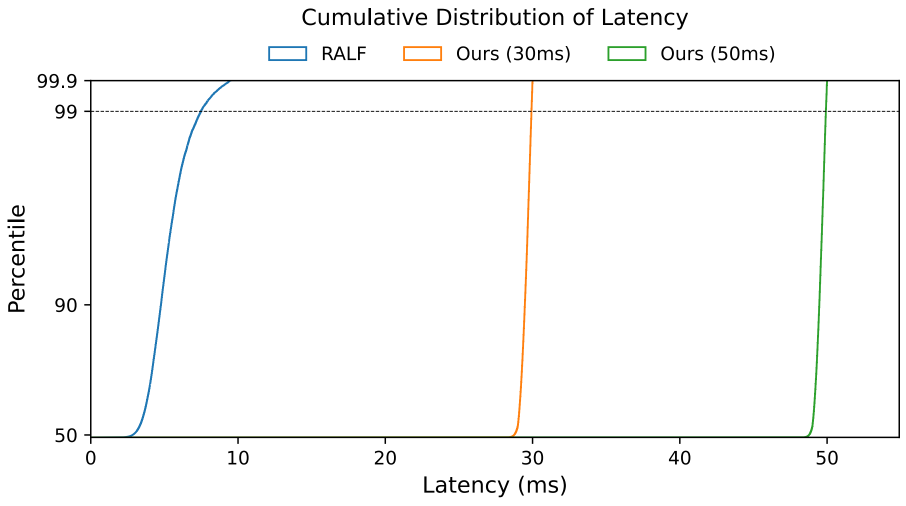


Figure 2.15: Cumulative distribution of query latencies of RALF and our system, with two different latency constraints (i.e., 30 ms and 50 ms). Because the latency of RALF does not change by the latency constraint, it is drawn once. The y axis is scaled to highlight the SLO violation rates.

Our system shows up to **14%** improvement in accuracy compared to RALF when the latency constraint is 50 ms. Our system increases accuracy by 65%. The accuracy for our system has reached 87.5% of the optimal accuracy (42%), which can be attained when unlimited computation time is available. The huge accuracy gain mostly comes from our co-scheduling policy that enables updating queried features before returning them. Particularly in our next item prediction task, this difference is critical in accuracy; a query and an event (from the same click) always arrive simultaneously, and the users' latest click information is a key ingredient in predicting their current interests. Because RALF cannot immediately reflect this information in the queried features, it suffers from significantly low accuracy. On the other hand, when a sufficient latency budget (e.g., $\geq$ 30 ms) is given, our system recovers most of the accuracy loss by intelligently prioritizing the updates for queried features.

Query Latency

Figure 2.15 shows the cumulative distribution of query latencies of RALF and our system when 30/50 ms latency constraints are given. Note that the latency (and actually the behavior) of RALF does not change according to the latency constraints, as RALF does not use the SLO information in scheduling.

As discussed in Section 2.5, RALF retains low and stable latency since it processes every feature query as a cache lookup. In contrast, our system shows much higher average latency as the system intentionally postpones responding to queries until their deadlines. However, our system still meets the SLO for more than **99.7%** of the queries, as shown in Figure 2.15. This is because the system falls back on its cached feature store when the processing time of queries accidentally exceeds the deadlines. We believe that, under such low SLO violation rates, the accuracy improvement by our scheduler far outweighs the potential advantages of the unnecessarily low latency of RALF in most cases.

2.7 Related Works

Recently, many ML serving systems have been proposed both in academia [10, 11, 17, 51, 68, 9, 55] and industry [62, 49, 43]. The main goal of these systems is to serve as many ML queries as possible while meeting the SLOs. Their core techniques are batching, load balancing, and auto scaling [16], which are not covered in detail in our paper.

A limitation of the existing ML serving systems is that they only handle *stateless* inference jobs, such as object detection and text classification, where no intermediate data (i.e., features) persist after the completion of the job. In essence, the systems regard an inference job as running a pure function that receives model inputs and returns a prediction. However, this is not the case in many ML serving pipelines; ML models often use pre-computed features for their inference, which should be cached and continuously updated over time by the system.

Feature store has emerged as the system layer responsible for storing the pre-computed features. Feast [15] is an open-source feature store that manages fast retrieval of features with consistency. However, Feast itself does not manage the featurization pipeline, and thus cannot exploit the opportunity to improve its accuracy/computation cost trade off by intelligently scheduling the feature updates. RALF [65] addresses this limitation by leveraging downstream feedback. It supports a fine-grained scheduling mechanisms over keys and periodically controls the feature update frequency to reduce the computation cost. Our system extends RALF to more intelligently schedule feature updates by exposing the SLO information of queries to the scheduler.

2.8 Discussion

Feature store is a very new concept in the system design area and researchers are currently conducting a series of exploration of its functionality, effectiveness, and tradeoffs. The three metrics that we propose are tentatively used by RALF and we regard them as the most important attributes of feature store research. In our exploration of feature store tradoffs with synthetic data, we've observed that lazy updates can cause high latency when query rate goes high and causes a congestion. On the other hand, if we can resolve the query congestion issue with some load shedding policy, lazy computation has some advantage in decreasing latency by avoiding waiting for featurization. Laziness also saves some compute power by skipping some updates when not all the updates are important for the downstream model. Additional update rules can be added to RALF by extending its load shedding method library, or adding specific data types that stores updates with different desired attributes. The synthetic data is currently generated from a counter workload, which doesn't use much CPU of the machine. In future works, we will replace it with a random dot product calculation to represent real world workload more precisely.

In Section 2.5, we rely on many assumptions to simplify the problem. First, we assume that our system runs on a single node, which has a fixed amount of CPUs. Thus, one of our future works would be to extend our scheduler to work on multiple nodes with heterogeneous devices (e.g., GPUs and TPUs). Second, our system does not consider batch processing of events for different features. As batching can greatly improve the system utilization in ML workloads, we can expect that introducing batching algorithms to our scheduler will further enhance the downstream accuracy. Lastly, our scheduler assumes that the system has no malicious users. Since the queries to non-existent features (i.e., those not yet created by the pending events in the low-prirority queue) get the highest priority, a malicious user can block the processing of other users' queries by continuously sending queries to new features. In our future work, we will explore a method (e.g., priority boosting) to protect the system from such an attack.

2.9 Conclusion

In this paper, we first explore the trade offs in designing a feature store. The main factors we considere are query latency, feature staleness, and computation cost. By comparing the eager and lazy evaluation mechanisms, we find that the optimal scheduling algorithm should adaptively use either of the mechanism, depending on the workload intensity.

Based on the explored trade offs, we design and implement an SLO-aware featurization scheduler on top of RALF. It allows users to specify their query deadlines, and co-schedules feature updates and query responses to reduce the staleness of the queried features. In our experiments, we show that our system improves the downstream accuracy by up to 14% while achieving the SLO violation rates less than 0.3%.

Bibliography

[1] Raghav Anand et al. "Serverless Multi-Query Motion Planning for Fog Robotics". In: *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*. IEEE. 2021.

[2] *AWS IoT Greengrass*. `https://aws.amazon.com/greengrass/`. Accessed: 2021-02-15.

[3] *AWS Lambda*. `https://aws.amazon.com/lambda`.

[4] Spencer B Backus, Lael U Odhner, and Aaron M Dollar. "Design of hands for aerial manipulation: Actuator number and routing for grasping and perching". In: *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*. IEEE. 2014, pp. 34–40.

[5] Kostas Bekris et al. "Cloud automation: Precomputing roadmaps for flexible manipulation". In: *IEEE Robotics & Automation Magazine* 22.2 (2015), pp. 41–50.

[6] Kehoe Ben, Berenson Dmitry, and Goldberg Ken. "Toward cloud-based grasping with uncertainty in shape: Estimating lower bounds on achieving force closure with zero-slip push grasps". In: *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*. 2012, pp. 576–583.

[7] Kehoe Ben et al. "Cloud-based robot grasping with the google object recognition engine". In: *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*. 2013, pp. 4263–4270.

[8] Qiwei Chen et al. "Behavior sequence transformer for e-commerce recommendation in alibaba". In: *Proceedings of the 1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data*. 2019, pp. 1–4.

[9] Yujeong Choi, Yunseong Kim, and Minsoo Rhu. "Lazy Batching: An SLA-aware Batching System for Cloud Machine Learning Inference". In: *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2021, pp. 493–506.

[10] Daniel Crankshaw et al. "Clipper: A low-latency online prediction serving system". In: *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 2017, pp. 613–627.

[11] Daniel Crankshaw et al. "InferLine: latency-aware provisioning and scaling for prediction serving pipelines". In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. 2020, pp. 477–491.

[12] C. Crick et al. “ROS and Rosbridge: Roboticists out of the loop”. In: *2012 7th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*. 2012, pp. 493–494. DOI: `10.1145/2157689.2157846`.

[13] *Docker*. `http://docker.com/`. Accessed: 2021-02-15.

[14] *Faiss library repository*. `https://github.com/facebookresearch/faiss`.

[15] *Feast: Feature Store for Machine Learning*. `https://feast.dev`.

[16] Anshul Gandhi et al. “Autoscale: Dynamic, robust capacity management for multi-tier data centers”. In: *ACM Transactions on Computer Systems (TOCS)* 30.4 (2012), pp. 1–26.

[17] Arpan Gujarati et al. “Serving DNNs like clockwork: Performance predictability from the bottom up”. In: *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 2020, pp. 443–462.

[18] Udit Gupta et al. “The architectural implications of facebook’s dnn-based personalized recommendation”. In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2020, pp. 488–501.

[19] Sami Salama Hussen Hajjaj and Khairul Saleh Mohamed Sahari. “Establishing remote networks for ROS applications via Port Forwarding: A detailed tutorial”. In: *International Journal of Advanced Robotic Systems* 14.3 (2017), p. 1729881417703355.

[20] Jeffrey Ichnowski and Ron Alterovitz. “Concurrent Nearest-Neighbor Searching for Parallel Sampling-based Motion Planning in SO(3), SE(3), and Euclidean Spaces”. In: *Workshop on the Algorithmic Foundation of Robotics (WAFR)*. Springer, 2018.

[21] Jeffrey Ichnowski and Ron Alterovitz. “Motion Planning Templates: A Motion Planning Framework for Robots with Low-power CPUs”. In: *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*. 2019.

[22] Jeffrey Ichnowski and Ron Alterovitz. “Scalable multicore motion planning using lock-free concurrency”. In: *IEEE Trans. Robotics* 30.5 (2014), pp. 1123–1136.

[23] Jeffrey Ichnowski, Jan Prins, and Ron Alterovitz. “Cloud-based Motion Plan Computation for Power-Constrained Robots”. In: *Workshop on the Algorithmic Foundation of Robotics (WAFR)*. Springer, 2016.

[24] Jeffrey Ichnowski et al. “Fog Robotics Algorithms for Distributed Motion Planning Using Lambda Serverless Computing”. In: *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*. 2020, pp. 4232–4238.

[25] *image_transport*. `wiki.ros.org/image_transport`.

[26] *Intel oneDNN library repository*. `https://github.com/oneapi-src/oneDNN`.

[27] Ichnowski Jeffrey, Prins Jan, and Alterovitz Ron. “The economic case for cloud-based computation for robot motion planning”. In: *Robotics Research*. Springer, 2020, pp. 59–65.

[28] Ben Kehoe, Dmitry Berenson, and Ken Goldberg. "Estimating part tolerance bounds based on adaptive cloud-based grasp planning with slip". In: *Proc. IEEE Conf. on Automation Science and Engineering (CASE).* 2012, pp. 1106–1113.

[29] Ben Kehoe et al. "A survey of research on cloud robotics and automation". In: *IEEE Trans. Automation Science and Engineering* 12.2 (2015), pp. 398–409.

[30] Ben Kehoe et al. "Cloud-based grasp analysis and planning for toleranced parts using parallelized Monte Carlo sampling". In: *IEEE Trans. Automation Science and Engineering* 12.2 (2014), pp. 455–470.

[31] Miu-Ling Lam and Kit-Yung Lam. "Path planning as a service PPaaS: Cloud-based robotic path planning". In: *Proc. IEEE Int. Conf. on Robotics and Biomimetics (ROBIO).* 2014, pp. 1839–1844.

[32] Pusong Li et al. "Dex-Net as a service (DNaaS): A cloud-based robust robot grasp planning system". In: *Proc. IEEE Conf. on Automation Science and Engineering (CASE).* 2018, pp. 1420–1427.

[33] Jia Zhi Lim and Danny Wee-Kiat Ng. "Cloud based implementation of ROS through VPN". In: *Int. Conf. on Smart Computing & Communications (ICSCC).* IEEE. 2019, pp. 1–5.

[34] Jeffrey Mahler et al. "Dex-Net 2.0: Deep learning to plan robust grasps with synthetic point clouds and analytic grasp metrics". In: *Proc. Robotics: Science and Systems (RSS).* 2017.

[35] Jeffrey Mahler et al. "Learning ambidextrous robot grasping policies". In: *Science Robotics* 4.26 (2019), eaau4984.

[36] Jeffrey Mahler et al. "Privacy-preserving grasp planning in the cloud". In: *Proc. IEEE Conf. on Automation Science and Engineering (CASE).* 2016, pp. 468–475.

[37] Gajamohan Mohanarajah et al. "Rapyuta: A cloud robotics platform". In: *IEEE Trans. Automation Science and Engineering* 12.2 (2014), pp. 481–493.

[38] Philipp Moritz et al. "Ray: A distributed framework for emerging {AI} applications". In: *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18).* 2018, pp. 561–577.

[39] Raul Mur-Artal and Juan D Tardós. "ORB-SLAM2: An open-source slam system for monocular, stereo, and RGB-D cameras". In: *IEEE Trans. Robotics* 33.5 (2017), pp. 1255–1262.

[40] Tian Nan et al. "A cloud robot system using the dexterity network and Berkeley robotics and automation as a service (BRASS)". In: *Proc. IEEE Int. Conf. Robotics and Automation (ICRA).* 2017, pp. 1615–1622.

[41] Luigi Nardi et al. "Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM". In: *Proc. IEEE Int. Conf. Robotics and Automation (ICRA).* 2015, pp. 5783–5790.

[42] *Nvidia Transformers4Rec repository.* `https://github.com/NVIDIA-Merlin/Transformers4Rec`.

[43] *NVIDIA Triton Inference Server.* `https://github.com/triton-inference-server/server`.

[44] Jia Pan, Sachin Chitta, and Dinesh Manocha. "FCL: A general purpose library for collision and proximity queries". In: *Proc. IEEE Int. Conf. Robotics and Automation (ICRA).* 2012, pp. 3859–3866.

[45] Adam Paszke et al. "Pytorch: An imperative style, high-performance deep learning library". In: *Advances in neural information processing systems* 32 (2019), pp. 8026–8037.

[46] Alyson Benoni Matias Pereira, Ricardo Emerson Julio, and Guilherme Sousa Bastos. "ROSRemote: Using ROS on Cloud to Access Robots Remotely". In: *Robot Operating System (ROS).* Springer, 2019, pp. 569–605.

[47] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: *ICRA workshop on open source software.* Vol. 3. 3.2. 2009, p. 5.

[48] P Ramon-Soria et al. "Autonomous landing on pipes using soft gripper for inspection and maintenance in outdoor environments". In: *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS).* 2019, pp. 5832–5839.

[49] *Ray Seve: Fast and simple API for scalable model serving.* `https://www.ray.io/ray-serve`.

[50] *Redis.* `https://redis.io`.

[51] Francisco Romero et al. "{INFaaS}: Automated Model-less Inference Serving". In: *2021 USENIX Annual Technical Conference (USENIX ATC 21).* 2021, pp. 397–411.

[52] *ROSMaster.* `http://wiki.ros.org/rosmaster`.

[53] Olimpiya Saha and Prithviraj Dasgupta. "A comprehensive survey of recent trends in cloud robotics architectures and applications". In: *Robotics* 7.3 (2018), p. 47.

[54] Vishal Satish, Jeffrey Mahler, and Ken Goldberg. "On-Policy Dataset Synthesis for Learning Robot Grasping Policies Using Fully Convolutional Deep Networks". In: *IEEE Robotics & Automation Letters* (2019).

[55] Haichen Shen et al. "Nexus: a GPU cluster engine for accelerating DNN-based video analysis". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles.* 2019, pp. 322–337.

[56] Gabriel de Souza Pereira Moreira et al. "Transformers4Rec: Bridging the Gap between NLP and Sequential/Session-Based Recommendation". In: *Fifteenth ACM Conference on Recommender Systems.* 2021, pp. 143–153.

[57] J. Sturm et al. "A Benchmark for the Evaluation of RGB-D SLAM Systems". In: *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS).* 2012.

[58] Ioan A. Şucan, Mark Moll, and Lydia E. Kavraki. "The Open Motion Planning Library". In: *IEEE Robotics & Automation Magazine* 19.4 (Dec. 2012), pp. 72–82. URL: `http://ompl.kavrakilab.org`.

[59] Fei Sun et al. "BERT4Rec: Sequential recommendation with bidirectional encoder representations from transformer". In: *Proceedings of the 28th ACM international conference on information and knowledge management*. 2019, pp. 1441–1450.

[60] Ajay Kumar Tanwani et al. "A fog robotics approach to deep robot learning: Application to object recognition and grasp planning in surface decluttering". In: *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*. IEEE. 2019, pp. 4559–4566.

[61] Ajay Kumar Tanwani et al. "RILaaS: Robot inference and learning as a service". In: *IEEE Robotics & Automation Letters* 5.3 (2020), pp. 4423–4430.

[62] *TensorFlow Serving*. `https://github.com/tensorflow/serving`.

[63] Markus Waibel et al. "RoboEarth". In: *IEEE Robotics & Automation Magazine* 18.2 (2011), pp. 69–82.

[64] Jiafu Wan et al. "Cloud robotics: Current status and open issues". In: *IEEE Access* 4 (2016), pp. 2797–2807.

[65] Sarah Wooders et al. *RALF: Accuracy-Aware Scheduling for Feature Store Maintenance*. URL: `https://github.com/feature-store/ralf`.

[66] Binhuai Xu and Jing Bian. "A Cloud Robotic Application Platform Design Based on the Microservices Architecture". In: *Int. Conf. on Control, Robotics and Intelligent System*. 2020, pp. 13–18.

[67] T Ylonen and C Lonvick. *The Secure Shell (SSH) Protocol Architecture*. RFC 4251. RFC Editor, Jan. 2006, pp. 1–29. URL: `https://www.rfc-editor.org/rfc/rfc4251.txt`.

[68] Chengliang Zhang et al. "Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving". In: *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 2019, pp. 1049–1062.