

THE UNIVERSITY OF CHICAGO

CORRALLING THE COMPUTING CONTINUUM:
ENABLING MULTI-SYSTEM WORKFLOWS
WITH SERVERLESS COMPUTING

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
MATTHEW EBEN BAUGHMAN

CHICAGO, ILLINOIS
AUGUST 2025

© 2025 by Matthew Eben Baughman
All Rights Reserved

To all those who have helped me get to where I am today, thank you.

“A person susceptible to ‘wanderlust’ is not so much addicted to movement as committed to transformation.”

—Pico Iyer

TABLE OF CONTENTS

LIST OF FIGURES	viii
LIST OF TABLES	xi
ACKNOWLEDGMENTS	xii
ABSTRACT	xiii
1 INTRODUCTION AND THESIS STATEMENT	1
1.1 Foreword	2
1.2 Motivation	3
1.2.1 Edge	3
1.2.2 AI	4
1.2.3 Data Locality	5
1.2.4 Application and Hardware Specialization	5
1.3 Thesis Statement	6
1.4 Contributions	6
2 BACKGROUND AND RELATED WORKS	8
2.1 Background	9
2.1.1 Grid Computing and the Cloud	9
2.1.2 Serverless Computing	11
2.1.3 Compute Allocations	13
2.1.4 Profiling, Predicting, and Provisioning	13
2.1.5 Wide-Area Scheduling	16
2.2 Related Works	18
2.2.1 Distributed Computing and Serverless	18
2.2.2 Spot Market and Cloud Economics	19
2.2.3 Compute Pricing	19
2.2.4 Performance Prediction and Profiling	21
2.2.5 Distributed Scheduling	24
3 RESEARCH OVERVIEW	26
3.1 Spot Market Analysis and Price Prediction	27
3.2 Workload and Resource Profiling	28
3.3 Distributed Execution of Lambdas with Tradeoff Analysis	29
3.4 Adaptive Task Management	31
4 CLOUD RESOURCE PRICE PREDICTION AND MARKET CHARACTERIZATION	33
4.1 Spot Market Analysis	34
4.1.1 General Statistics	34

4.1.2	Event Frequency and Period	35
4.1.3	Event Occurrence	38
4.1.4	Changes in Market Dynamics	40
4.2	Practical Implications of Changes	42
4.2.1	Methodology	42
4.2.2	Assumptions	43
4.2.3	Results	44
4.2.4	Suitability for Scientific Computing	45
4.3	Spot Price Prediction	46
4.3.1	Pricing Prediction Evaluation	50
4.4	Changes in AWS Spot Market Forecastability	54
4.4.1	Data Description	54
4.4.2	Data Preparation	57
4.4.3	Evaluation	58
4.4.4	Forecastability using ARIMA Models	60
4.4.5	ARIMA Model Results	62
4.4.6	ARIMA Convergence	63
4.4.7	Distribution of ARIMA Accuracy	63
4.4.8	ARIMA Parameterization Accuracy	64
4.4.9	ARIMA Performance Trends	64
4.4.10	Practical Implications	65
5	WORKLOAD PROFILING	
	WITH AUTOMATED EXPERIMENTAL DESIGN	66
5.1	Role of Data in Profiling	67
5.1.1	Profiling Using External Histories	67
5.1.2	Profiling Using Automated Experiments	68
5.2	Profiling Application Performance	70
5.2.1	Application Performance on Cloud Instances	70
5.2.2	Application Performance on HPC	72
5.3	Predictive Performance Models	73
5.3.1	Instance Type Model	74
5.3.2	Scaling Prediction	76
5.3.3	Composite Performance Models	77
5.4	Automated Experimental Design	78
6	DISTRIBUTED EXECUTION OF LAMBDAS	
	WITH TRADEOFF ANALYSIS	83
6.1	Challenges and Problem Formulation	84
6.1.1	Challenges	84
6.1.2	Problem Formulation	86
6.2	DELTA Design and Implementation	88
6.2.1	Core Components	88
6.2.2	External Services	90

6.2.3	Function Execution	91
6.2.4	Endpoint Monitor Implementation	92
6.2.5	Predictors	92
6.2.6	Transfer Manager Implementation	95
6.2.7	Scheduling	95
6.3	DELTA Evaluation	96
6.3.1	Experimental Setting	96
6.3.2	Micro Experiments	97
6.3.3	Macro Experiments	101
6.4	DELTA+: Addressing Key Shortcomings	107
7	ADAPTIVE TASK MANAGEMENT FOR MULTI-SITE SCIENTIFIC WORKFLOWS	111
7.1	System Architecture	112
7.1.1	Infrastructure Control	113
7.1.2	Placement Algorithm	114
7.1.3	Resource Shaping	118
7.1.4	Error Handling and Retries	119
7.1.5	Sample Execution	120
7.2	Methods	121
7.2.1	Testbed Resources	121
7.2.2	Simulator	122
7.2.3	Workloads	123
7.2.4	Baseline Algorithms	125
7.3	Evaluation	126
7.3.1	Simple Load-Balancing	126
7.3.2	Task-Machine Affinity	126
7.3.3	Learning Rate	128
7.3.4	Scheduling Algorithm Comparison	128
7.3.5	ATM Scaling	130
7.3.6	Endpoint Shaping	131
7.3.7	Science Applications	132
7.3.8	Beyond Walltime	133
8	FUTURE RESEARCH	136
8.1	Extensions Addressing Key Limitations	137
8.2	Novel Research Directions	138
9	SUMMARY	139
	REFERENCES	144

LIST OF FIGURES

2.1	Spot price history for c3.2xlarge instances from September 3, 2016, to September 10, 2016, in three US regions. Each series represents a different availability zone.	11
4.1	Number of events per day for each region between July 2017 and April 2018 (with Jan–Mar 2018 zoomed in).	34
4.2	Distribution of time between events for each three month period. Here a log transformation is used to spread the frequency distribution of time intervals more uniformly, aiding both visualisation and data interpretation.	35
4.3	Average daily price by family, in US\$, between July 2017 and March 2018.	36
4.4	Frequency of event occurrences per time of day by day of week.	38
4.5	Frequency of event occurrences per timeslot (minute of hour) by day of week.	39
4.6	Frequency of event occurrences per calendar week by minute of hour.	39
4.7	Cumulative availability of the r3.4xlarge instance type vs. bid price for both the old and new pricing schemes.	41
4.8	Cumulative duration of the r3.4xlarge instance type for both the old and new pricing schemes.	42
4.9	Historical AWS Spot Market price for a given instance over multiple availability zones prior to the pricing change.	47
4.10	Training MSE (USD squared) after ten epochs of training each combination of LSTM width and depth.	48
4.11	Training MSE (USD squared) over 250 epochs.	51
4.13	MAPE range for all models across all instances aggregated by family, generation, and size.	58
4.14	MAPE range for the best model produced for an instance type aggregated by family, generation, and size.	59
4.15	Trends in predictability over time—broken down by family, generation, and size—for average and top performing models respectively.	65
5.1	The execution profile of a Platypus genotyping workflow executed on an m4.4xlarge instance. The starting time of each application is denoted by a vertical dotted line.	69
5.2	Average application performance across instance types.	71
5.3	Execution time for four genomics applications with identical input data, on instance types with varying numbers of vCPUs and varying amounts of memory.	71
5.4	Execution time for the Platypus workflow across instance type, grouped by instance family.	72
5.5	Execution time vs. input data size for six common applications executed on Beagle2. Each dot indicates an execution of the application. The line shows a fitted lower bound.	73
5.6	The fit for the BwaMem application given different resource capabilities.	77
5.7	Combining both the predictive models.	79

5.8	An example of the experiment selection algorithm at work, showing the first four instance types selected as it explores the search space.	80
5.9	Validation of the active experiment design approach.	81
6.1	Relative throughputs (tasks/sec) for three tasks on the resource types of Table 6.1, with values normalized to the maximum observed on any resource.	85
6.2	DELTA architecture, showing components of the scheduling service and the communications required to execute tasks. (1) Submit tasks and fetch results; (2) and (3) supply endpoint and task information to predictors; (4) regularly provide endpoint status; (5) distribute tasks across endpoints; (6) schedule and track data transfers; (7) regularly monitor each task’s status.	89
6.3	Data-transfer rates between Globus endpoints in our testbed (see Section 6.3.1): specifically, from endpoints avg (a desktop on the departmental network) and many (at Argonne National Laboratory) to those endpoints and also fast (another department desktop), aws (an AWS instance in Virginia), and midway (a cluster in UChicago’s Research Computing Center). The circles represent medians of measured values and our model’s predictions are shown by the dotted lines.	94
6.4	Import latencies for four common Python packages on different devices. Download and installation latencies vary similarly.	95
6.5	Learning function performance behavior with different strategies under a small load. Note different scale for round-robin	98
6.6	Accounting for data transfer cost as part of function execution. The workload is 100 Counter tasks; the figure distinguishes between runtime costs, data transfer costs, and all other costs, with the lattermost including time spent waiting for resources.	99
6.7	Tolerating failure of endpoints and slowdown of tasks with automatic delay-detection and backup tasks.	100
6.8	Overloading with ParCount tasks shows that, compared to baseline strategies, EET prediction significantly reduces time-to-completion and boosts throughput. The top figure reports the throughput of each batch. The bottom figure reports the box plot of task time-to-completion within each batch for each strategy. The box shows the first and last quartile; whiskers are shown at 1.5 interquartile range.	102
6.9	In red, fraction of tasks sent to each endpoint when Delta is overloaded with Counter tasks; in black, relative endpoint speed for a Counter function.	103
6.10	Running tasks with multiple input sizes shows that treating different input sizes differently yields increases in performance. When under load, it is beneficial to send smaller tasks to slower endpoints, even Raspberry Pis. Figure a reports throughput when the fastest-endpoint and smallest-EET consider (“with-size”) and do not consider input size in runtime predictions. Figure b shows how the smallest-EET strategy allocates different sizes of tasks to endpoints.	104
6.11	Throughput achieved by different scheduling strategies for the Multiple Heterogeneous Tasks experiment.	105

6.12	Distribution of tasks across endpoints for the Multiple Heterogeneous Tasks experiment. Note different scale for fastest-endpoint	107
6.13	Illustration of optimizing across the tradeoff of resource performance versus cost. In this case, we have constraints of 4hr. maximum time and a \$4 budget (including a \$1 fixed overhead cost). Optimizing for likelihood of constraint satisfaction, the best resources lie along the red line (formed between the origin and the orange dot). Shading illustrates where time (blue) or cost (gray) budgets would be violated.	109
6.14	Simulated task distribution across 8 different endpoints for 10 successive batches of different tasks. Each batch illustrates how the 100 tasks comprising that batch are distributed between the 8 endpoints and this figure demonstrates DELTA's learning capabilities. This simulation was run 1,000 times and the values presented are the mean of those runs.	110
7.1	Overview of ATM's client and operation within a computing environment using Globus Compute.	113
7.2	Demonstration of how proportions of batched tasks become reallocated to different machines over time with respect to ideal machine performance.	125
7.3	Sensitivity analysis of performance impact of learning rate with different levels of task heterogeneity.	127
7.4	Demonstrates how tasks migrate over time to show affinity to the resources for which they are most compatible within an experiment.	127
7.5	Comparison of scheduling algorithms: (a) SEBS batches using Globus Compute; (b) matrix multiply tasks on Jetstream2.	129
7.6	Molecular design experiment across four HPC systems, demonstrating the real science results and how learning how to better distribute those tasks over machines continuously accelerates execution.	130
7.7	Total ATM overhead for different machine and task quantities.	131
7.8	Endpoint shaping on a single machine with both sleep and matrix multiplication tasks. Worker configuration adapts based on batch time for the different tasks. Sleep converges to the number of threads on the machine (8) but matmul converges to 5 workers as the numpy library uses vectorized operations.	131
7.9	Endpoint shaping experiment on the desktop testbed demonstrating the 6.5% reduction in batch time achieved by adding endpoint shaping after task distribution convergence over 20 prior batches.	132

LIST OF TABLES

2.1	Comparison ACCESS Credits required per core-hour for select CPU-based ACCESS resources.	14
4.1	Kolmogorov–Smirnov tests. D is the estimated maximum distance between the two distributions under analysis. Considering a level of significance $\alpha = 0.01$, we reject the null hypothesis of equivalence of the two distributions in all cases. . .	36
4.2	Quantiles, means, and standard deviations of price differences between events, all in US\$. (An asterisk denotes all instance of the highest single price was dropped from the series.)	38
4.3	Root Mean Square Error and Mean Absolute Error for the ARIMA and LSTM models in cents (USD).	53
4.4	Features of the dataset used in this study, in numbers of price events.	55
4.5	Count of the best model type for each period.	58
4.6	Mean instance prices (USD) aggregated by family, generation, and size to provide context of MAPE errors	61
4.7	Mean accuracy (MAPE) across instance types.	64
4.8	MAPE difference (%) between mean and optimal ARIMA parameters over all instance types.	64
5.2	Model accuracy for lower bound fit	78
5.3	Difference in MAPE for models trained with varying numbers of observations from the MAPE obtained by the model trained with all observations.	80
6.1	The heterogeneous testbed used for experiments included 11 devices.	97
6.2	Statistics for Multiple Heterogeneous Tasks experiment: average completion time for each task type, and average makespan both across all batches and across only non-exploration phases. All times are in seconds.	106
7.1	List of non-public resources used for ATM evaluation.	122

ACKNOWLEDGMENTS

I have to thank my lab, my parents, my friends, and my advisors. Without them, my completing this work and this degree would never have been possible. My parents, David and Sara, have supported me unwaveringly in all of my endeavors. My lab, Globus Labs, has become a sort of second home I never imagined I would find in grad school. My friends, too numerous to list, have kept me sane, provided me support and encouragement, and driven me to be the best version of myself. Finally, my advisors, Ian Foster and Kyle Chard, have provided me more opportunities and more guidance than I could have ever expected. It is uncommon for friends to be there for you—let alone PhD advisors—as much as Kyle and Ian have been for me. While saying thank you alone is not enough, "thank you" nonetheless.

ABSTRACT

The computing continuum describes the convergence of global compute infrastructure as network bandwidths increase. To mobilize that infrastructure, we need to create a system that ties these diverse resources together—we need to corral the computing continuum.

This effort began with task-wise solutions, addressing different components of task placement—profiling, predicting, and provisioning. These individual solutions enabled an early system that took into account compute costs, workload execution profiles, and the ability to move compute between systems. We combined and extended these works into a more robust task scheduling system called DELTA and its successor DELTA+. These systems incorporated notions of task execution time, data transfer costs, and machine performance but could not be used on batch scheduled systems or in multi-node environments.

While compute is the currency of the future, there is no unified way to access that currency. To fill this gap, we introduce Adaptive Task Management (ATM)—a framework that acts as a multi-system task manager, mapping tasks to the many resources that comprise the continuum. ATM is designed on top of the Globus Compute framework, using existing infrastructure from edge devices to batch-scheduled HPC systems. ATM includes a novel placement algorithm and novel monitoring and task management systems designed to accommodate both large batches of tasks as well as more complex DAG-based workflows.

To ground and evaluate the development of these frameworks, we explore the application of cost-aware principles in federated learning, material design and protein docking science applications, and in the performance optimization of serverless computing benchmarks.

CHAPTER 1
INTRODUCTION AND THESIS STATEMENT

“When the network is as fast as the computer’s internal links, the machine disintegrates across the net into a set of special-purpose devices.” —George Gilder

1.1 Foreword

The computing continuum is upon us. We have to look no further than our home internet to see that the vast world has been brought to our very fingertips. The existence of this cooperative, emergent, distributed system that we call the internet is not a shock to most of us—we use it daily and appreciate it rarely. Now, in the age of truly astronomically scaled compute workloads—from AI training to molecular simulation—it is not a question of whether we are able to read a file at the speed of light from half a world away; the question is whether it is better to read that file where it lives or move it to a system that may be able to read it faster, more cheaply, or in some other advantageous manner.

This is the idea of the computing continuum—network bandwidth is increasing so rapidly that oftentimes it can make sense to move compute tasks to specialized or more robust devices elsewhere in the world. As George Gilder said, “the machine disintegrates.” In a time where Moore’s law is slowing and compute workloads are growing exponentially, the only way to scale is out. By fluidly computing across the network, we can overcome those constraints (or at least kick them down the road for another few years).

So, how do you access this continuum? Is it the cloud? Not exactly. Is it a cluster or university network? No, neither of those. If not a system that one can log on to, centrally manage, or deploy workloads via a batch schedule system, then what is it? The computing continuum is a paradigm of compute, not a single system. Most individuals use the computing continuum every day. As mentioned above, you can view websites or access files from around the world. This is not a free operation—it requires compute to find and send that information. This is accomplished via a file or web serving system of some sort. These systems have a pre-set list of routines that they run based on user input and requests. What

if these routines, these functions, that run many thousands of miles away could be dictated by you, the end user?

This brings us back to our question—how do we access the computing continuum? The answer may simply be that we figure out how to compute someplace other than our own computer but do so in a manner that takes advantage of the compute we have available, as we need, in a way that abstracts away the actual execution behind the curtain of the machine. Otherwise, this is just running a command over SSH with extra steps.

Ideally, we would access the computing continuum without even knowing. For most individuals, there would be no need. For most, their own laptops far exceed the computational capacity they need on a day to day basis. However, for those with greater needs or for the times when you need just a little extra compute than your laptop from college can offer, the continuum could be used as a near limitless pool of compute that could be bursted out into, enabling whatever work the user may need all without them knowing.

Once the task has left the building, so to speak, the true opportunity of the computing continuum becomes apparent. When you can run a workload wherever, you can decide how to put it there however. This is how we corral the computing continuum.

1.2 Motivation

What makes corralling the computing continuum necessary? We posit it is both the increasing size and diversity of scientific and production workflows—coupled with decreasing marginal gains in hardware performance, generation over generation—that calls for a new way of computing.

1.2.1 *Edge*

The concept of edge computing—the idea that we can and should utilize the copious quantities of data and compute stranded on capable but low-power devices living in our pockets, in

smart cameras, on telephone poles, everywhere—can be immensely enriched with an infusion of the computing continuum. In some cases, such an inclusion is even necessary.

For example (see Patros et al. (2022) [Patros et al. \[2022\]](#)), there could be a device that lives on the edge that has more data than it is able to transfer. In this case, a dynamic decision must be made to compute on part of that data and selectively offload some data. By doing so, data is maximally utilized by computing both on-device and off-device simultaneously. Creating a robust solution to this problem requires characterizing the compute requirements, network capabilities, and data production and storage rates. While performing such characterization could be done manually or heuristically, offloading that configuration to a system that exposes our edge device to the computing continuum, offloading data and compute tasks as necessary, enables new possibilities, especially if we need to reincorporate the learnings from that data, such as in a federated learning scenario.

1.2.2 AI

As use-cases and pervasiveness of AI continues to grow, the underlying demand for compute explodes at an even greater rate. Since the early 2020s, GPU scarcity due to first supply chain issues then the AI revolution has driven massive growth in GPU and semiconductor providers. Projections show that this growth would continue unabated. However, limitations ranging from heat management to power delivery constraints mean that the largest AI workloads cannot live under a single roof. Additionally, the immense quantity of monetary capital required to purchase and maintain very large datacenters make such compute inaccessible to the vast majority of companies, let alone research groups or even individuals.

The computing continuum offers a unique opportunity for AI training (particularly data-parallel AI training) by leveraging GPUs and ASICs (application-specific integrated circuits) from multiple computing locations to be able to work on the same task. Already, we have seen efforts to train large AI models across multiple computing sites, both in industry and

in the National Labs. Extending on this, increasing open source efforts to place AI inference workloads across multiple locations demonstrate the demand for aggregating multiple low-performance resources to achieve otherwise unattainable AI capabilities.

1.2.3 Data Locality

One of the primary drivers for open FaaS frameworks is the ability to configure your own resources with FaaS capability and then, when necessary, push compute to wherever the data are located within your system. This same concept is extensible to ever larger producers of data such as scientific facilities. By enabling this ability to “compute where the data is,” serverless computing embodies the aforementioned idea that compute tasks can be fluid.

This use case perfectly reflects the computing continuum and further motivates its use in such scenarios. By using serverless and pushing the compute workloads to the location of the data (or nearer thereto), we are making an implicit assumption that doing so is more efficient (or cheaper or satisfying another condition) than moving the data to an additional location that may have superior computing resources. These heuristic-based choices must be orchestrated ahead of time and implemented carefully. The role the computing continuum would play is abstracting this away from the end user by simply allowing the user to identify the available resources and data location and allowing the system to orchestrate that compute-by-task-by-machine mapping.

1.2.4 Application and Hardware Specialization

Finally, as previously touched on, there are some devices that are simply better at executing specific workloads than others. In these cases, we want to ensure mapping between the proper devices and applications to ensure optimal system efficiency. Some early work on the computing continuum demonstrated the advantage gained by sending specific parts of a workload halfway across the country and returning a result rather than computing locally

(on one of the world’s most capable HPC systems no less) [Foster \[2019\]](#). This was due to the presence of FPGAs (Field-Programmable Gate Arrays) at the secondary location, enabling a very significant speedup for the specific workload.

The continuum holds promise for automating such offloading capability. Whether it is sending AI workloads to machines with GPUs or sending a highly optimized workflow to an FPGA several hundred miles away, the creation of a system that can learn these specialties and abstract away the decision making from users could significantly accelerate science.

This is the ideal application to end our motivation of the computing continuum on, as it highlights that very ideal laid out by Gilder—each application-specific device, even if hundreds of miles away, becomes just a component in a much larger compute fabric. Just like how the processor in your phone routes integer math to a different location than floating point operations, so too can the continuum utilize these very different global resources to address specific computing workloads.

1.3 Thesis Statement

In this dissertation we will show the following: **By efficiently leveraging geographically distributed and highly heterogeneous resources using novel workload coordination methods, we can significantly improve the execution of multi-site workflows in terms of associated costs and time, thereby accelerating science applications.**

1.4 Contributions

The primary contribution of this thesis is the creation and evaluation of a set of frameworks—DELTA, DELTA+, and Adaptive Task Management (ATM)—that enable efficient, automated, and multi-system task scheduling across the computing continuum. The specific contributions are:

1. A comprehensive analysis of cloud computing market dynamics, including the development of predictive models for AWS spot instance pricing both before and after major pricing scheme changes.
2. A novel methodology for workload profiling that combines historical execution traces with automated experimental design, enabling the creation of robust, composite performance models for heterogeneous resources.
3. The design, implementation, and evaluation of **DELTA** and **DELTA+**—provisioning frameworks that use predictive models and n-dimensional cost-tradeoff analysis to inform multi-system scheduling decisions using serverless computing.
4. The design, implementation, and evaluation of **Adaptive Task Management (ATM)**—a workflow coordination framework that balances the placement of tasks across diverse resources using a cost-aware policy table that is dynamically updated from online execution traces.

CHAPTER 2

BACKGROUND AND RELATED WORKS

2.1 Background

2.1.1 *Grid Computing and the Cloud*

Not dissimilar to the computing continuum, the idea of the computing grid was that, like electricity, you should be able to plug a monitor and keyboard into the wall and have compute metered to you. This idea revolutionized paradigms of compute and eventually gave way to the concept of the cloud. In its modern form, the cloud is just like the grid in that you can get compute metered to you when you need it. You can draw a lot of compute for a short time or a little compute for a very long time.

While the cloud does not realize the seamless access that the grid envisioned (nor, might I add, the true decentralization of compute), infrastructure built on the cloud enables users of all backgrounds and levels of technical know-how to take advantage of its services. Amazon Web Services (AWS) is one of the most pervasive cloud providers in the world. On top of its baseline offerings of more than 400 machine types available to its customers, AWS offers more than 100 different preconfigured tools and systems that operate on top of those cloud resources and leave the configuration up to AWS while the user need only interact with the system they desire.

Additionally, beyond their standard tier of on-demand resources, AWS offers *spot instances*, which are computing resources offered at a discount with the caveat that there is no guarantee of persistence. Essentially, AWS is providing their excess compute at a much cheaper rate with the understanding that, should someone be willing to pay the full price for it, the spot instance will be terminated.

In this way, AWS has created a compute market. At first, this market was transparent, exposing to users every price change for a resource. Sometimes as frequently as several per second, these pricing *ticks* (borrowing “tick” from the stock market terminology to refer to each novel pricing event) showed in real time the exact solution of the supply and demand

equation for AWS’s excess compute capacity.

While AWS always reserved the right to terminate spot instances at will, these arbitrary terminations were rare under the old pricing regime. Generally, most terminations were due to the current price of the instance exceeding the user’s specified maximum bid price (i.e., the maximum amount they stated they were willing to pay for that instance). Due to this mechanism, a sudden full-paying demand for resources would lead to massive spot pricing spikes, at times up to 10x the on-demand price. To understand this intuitively, when there was no more compute available, the price would go high enough to terminate all (or almost all) spot instances.

The inverse was also true—spot instances could be had for as little as one-tenth the on-demand value at the time that spot instance demand was low and spare capacity was high. While we will delve more into the opportunities presented by this pricing regime later (see Sec. 4.4), the massive price swings allowed for motivated users to predict those pricing swings to both wait until the price was adequately low to begin computing and to set the maximum bid at the appropriate level to reach either QoS or expected pricing targets for a given workload. Such opportunity for cheap compute was extremely useful for budget science and other transient or fault tolerant applications.

Spot market data

Amazon provides up to three months of price histories for each instance type, availability zone, and region. This information provides a basis from which one can try to forecast prices and derive patterns. An example of the price history for the `c3.2xlarge` instance type in the `us-east-1`, `us-west-1`, and `us-west-2` regions is shown in Figure 2.1. This figure highlights the significant savings (compared to on-demand instances) possible in the spot market. It also shows the variability of the spot price within and between availability zones.

In order to improve prediction accuracy and our ability to apply machine learning models, we have collected spot pricing data from multiple sources spanning 3 years, with data for all instance types and availability zones in the US regions for the past year. This data is stored as “tick” data: it only includes a point in the series when the price changes from the previous price.

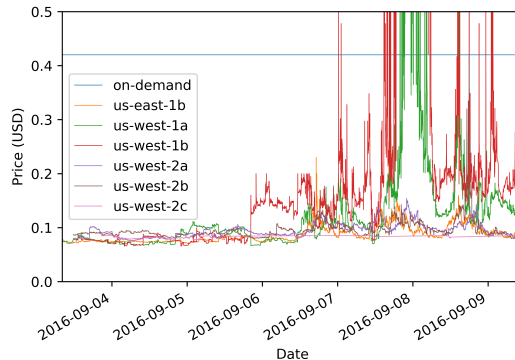


Figure 2.1: Spot price history for `c3.2xlarge` instances from September 3, 2016, to September 10, 2016, in three US regions. Each series represents a different availability zone.

2.1.2 Serverless Computing

Serverless computing is another computing paradigm in which the need to configure or control the hardware for a compute workload is abstracted away from the end user. Serverless employs function-as-a-service (FaaS) platforms to execute compute tasks at the function level. This type of compute was first offered by AWS as their Lambda service back in 2014. Since then, FaaS frameworks have taken off with both service specific and open source variants being very popular.

As the descriptions suggest, some of these frameworks, like Lambda, are tied to one computing platform (e.g., AWS). Others, like OpenWhisk allow users to deploy their own service and route tasks amongst their own devices. Still others, like Globus Compute, offer a publicly available service to queue and route the invoked functions while still allowing users to bring their own compute. In this way, compute resources are siloed to users who have access to those resources but further abstracts away the tedious component of instantiating a service that other FaaS frameworks require.

Globus Compute

Globus Compute [Chard et al. \[2020a\]](#) is a federated function-as-a-service (FaaS) platform designed specifically for scientific computing. Unlike traditional cloud-based FaaS solutions, such as AWS Lambda [Amazon Web Services, Inc. \[2025\]](#), or open-source alternatives like OpenWhisk [Apache Software Foundation \[2025\]](#) and OpenFaaS [Le et al. \[2022\]](#), Globus Compute implements a hybrid cloud-edge model that combines a central cloud-hosted web service with user-deployed endpoints. The cloud service manages functions, endpoints, and tasks via a REST API for users and a message queue for remote endpoints. Users interact with the platform by registering Python functions and invoking them through various interfaces provided by the Globus Compute SDK, such as batch submission and Python’s concurrent futures interface. Globus Compute uses Globus Auth [Tuecke et al. \[2016\]](#) for secure authentication and authorization, supporting diverse identity providers and controlled sharing of endpoints. Endpoints can be shared with specific users under administrative oversight and are associated with managed Globus Groups for enhanced security.

The endpoints in Globus Compute abstract remote computing resources, retrieving tasks from the cloud service and executing them locally. Leveraging the Parsl [Babuji et al. \[2019b\]](#) library, endpoints dynamically provision resources and execute tasks through two constructs: Providers and Engines. Providers handle resource provisioning across diverse systems, including HPC schedulers and Kubernetes, while Engines manage task execution using a pilot job model. This approach enables Globus Compute to support containerized and customized execution environments, scaling dynamically across resource types to meet workload demands. Unlike typical cloud FaaS systems, this architecture extends scaling behavior to hybrid resource environments, offering a flexible and robust solution for computationally intensive scientific workflows.

A unique capability of Globus Compute is its support for multi-user endpoints (MEPs) that allow for remote and programmatic deployment of User Endpoints (UEPs) with arbi-

trary resource *shapes* (e.g., workers, cores, memory and other resources). This ability allows for greater control of endpoints, specifically to modify resources for specific tasks.

We build our system on top of Globus compute for these reasons. The availability of a professionally managed service combine with the flexibility of open source client and endpoint software makes Globus Compute ideal for building out a serverless infrastructure to corral the computing continuum. Additionally, we decided to use a FaaS framework as our base as this paradigm is currently the most well-explored mode of flexible remote computation. Moreover, we believe that performing distributed scheduling at the function level is ideal as high-level user functions are often large enough to represent a substantial amount of compute (i.e., they take more than a few processor cycles to process) while still being small enough that a workflow or bag-of-tasks can be decomposed such that these tasks can saturate multiple machines.

2.1.3 *Compute Allocations*

HPC time is usually granted as Service Units(SUs)—core- or node-hours—tied to a machine but sometimes fungible within a center. Federations, like NSF ACCESS [Boerner et al. \[2023\]](#), now provide fungible allocations that can be redeemed across different systems and sites. These fungible SUs are redeemable at different rates on different systems, presenting a unique opportunity for deploying a system that considers tradeoffs between SUs and time, given each of these systems may have different hardware, queue times, and exchange rates. In Table 2.1, we present the per core exchange rate for several of the CPU-based ACCESS resources to highlight these differences.

2.1.4 *Profiling, Predicting, and Provisioning*

With this new world of resources, we must begin the task of developing methods with which we can use these resources as a single, cohesive ecosystem. To do this, we need to match

Table 2.1: Comparison ACCESS Credits required per core-hour for select CPU-based ACCESS resources.

System Name	CPU model	ACCESS Credits
TAMU FASTER	Intel Xeon 8352Y	0.1248
NCAR Derecho	AMD Epyc 7763	0.9872
SDSC Expanse	AMD Epyc 7742	1.000
TACC Stampede3	Intel Xeon Max 9480	1.674

workloads with resources. More specifically, the diversity in costs and performance presented by different resources requires that we need to understand how these different factors interact and then make a selection of a resource for any given workload. In this thesis, we formalize this problem by breaking it down into a cycle of subcomponents—profiling, predicting, and provisioning. These three steps should address the following three questions respectively:

- Given how quickly a workload executes on one resource, how quickly would it execute on another?
- Given an estimation of execution time on a given resource, how much would it cost to execute that workload?
- Given time and cost estimates, how do we select a resource and deploy the workload with respect to the tradeoffs presented?

One area that has provided a significant increase in the amount of complexity inherent in making this choice is the cloud. Due to this complexity and the configurability offered by commercial clouds, these ecosystems present a perfect test environment for the ideas presented above. For example, Amazon Web Services (AWS) provides nearly 400 distinct *instance types*, each a virtual computer with specific resource capabilities (e.g., CPU, memory, disk, and network). However, without expert knowledge of both the inner workings of an application and the capabilities of all available instance types, it is difficult to select the *best* (from the perspectives, for example, of cost, performance, or reliability) instance type

for a given application. Moreover, selecting the wrong instance type can greatly reduce application performance or be unnecessarily expensive [Chard et al. \[2015a,b\]](#). Thus, the need for expert knowledge is a significant obstacle to effective cloud computing.

Given the enormous search space—a near infinite combination of applications, configurations, and resource types—it is infeasible for users to estimate how an application might perform on a given instance type or with a particular configuration. Instead, new automated methods are needed to predict performance given features describing the application and available instance types. To do this, we will implement a lightweight, modular regression model that allows for use of historical traces and implements an experimental design approach to collect new data. In this way, we will show how we can create an accurate model of execution characteristics on heterogeneous resources.

The usefulness of these workload and machine characterizations, however, require estimates of the costs they incur with respect to cloud or allocation costs, time, data costs, and more. As we discussed in Sec. 2.1.1, there are significant market dynamics at play when determining the pricing of resources. Moreover, the rapidly changing nature of the cloud and of compute resource usage in general increases the complexity of mapping accurate cost estimates to workload deployments.

Regarding provisioning, another way we can conceptualize this is by asking how we can use the data gained above and put it to good use. On its own, without any strategy, we can think of provisioning as easy. At its core, provisioning simply means gaining or utilizing a resource for our specific compute. The most basic form of this is simply running a workload on a machine we have direct access to—running a Python script from the terminal on a laptop, for example. With respect to the ongoing focus we’ve placed on the cloud, provisioning includes reserving some resource in the cloud and then accessing it remotely to deploy a workload. Similarly, HPC follows much the same standard, though generally the cloud requires a payment along with a request and HPC resources require an allocation from

some granting body.

2.1.5 *Wide-Area Scheduling*

Typically, traditional distributed systems are developed and deployed following fixed patterns. The devices they use are generally either cloud resources or have been set up for the express purpose of running that distributed system. Even as distributed systems scale, scaling up and scaling out occur in well-established, heuristic-controlled patterns. This design allows for high assurance, reliable systems that end users can easily interact with (or even forget they’re interacting with).

However, such systems need to be carefully curated and deployed. What they gain in efficiency and quality, they lose in flexibility. When faced with the proposition of extending computational workloads across multiple machines and multiple sites, users would need to consider many different facets such as communication patterns, port access, and filesystem permissions just to name a couple.

One of the most important of these decisions though is deciding which part of the compute job to place on which physical device. Through a choice of one or many different protocols, wide-area scheduling systems attempt to solve this issue by making that choice for the end user.

Wide-area scheduling orchestrates tasks across geographically distributed computing sites [Xhafa and Abraham \[2010\]](#), [Tang et al. \[2023\]](#). This concept emerged in early grid computing, which focused on large-scale resource sharing across organizations [Foster et al. \[2001\]](#). Key objectives of wide-area schedulers include maximizing overall resource utilization, minimizing data movement and latency [Hudson et al. \[2024\]](#), and meeting diverse application requirements across sites [Wu et al. \[2025\]](#). Achieving these goals is challenging due to heterogeneous resources, variable network performance, and machine configuration across many different computing sites.

This paradigm of the computing continuum drives new scheduling strategies that account for extreme heterogeneity and dynamic, geo-distributed conditions. Modern approaches increasingly address conflicting performance, energy, and cost objectives in such environments [Mehran et al. \[2023\]](#), [Kamatar et al. \[2024\]](#), [Zhang et al. \[2014\]](#), [Luo et al. \[2021\]](#). Researchers are also exploring hierarchical and decentralized scheduling models to span multiple tiers of infrastructure [Cai et al. \[2024\]](#), [Mattia and Beraldi \[2024\]](#). These trends demonstrate the importance of wide-area scheduling in harnessing distributed infrastructures for real-world applications.

One of the most popular methods of wide-area scheduling is simply to use work-stealing from a central or distributed queue. Essentially, workers churn through compute tasks and, when they complete one task, they pull another off the queue. More complex versions of this exist with capabilities like prefetching (estimating the end of a compute task so that you can begin loading the new task prior to sitting idle) or peer queue stealing (a special case of decentralized queue where every worker is assigned a set of tasks and workers can pull from other workers' queues when their own becomes depleted). Work stealing can be referred to as a pull-driven system as the workers pull for tasks when they are ready for new tasks.

Another popular pattern for wide-area scheduling is distributed load balancing. In this approach, the central work broker maintains some notion of load for each of the workers and distributes tasks accordingly. These metrics can vary and will be explored more in depth later in this paper (see Sec. 7). For context, however, two such load-balancing algorithms include round-robin—where the metric can be quantified as the number of tasks sent or the time since last task (both approximated)—or least-loaded—where tasks are sent to workers based on either their total capability (e.g., number of cores) or their level of utilization (not dissimilar to work stealing). In this manner, the load balancing we describe is representative of a push-driven scheduling system, wherein tasks are sent to workers based on some centralized metric. This is the pattern we will be exploring in this work. We will further explore systems

that use these various methods in following sections (see Sec. 2.2.5).

2.2 Related Works

2.2.1 *Distributed Computing and Serverless*

On its own, Globus Compute [Ananthakrishnan et al. \[2024\]](#) enables users to distributed fire-and-forget functions at their will, simply specifying the task, relevant arguments, and destination computing resource. However, the key issue here is the specification of the final resource as users, and particularly domain science users, do not want to spend time manually allocating tasks to resources. Thus, there is a need for a management layer.

More widely, there are similar distributed computing frameworks such as the highly disaggregated BOINC and Folding@Home networks [Anderson \[2021\]](#), [Voelz et al. \[2023\]](#) that distribute tasks to idle resources.

Public cloud provider FaaS capabilities [Amazon Lambda](#), [GCP Functions](#), [Azure](#) are deployed on homogeneous cloud infrastructure, with the exception of Amazon Greengrass that can be deployed locally for IoT use cases [Amazon Web Services \[2025\]](#). Open-source FaaS solutions, such as OpenLambda [Hendrickson et al. \[2016\]](#), Apache Openwhisk [Apache OpenWhisk.](#), and Kubeless [Kubeless.](#) enable FaaS deployments on local resources, often using Kubernetes clusters for container provisioning.

Recent FaaS research has focused on challenges associated with centralized deployments and on reducing cold-start times [Oakes et al. \[2018\]](#), [Wang et al. \[2018\]](#). For example, Azure shows cold-start latencies of up to 3500ms, whereas AWS uses optimized Firecracker virtualization [Firecracker.](#), and likely maintains warm virtual machines, to hide some cold-start costs.

2.2.2 *Spot Market and Cloud Economics*

Prior to the change in pricing and termination policy it was often believed that the spot market prices were mostly demand-driven. However, researchers were able to show that market demand does not account for the entirety of price changes that can be observed, and that observed prices were driven by (hidden) externalities [Agmon Ben-Yehuda et al. \[2013\]](#), [Xu and Li \[2013\]](#). While a purely demand-driven model was not empirically observable, the introduction of the new pricing policy moves spot market pricing even further away from demand-driven fluctuations. Notably, adjustments to the spot prices have become smoother following the policy change [Baruwal Chhetri et al. \[2018\]](#). While the average spot price has decreased, the real cost has increased as artificial peaks included before the change were likely not market driven [George et al. \[2019\]](#).

Examples of bidding strategies and frameworks to harness these spot instances include minimum bid estimations under price and availability constraints [Lumpe et al. \[2017\]](#), revenue maximization models [Xu and Li \[2013\]](#), bidding strategies taking into account if jobs can be interrupted [Zheng et al. \[2015\]](#), and integration of fault-tolerance policies [Voorsluys and Buyya \[2012\]](#). Some strategies have even included the use of a metamarket in the form of virtual options contracts to manage the conditionality of working within an unpredictable market [Keller et al. \[2020\]](#) with great success in selecting spot instances based on market pressures. In addition, the uncertainty in the availability and runtime of spot instances was curtailed by using probabilistic guarantees of instance availability for different execution durations [Wolski et al. \[2017\]](#). Other work [Danysz et al. \[2020\]](#) expands this concept to the management of spot instance pools.

2.2.3 *Compute Pricing*

A wide range of predictive models have been explored to predict spot prices. For example, researchers have used parametric models based on exponentials fitted using Expectation-

maximization (E-M) [Javadi et al. \[2011, 2013\]](#) maximizing price expectations; Constrained Markov Decision Processes to minimize the expected cost of an instance, taking into account various costs, including checkpointing and restart delays [Tang et al. \[2012\]](#); and organized the problem as a cost minimization problem based on a Markovian state model that estimates transition probabilities from spot price histories [Guo et al. \[2015\]](#). Closely related prediction models have been developed to support application migration and checkpointing to avoid service downtime [He et al. \[2015\]](#), [Sharma et al. \[2015\]](#), [Subramanya et al. \[2015\]](#). In these cases, schedulers capable of migrating services or applications use prediction models to predict when instances will be terminated.

Little prior work has investigated the prediction of spot prices using artificial intelligence. In fact, apart from our work, we have found few other publications that investigate the application of neural networks to spot price prediction [Wallace et al. \[2013\]](#) and those that have used simple, MLP-based models [Turchenko et al. \[2014\]](#). However, there has been significant interest in using neural networks for effective prediction of stock market prices [Ticknor \[2013\]](#), [Sheta et al. \[2015\]](#). Statistics-grounded time series analysis methods have been applied widely to market prediction [Sheta et al. \[2015\]](#), [Sands et al. \[2015\]](#) and specifically to spot prices [Chhetri et al. \[2017\]](#).

The ability to rapidly provision spot instances makes them particularly attractive as a way of obtaining elastic resources at low cost [Chard et al. \[2015a\]](#). The value of these low cost instances has been exploited in genomics [Madduri et al. \[2014\]](#), food security [Montella et al. \[2015\]](#), brain imaging [Chard et al. \[2014a\]](#), and social science [Babuji et al. \[2016\]](#). In these cases, the automated provisioners used to dynamically obtain and release spot instances [Chard et al. \[2017\]](#) rely on methods to predict resource prices or assumptions of persistence to configure and deploy these instances.

2.2.4 Performance Prediction and Profiling

There has been considerable work to determine the execution requirements of applications, most often in high performance and distributed computing environments [Berman et al. \[2001\]](#), [Smith et al. \[2004a\]](#), [Matsunaga and Fortes \[2010\]](#), [Tang et al. \[2013\]](#), [Do et al. \[2011\]](#), [Wood et al. \[2008\]](#). Recently, researchers have begun to explore predictive modeling as a solution to the instance selection problem. PARIS [Yadwadkar et al. \[2017\]](#) is a data-driven system that aims to select the best instance type given minimal data collection. PARIS uses a hybrid online and offline data collection and modeling system to accurately predict workload performance. However, unlike our work, PARIS focuses on video-encoding workloads and serving latency. Our work targets scientific uses cases that differ substantially from many business-oriented cloud uses. For example, the genotyping workflow that we evaluate here consists of extremely diverse applications with a wide range of computational requirements. CherryPick [Alipourfard et al. \[2017\]](#) uses Bayesian Optimization to model application performance to determine the best, or close-to-best, instance type with minimal experiments, while other work [Liu et al. \[2019a\]](#) uses Pareto analysis to select necessary resources from a set of efficient options based on specified costs. Ernest [Venkataraman et al. \[2016\]](#) is able to predict the execution time of distributed analytics jobs based on cluster size. However, unlike our approach, Ernest cannot forecast performance without previously deploying an application on a given resource. Paragon [Delimitrou and Kozyrakis \[2013\]](#) and Quasar [Delimitrou and Kozyrakis \[2014\]](#) employ historic execution traces to rapidly classify new applications. In combination with a small amount of online profiling, Quasar is able to manage resource allocations and guide job packing. However, these traces do not incorporate fine-grained resource utilization statistics. To incorporate these, the Arrow framework [Hsu et al. \[2018\]](#) incorporates low-level performance statistics during its iterative optimization. This framework uses these statistics along with the standard runtime and cost statistics to inform a Bayesian optimization process to reduce the fragility of the optimization system

and to more efficiently reduce “areas” of uncertainty in the Bayesian process. They have seen promising results that have overcome many of the previously observed shortcoming found in Bayesian optimization when applied to workflow profiling.

The problem of selecting suitable resources for a given application is ubiquitous across cloud providers. Google now provides a *VM sizing recommendation* service [goo](#) that monitors system utilization and recommends appropriately sized virtual machines. Although such a service relies on understanding prior execution information, it does not enable forecasting for variable usage requirements (e.g., in terms of applications, data sizes, settings), or let users explore trade-offs between application performance and cost.

Micro-benchmarks have also shown great promise to aid in predicting the performance of a given workload in a foreign computing environment [Scheuner and Leitner \[2018\]](#). This is accomplished by using a suite of benchmarks to evaluate the unknown compute infrastructure and then use the execution characteristics of those benchmarks on the known infrastructure (i.e., where the execution characteristics of the workflow are known) to predict the workflow’s execution characteristics in the new environment. Similar work exists in the area of selecting resources from across different cloud environments [Liu et al. \[2019b\]](#). Others have also sought to learn application performance models from data obtained from production environments [Bodik et al. \[2009\]](#), while trace specific clustering has been used to select cloud resources based on workflow classifications requiring varying computational needs [Guo et al. \[2019\]](#).

There has been some recent work to explore location-aware computing [Akkus et al. \[2018\]](#), shared hosts with multiple workers [Shillaker and Pietzuch \[2020\]](#), and dynamic provisioning [Sreekanti et al. \[2020\]](#) in a FaaS context. Dynamic provisioning in spot markets has been explored more in depth in several works [Shastri and Irwin \[2017\]](#), [Hasan and Hammad \[2020\]](#), [Wong et al. \[2021\]](#), where dynamic allocation of spot instances following termination notifications is shown to significantly reduce downtime and cost. However, these methods account

for underlying market pressures by hopping between availability zones based on availability rather than overall cost. Additionally, there is significant work with respect to cost-aware provisioning of resources with respect to differing costs and qualities of service (QoS) [Chard et al. \[2017\]](#), [Guo et al. \[2014\]](#). Specialized research into hierarchical applications and distribution have been explored for use of terminable cloud instances [Davatz et al. \[2017\]](#). Lastly, many of these approaches focus on application or system specific implementations, which informed our design choices to include lightweight and portable solutions and build on existing frameworks that function across multiple types of resource and computational domain.

Lin and Khazaei [Lin and Khazaei \[2020\]](#) address the need for accurate performance and cost prediction in serverless architectures. They introduce a novel analytical model alongside the Probability Refined Critical Path (PRCP) algorithm to optimize serverless applications deployed on AWS, showcasing significant improvements in efficiency and cost-effectiveness.

The Costless framework [Elgamal et al. \[2018\]](#) focuses on cost optimization in AWS Lambda by employing function fusion, placement strategies, and memory allocation. Using a constrained shortest path problem within a cost graph, the framework effectively balances cost and latency, ensuring optimal resource utilization for serverless deployments.

Ali et al. [Ali et al. \[2024\]](#) proposes using function profiling to inform the deployment of serverless functions in cloud continuum environments. By analyzing a function’s performance across various runtime settings, their proposed system aims to improve deployment decisions, leading to better QoS for users and optimized resource utilization for providers.

Additional work [Maros et al. \[2019\]](#) has been done with application-specific platforms that explores the use of supervised machine learning (ML) models to predict the performance of Spark applications. The authors find that their "black box" models, using features available before execution, match or exceed the performance of the existing Ernest model, especially for applications with irregular patterns or when extrapolating to larger datasets. They conclude that while there is no single best ML technique, different techniques should be explored for

each scenario to select the most accurate model

Lastly, Galimberti et al. [Galimberti et al. \[2023\]](#) introduce an auto-profiling framework, OSCAR-P, designed for the open-source serverless platform OSCAR. This tool automates the testing of workflows across diverse hardware configurations and generates machine-learning-based performance models, prioritizing efficient resource allocation.

2.2.5 Distributed Scheduling

Wide-area compute scheduling presents unique challenges in heterogeneous environments. DELTA [Kumar et al. \[2021a\]](#) distributes function-based workloads across serverless computing endpoints using individual performance predictors to select optimal resources. Similarly, Parsl [Babuji et al. \[2019b\]](#) builds dynamic dependency graphs to scale execution over thousands of workers while providing elastic resource provisioning and fault tolerance.

TaskVine [Sly-Delgado et al. \[2023\]](#) focuses on data-intensive workflows, leveraging data locality and caching to minimize transfers and optimize task execution. In addition, Iridium [Pu et al. \[2015\]](#) optimizes task placement by jointly considering computation and data movement, reducing WAN latencies through intelligent replication, and pre-staging data. Phare [Castellano et al. \[2024\]](#) employs a decentralized, affinity-based scheduling strategy to balance tasks across federated edge clusters, thereby minimizing cross-site communication overhead and ensuring robust performance even in the face of failures.

Scheduling for deep learning (DL) workloads has also seen significant innovation. Gandiva [Xiao et al. \[2018\]](#) introduces an introspective framework that leverages the predictability of DL training iterations to enable efficient time-slicing and dynamic job migration, improving cluster utilization and early feedback for hyper-parameter searches. Building on this, Gavel [Narayanan et al. \[2020\]](#) proposes a heterogeneity-aware scheduler that systematically generalizes existing scheduling policies by expressing them as optimization problems over a job's "effective throughput." More recently, Pollux [Qiao et al. \[2021\]](#) introduces a co-adaptive

approach that simultaneously optimizes cluster-wide resource allocation and per-job training parameters (e.g., batch size and learning rate) by maximizing "goodput," a metric that combines system throughput with statistical efficiency.

FnSched [Suresh and Gandhi \[2019\]](#) seeks to maximize utilization while meeting service-level objectives (SLOs) by regulating function resource usage and scaling utilization of resources based on load. Other work on meeting SLOs in heterogeneous computing environments includes a probabilistic *task pruning* method [Denninnart et al. \[2019\]](#), which uses a function’s execution history to predict whether a task will meet its SLO. Reuther et al. quantify scheduler latency across Slurm, Mesos, Hadoop, YARN, and others, and show that multi-level “array” schedulers such as LLMapReduce can improve utilization for bursty, sub-minute jobs [Reuther et al. \[2018\]](#). Wukong implements a decentralized model for scheduling functions expressed in a workflow DAG on FaaS platforms [Carver et al. \[2020\]](#). Additionally, Maheswaran et al. [Maheswaran et al. \[1999\]](#) defined significant groundwork in heterogeneous task allocation.

CHAPTER 3

RESEARCH OVERVIEW

3.1 Spot Market Analysis and Price Prediction

We began this work by examining the price dynamics of cloud computing markets created by top providers. We analyzed these markets with respect to how they change over time and how those changes effect predictability of those markets. Our analysis revealed the substantial reduction in market variability following a substantial overhaul of market pricing strategies from the primary resource provider—Amazon Web Services. However, we developed predictive models that were able to accurately predict market pricing both before and after this pricing paradigm shift.

Prior to the change, our predictive methods relied on somewhat complex LSTM-based neural networks developed through a neural architecture search process. While these models were relatively lightweight and did not require substantial levels of compute to train, their development was still nontrivial in order to achieve leading performance. Following the change, we performed a much more substantial analysis of market predictability and demonstrated that the price smoothing features added to the market by AWS lead to a prediction task that did not benefit from the use of more complex neural network-based models. Instead, we were able to use more traditional ARIMA-based models to achieve comparable performance in most cases.

While the ARIMA-based methods provided performant results at low computational cost, we found that parameterizing ARIMA models incorrectly could have substantial negative effects on the predictive performance of the models. To address this, we trained and analyzed the performance of over 2 million ARIMA models on our cloud pricing dataset. We found that the performance of the models was usually consistent but prone to extreme outliers when poorly parameterized. Most significantly, our analysis demonstrated that, under the new pricing regime, most models achieved near-perfect performance, making the field of AWS spot market pricing prediction a trivial task. Regardless, while predicting resource pricing became trivial, the introduction of much higher rates of *pressure-based terminations*

compared to *price-based terminations* made AWS spot instance less reliable than prior to the change, given our price prediction strategies.

3.2 Workload and Resource Profiling

Building on the ability to accurately predict resource pricing given an estimate of execution duration on a resource, it then became necessary to have better estimates of runtime. To do this, we began developing profiling methods for running workloads on AWS instances and on HPC systems. Due to their dependence on input data, high levels of task execution variability, and significant computational requirements, we decided to use genomics analysis workflows as the target application for this profiling work.

We began by creating models to characterize the runtime of each component task in the workflow with respect to some resource-level variable, such as CPU core count, RAM availability, or input data size. These models were simple polynomial regressions. There were three key research challenges in actually incorporating these models into a system that could be used to accurately and robustly predict runtime—1) integrating these models into a unified system that can be used to predict runtime out of sample; 2) dealing with extremely noisy and variable runtime data due to severe resource contention on HPC systems; and 3) developing an approach to reproduce these profiling results for novel workloads.

The first challenge was incurred due to these models characterizing runtime with respect to only one variable at a time. Following, combining models—say, one for CPU core count and one for RAM availability—is difficult as the two regressions intersect at only one point in the 3D prediction space. To address this, first, we combine the two models by simply solving algebraically for their intersection. However, this produces very poor results in the out of sample prediction space as it is not able to capture the joint effects of the two variables (i.e., based on the individual models, there is no indication what the effect on runtime is when increasing or decreasing both variables simultaneously). Therefore, we elect to retrain

the composite model with a single additional experiment parameterized by the mean of the out-of-sample space. By fitting the composite model to this additional data point, we were able to achieve results of less than 30% error across the entire parameter space.

To compensate for extremely noisy data, we elected to fit the models to the 10th percentile of the runtime distribution for each task. By doing this, we approximate the runtime of the task as the likely runtime without the effects of resource contention. Doing so allows for predictive accuracy of 7% when excluding contention effects, which are generally considered stochastic and unpredictable.

Finally, to enable the reproducibility of these profiling methods to other workloads, we developed an automated experimental design approach to select an optimal set of points in the parameter space to profile. This approach was inspired by the refitting of our composite model by selecting a point based on where there was the least amount of model support. Extrapolated across the entire parameter space, our automated experimental design approach selects the next experiment based on algorithmically solving for which point would most constrain the error bounds of the model based on new information at that point. Effectively, if the new experiment confirms our current model, error bounds are universally tightened. Conversely, if the new experiment contradicts our current model, error bounds on the recalibrated model are disproportionately widened elsewhere in the parameter space, further guiding the selection of the next experiment. With this approach, we were able to achieve within 3% of the optimal error bounds for the entire parameter space with half of the experiments and within 30% with only 30% of the experiments.

3.3 Distributed Execution of Lambdas with Tradeoff Analysis

With the ability to accurately predict runtime and resource pricing, we distilled the application of these methods into what we call the *profiling, predicting, provisioning* cycle. This cycle functions as a feedback loop to run workloads, collect information on that execution,

and then update models predicting that execution and use those models to update cost estimates. With two of the three components built, we next began to develop a system to coordinate this cycle and to interface with the resource provider to provision resources. This system was called DELTA—Distributed Execution of Lambdas with Tradeoff Analysis. DELTA integrated micropredictors, similar to those in the profiling work, combined with a cost model, to select the optimal resource for task execution. DELTA was build on top of a federated function- as-a-service framework called funcX as the serverless execution paradigm best fit the flexibility needed to rapidly move tasks to different resources.

DELTA created a set of models for each task-resource pair, predicting the execution time, data transfer time, resource startup latency, system overhead, and total execution time. These models were then used to minimize the total execution time of submitted tasks or batches of tasks. We evaluated DELTA on a testbed of devices ranging from Raspberry Pis to HPC nodes. DELTA not only was able to distribute tasks across the entire testbed (ensuring that the computational power of an SBC can be used alongside HPC systems), but also was able to move tasks to resources on which they were most well-suited. In this manner, DELTA motivated the idea that the computing continuum is not simply load-balancing across distributed resource, but rather requires the intelligent mapping of tasks to resources.

Due to DELTA’s reliance on micropredictors, focus on execution time alone, and lack of consideration of multithreading, we developed two extensions to DELTA that we collectively called DELTA+. The extensions are as follows: 1) a multidimensional cost model that takes into account more than just task rount trip time and 2) the abiliity to dynamically provision additional resources if cost-effective to do so. In keeping with the core scheduling strategy of DELTA, we relate multidimensional costs to each other as to minimize the weighted product of the costs. This allows us to specify the relative importance of each cost to the overall cost of the system (e.g., specifying that \$5 is worth reducing runtime by 20 minutes). Next,

we extended this to a more robust roofline optimization model that solved the optimization tradeoffs based on specified budget constraints. This could be time, money, or allocation budgets. We used this new model to decide whether to provision additional resources to the system, based on estimates of marginal time and monetary costs. If a resource’s cost and performance place it closer to the “peak” of the roofline, we would provision additional resources to the system. We evaluated DELTA+ through simulation to isolate the framework’s ability from resource contention and other stochastic effects. We found that DELTA+ was able to seamlessly schedule tasks using the updated cost model and provision additional resources via the necessary connectors. Additionally, by establishing a single prediction model for each task-resource pair, we were able to substantially reduce the overhead of DELTA+ compared to the original system.

3.4 Adaptive Task Management

Finally, to build on the lessons learned from DELTA+ and to create a more generalizable system, we developed a framework called ATM— Adaptive Task Management. ATM is a framework also built on top of a serverless computing platform called Globus Compute (the successor to the now deprecated funcX). ATM uses a similar cost model to DELTA+ but extends it to map tasks over the entire resource space rather than scheduling tasks with individual prediction models. This new method relies on historical traces and performance interpolation to generate a policy table for mapping tasks to resources. Additionally, ATM uses probabilistic scheduling to allow for a slightly more directed approach to navigating the exploration-exploitation tradeoff. By using only a single model to map historical traces to resources, ATM’s overheads are substantially reduced. This allowed us to demonstrate scaling up to scheduling more than 16,000 task types across more than 16,000 resources. One drawback of this approach is that by eliminating the individual prediction models, we lose the ability to quickly find optimal resource mappings for specialized tasks. However, ATM’s

ability to rapidly rebalance and take a holistic view of all available resources far outweighed this drawback. Additionally, we were able to demonstrate that ATM can accelerate real science workflows on 100 nodes of an HPC system and on more than 10 different computing sites simultaneously.

Two important additions we included within ATM are its ability to dynamically shape resources—determining the number of tasks to pack simultaneously onto a single endpoint—and its enhanced capabilities with respect to navigating the cost tradeoffs, demonstrating the ability to drop resources if they are underutilized and to trade small amounts of performance over time for substantial cost savings.

We also evaluated the scheduling component of ATM against other distributed scheduling frameworks and found that ATM meets or exceeds their overall performance while maintaining less batch-to-batch variability over the course of an experiment and better taking into account cost tradeoffs. We believe this demonstrates that ATM is a state-of-the-art framework for coordinating large science workflows across multiple computing sites and that this motivates its use to accelerate scientific computing.

CHAPTER 4
CLOUD RESOURCE PRICE PREDICTION
AND MARKET CHARACTERIZATION

4.1 Spot Market Analysis

We now present a before-after analysis of the spot market. We begin by summarizing general statistics including the mean price and variability, we then analyze the frequency of event changes, before reviewing changes to market dynamics.

4.1.1 General Statistics

The primary change, both stated by AWS and evident in the spot pricing, is the reduction of variability in the pricing data. Overall, across our 841 instance type and availability zone combinations, the average instance is 11.55% cheaper than it was before the change, though the deltas vary wildly between instance types. While this increase in mean price is significant, the reduction of spot price variability represents a much much more noticeable change. Before the change, the mean standard deviation was 87.9% of the mean spot price; following the change, the mean standard deviation was just 7.9% of the mean spot price. The most notable difference is related to the duration of a given spot price: after the change, spot price persisted, on average, more than 85x longer than they did before the change. This increase is consistent with AWS’s claim that spot pricing is more consistent and that the price is regularized to fit more long term trends.

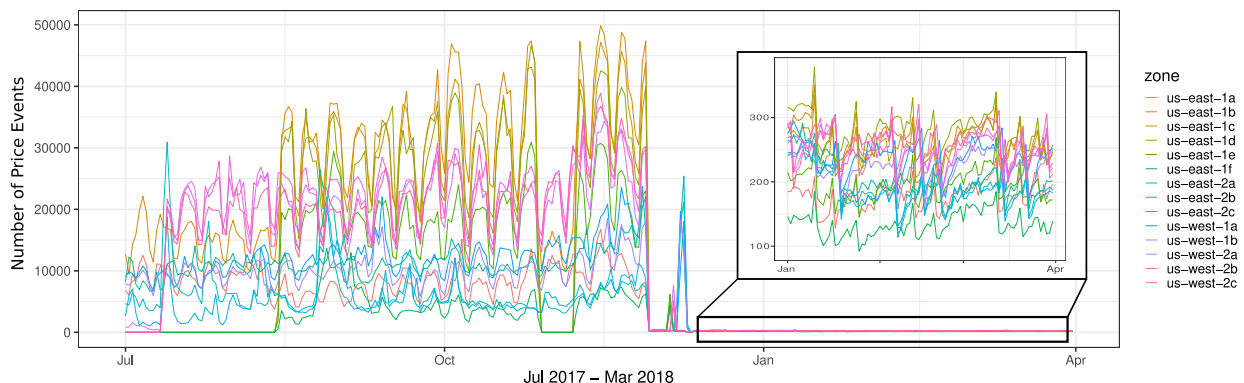


Figure 4.1: Number of events per day for each region between July 2017 and April 2018 (with Jan–Mar 2018 zoomed in).

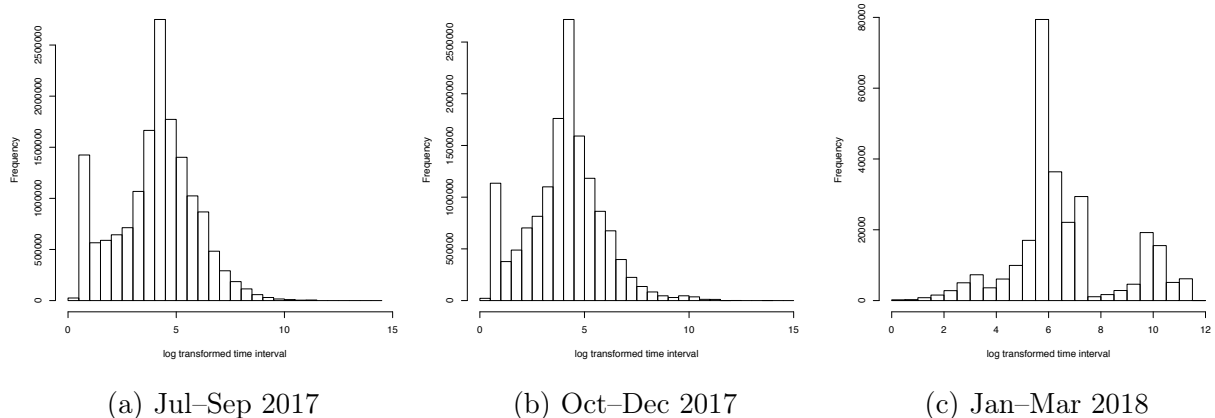


Figure 4.2: Distribution of time between events for each three month period. Here a log transformation is used to spread the frequency distribution of time intervals more uniformly, aiding both visualisation and data interpretation.

4.1.2 Event Frequency and Period

The number of price updates and thus price events drops dramatically following the switch to the new policy, as shown in Figure 4.1. The spike in late November suggests an amount of reorganization at that time, perhaps as AWS further tuned its pricing algorithm.

Some regions have significantly more price updates than others. This is true for both different instance types and families. Therefore, the update to the AWS pricing mechanism appears to increase the time to algorithmically “consider” a price change. Figure 4.2 illustrates how the duration between events has changed over time. For ease of presentation, time between events (which follows an exponential distribution) has been log transformed. The distribution of intervals shows that there are many small intervals and a small number of extremely large intervals. The extent of these extreme outliers (right tails of the distribution) can be seen in Figures 4.2a and 4.2b. Also notable is that the frequency of events (y-axis) is much lower in Figure 4.2c. Two-sample Kolmogorov-Smirnov tests are used to check if the time between events follow similar distributions, i.e. if price changes occur similarly before and after the change in pricing mechanism. Table 4.1a indicates that the time between events has changed significantly. However, it is worth noting that based on the

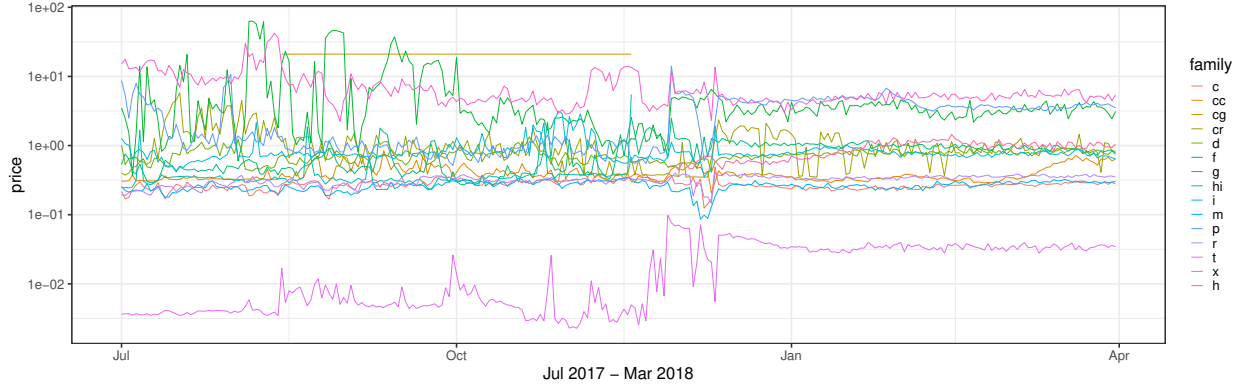


Figure 4.3: Average daily price by family, in US\$, between July 2017 and March 2018.

size of our data, even subtle changes would be considered significant. Instead, we note that Jan-Mar 18 is further away, i.e., more different to Jul-Sep 17 and Oct-Dec 17 than they are from each other, as indicated by a higher D statistic. This is corroborated in Figure 4.2.

Table 4.1: Kolmogorov–Smirnov tests. D is the estimated maximum distance between the two distributions under analysis. Considering a level of significance $\alpha = 0.01$, we reject the null hypothesis of equivalence of the two distributions in all cases.

(a) Kolmogorov–Smirnov tests inspecting the time between events.

Time Periods	D	p-value
Jul–Sep 17 / Oct–Dec 17	0.03060	< 0.001
Jul–Sep 17 / Jan–Mar 18	0.60879	< 0.001
Oct–Dec 17 / Jan–Mar 18	0.62835	< 0.001

(b) Kolmogorov–Smirnov tests inspecting the change in price between events.

Time Periods	D	p-value
Jul–Sep 17 / Oct–Dec 17	0.03160	< 0.001
Jul–Sep 17 / Jan–Mar 18	0.26168	< 0.001
Oct–Dec 17 / Jan–Mar 18	0.24311	< 0.001

The reduction in the number of events prompts the inspection of the effect on price itself. Figure 4.3 depicts the average daily price (across all instance types, generations, sizes and regions). Here, however, we need to be cautious in the presentation of the data. The price signal has, in general, a high variance, and significant outliers (as already seen) that

have high leverage over measures like the mean (plotting the median, however, does not look fundamentally different). We cannot show error bars indicating one standard deviation around the mean as they make Figure 4.3 unreadable. Similar to the number of events, the mean of the more expensive families has reduced and there is less variance with time. This is to be expected as newer instance families will reduce in price over time.

Similar to the time between events, we see significant differences in the change of price captured by events, see Table 4.1b, with the same remarks with respect to statistical power, and the difference in distance between the distributions (denoted by the D statistic), i.e. the Jan–Mar 2018 distribution is further away (more different).

In summary, the change in pricing mechanism reduced the number of events (Figure 4.1), increased the time between events (Table 4.1a and Figure 4.2), and reduced the degree of price change between events (Table 4.1b and Table 4.2). Table 4.2 also includes statistics after removing occurrences of the artificial spot price cap set by AWS (10x the on-demand price) as these prices may represent situations in which AWS has potentially artificially raised the spot price to terminate running spot instances [George et al. \[2019\]](#). In doing so, we eliminate the influence of extreme values and observe reduced overall variation, but still much greater variation than the new pricing model.

We also note that the average change in price represented by an event is negative, which would suggest a general reduction of price over time. Yet more notable are the extremes of price change. The increase in time between events seems to afford AWS the opportunity to be algorithmically less variable with respect to changes in instance costs. This is seen in the reduction in the standard deviation of changes to price between events. As such, it appears that price adjustments now occur less often, are smaller, and have lower variance. These changes indicate a benefit to users requiring instances for prolonged periods of time (reduced variance and increased price durations means more predictable costs), while detrimentally affecting users that opportunistically seek short-term use of cheap instances.

Table 4.2: Quantiles, means, and standard deviations of price differences between events, all in US\$. (An asterisk denotes all instance of the highest single price was dropped from the series.)

	0%	25%	50%	75%	100%	mean	stdev
Jul–Sep 17	-255.35	-0.0002	0.0001	0.0002	255.1	-0.0000	7.936363
Oct–Dec 17	-262.671	-0.0002	0.0000	0.0002	263.2583	-0.0004	3.2520
Jan–Mar 18	-5.0264	-0.0001	0.0000	0.0001	2.8491	-0.0002	0.0310
Jul–Sep 17*	-71.7526	-0.0003	0.0001	0.0004	72.2574	0.0000	0.2622
Oct–Dec 17*	-70.1337	-0.0003	0.0001	0.0004	72.1944	0.0000	0.3003
Jan–Mar 18*	-5.0264	-0.0001	0.0000	0.0001	2.8491	-0.0002	0.0310

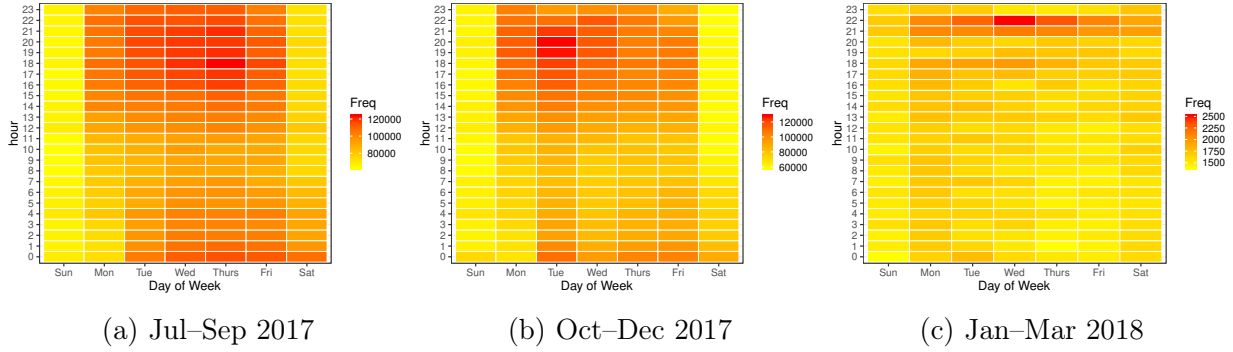


Figure 4.4: Frequency of event occurrences per time of day by day of week.

4.1.3 Event Occurrence

Having observed a change in the number of events, a change in time between events (it increases), and the price difference between events, we can also observe that events occur following different patterns. Figure 4.4 presents a heat map that corresponds to the frequency of events with respect to the time of day and day of week that events occur. It is evident that there has been a change in the weekly pattern of events. From Jul–Dec 2017 (Figures 4.4a, 4.4b) there is a clear seasonality in the event distribution: fewer events occur at weekends, and there is a slight tendency for more events at later hours. Note that UTC is 4 to 8 hours ahead of US time zones.

We observe that in Figure 4.4c (Jan–Mar 2018) this pattern disappears completely. Aside from a general reduction in events, the distribution of events over the 24 hour period, as well as over the week, appear more normalized. Some of the weekly pattern remains: Sunday

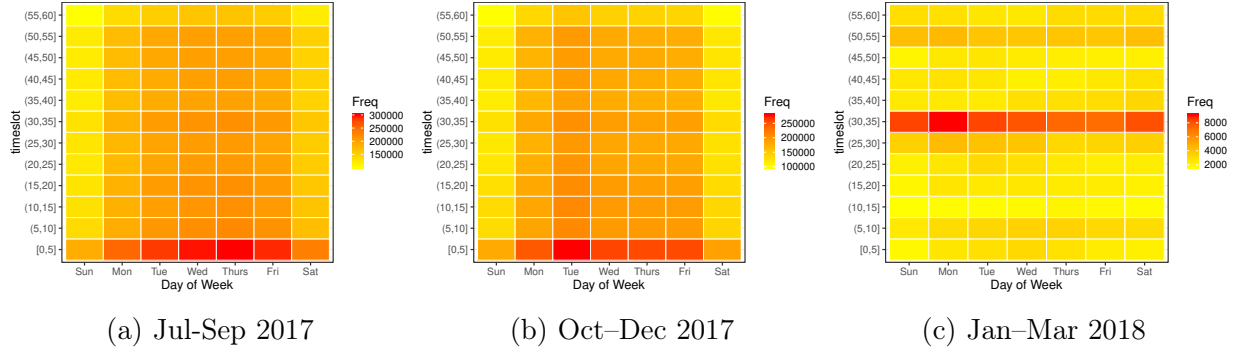


Figure 4.5: Frequency of event occurrences per timeslot (minute of hour) by day of week.

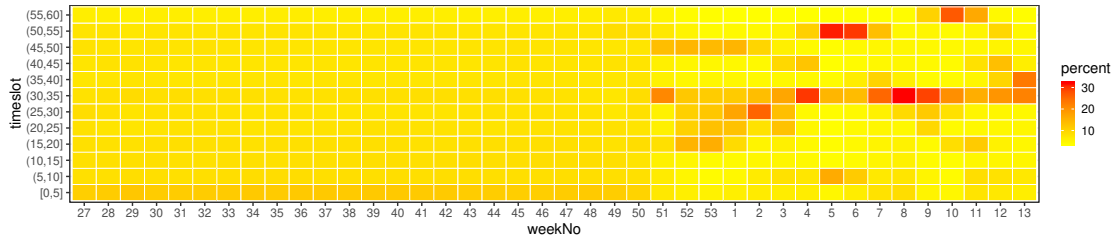


Figure 4.6: Frequency of event occurrences per calendar week by minute of hour. appears to have slightly fewer events, and around 10pm UTC midweek also experiences slightly more events.

Figure 4.5 shows a heatmap of event frequency per day of the week based on the minute of the hour. Here, for readability we have binned (discretized) minute into five-minute portions. The figure shows a shift in when events occur. Initially, the majority of events occur at the start of the hour in Jul-Dec 2017 (Figures 4.5a, 4.5b), which aligns well with AWS’s per-hour billing. The week seasonality is also visible here too. Post change, in Figure 4.5c (Jan-Mar 2018), the seasonal effect disappears and the majority of events now occur at the half hour.

We need to be a little cautious with the exact interpretation of the heat maps as we are comparing different frequency ranges: there are significantly more events per day in Jul-Dec 2017 than Jan-March 2018. To provide a little context and assist in the interpretation of these figures, Figure 4.6 presents the frequency of events as a normalized weekly percentage, i.e. each column sums to 100% and each cell represents the percentage of events that occurred in that 5 minute interval for the corresponding week. What we see is that while more events

occurred at the start of the hour as a proportion of the total weekly events this is less pronounced. Interesting is that Figure 4.6 clearly delineates exactly when AWS rolled out the new pricing mechanism in US regions. We also note that the effect of Figure 4.5c is also visible in Figure 4.6 but is less pronounced.

4.1.4 Changes in Market Dynamics

Having explored the changes in event occurrences we now explore how the spot market changes have affected market dynamics.

Spot Prices Over Time

In general, the spot market before the change was highly variable, often exhibiting a high frequency of large price changes. However, the new system seems to have some small sporadic changes but these are much less pronounced and not clearly discernible at the resolution shown in Figure 4.3.

Availability of Spot Instances

Figure 4.7 shows the availability of the `r3.4xlarge` instance type across regions and availability zones as the price increases. (Distributions for other instance types exhibit similar trends.) We observe two features that are particularly interesting. The first, which is readily observed in Figure 4.7 and well expected given AWS's own statements, is the definite upper limits of availability in the new data (which has not been cropped at all). Additionally, the relative linearity seems to indicate the artificial setting of spot prices, as well as defined upper and lower bounds. The second, which appears both before and after the change, is the appearance of multiple discernible inflection points along the availability curve. This could show one of two things: 1) AWS has imposed pressure on pricing or hard upper and lower price bounds for both the old and new pricing schemes, or 2) bids are generally made near

round pricing figures, so this would indicate the new spot pricing is still influenced by bid pricing (indirectly by way of demand at a given price).

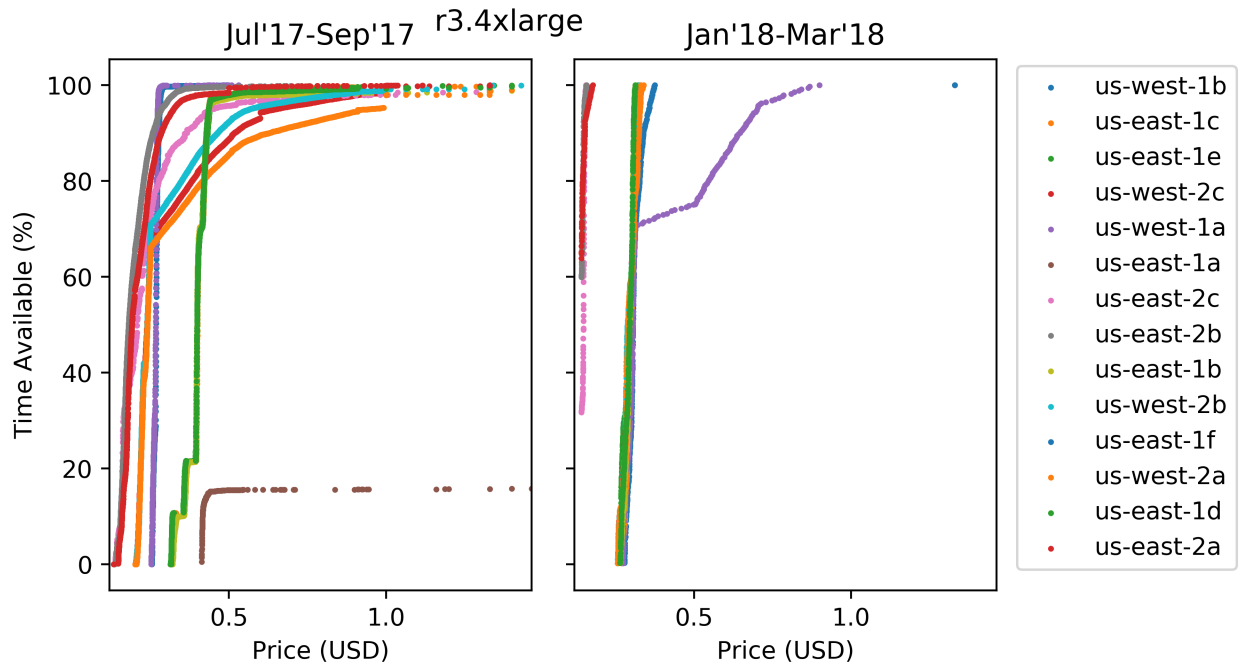


Figure 4.7: Cumulative availability of the `r3.4xlarge` instance type vs. bid price for both the old and new pricing schemes.

Spot Price Duration

Figure 4.8 shows the cumulative price duration—the time that a bid price remains above the spot price. The primary observation is the orders-of-magnitude difference between price duration under the old versus new pricing schemes. However, more interesting is the prevalence of “stair-stepping” that we observe where the duration jumps at given points, while still growing in between. This possibly indicates that AWS adjusts (both before and after the scheme change) the spot price both at regular intervals and when demand exceeds a certain threshold. We can also identify that, at least this specific instance type, `us-east-1f` has a cumulative duration curve nearly identical both pre- and post-change. Finally, we see a more logistic shape in the data, whereas the new data (at least for shorter durations)

exhibits much more linear or even exponential behavior.

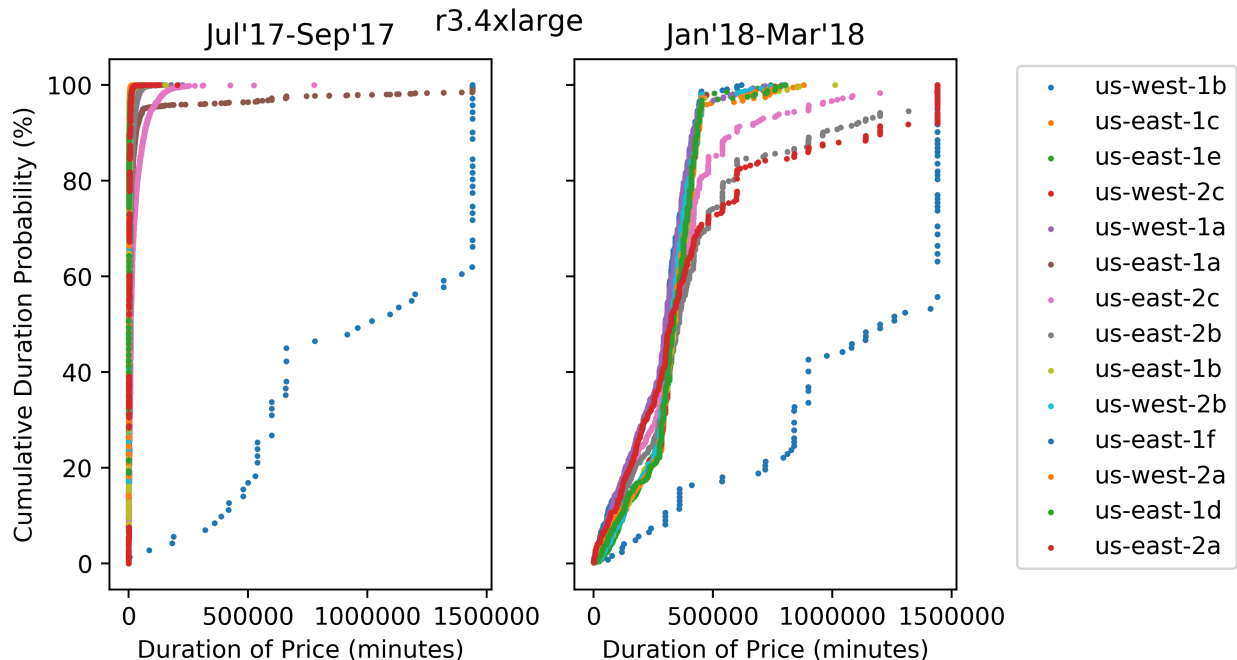


Figure 4.8: Cumulative duration of the r3.4xlarge instance type for both the old and new pricing schemes.

4.2 Practical Implications of Changes

We next investigate the implications of the spot pricing change for end users. To this end, we use historical spot pricing data to simulate bidding scenarios, and investigate the outcomes of those scenarios under the old and new pricing schemes.

4.2.1 Methodology

For each availability zone-instance type combination, we calculated the expected cost of an arbitrary workload that would take X seconds to run on that specific instance type. To do this, we measured, for each possible bid price, the distribution of durations given that specific max bid. If all occurrences of that given max bid are durable for longer than the simulated workload duration, the expected cost is simply the duration-weighted mean spot price for

the simulated duration of the workload. If no durations possess the necessary durability (i.e., if the spot instance does not stay under the simulated max bid price for the entire duration of the workload at any point in the time series), then we record this as a “did not finish,” automatically forcing a higher maximum bid and, therefore, higher expected cost.

In most cases, we obtain duration distributions where the simulated workload would finish *sometimes* but not *every time*. For example, if we consider a bid of \$0.20 and a simulated 6-hour workload. We observe that half of our sampled durations last for more than six hours and half for less than six hours (i.e., somewhere in those six hours the spot price exceeds the bid price). In this case we calculate the expected cost as two times the duration-weighted mean of the time-series within those windows. This is because, on average, if we have a 50–50 chance of instance durability, then we will have to run the workload an equivalent of two full times in order to achieve completion without a preemption.

Using these expected costs, we can then compare the incurred costs of different workload durations on different instance types between on-demand and spot instances under both the old and new pricing schemes. When simulating on-demand instances, the cost is simply the duration multiplied by the cost per time unit.

4.2.2 Assumptions

While we believe that this analysis provides necessary and accurate insights into the dynamics and cost effectiveness of spot instances and spot pricing methodologies, we do rely on three obvious, though largely unavoidable assumptions in these calculations. The first is that we use the same provisioning and termination algorithm for the post-change simulation as the pre-change simulation. While we know for a fact that the pricing algorithm has changed, the new method used by AWS is not public and, therefore, we could not easily recreate it for our own purposes. However, we do know that the new pricing is based on “long-term pressures” which indicate some consideration of the relative values of supply and

demand, and as such may still roughly approximate the old behavior though on a much more long-term and artificially smoothed scale. The second assumption we use is that these simulated workloads of arbitrary duration are not checkpointed, though one could view these simulations as the minimum required time between checkpoints rather than a workload as a whole. The third and final assumption we make is the independence of pricing and market pressures with respect to our own simulated workloads. In other words, we do not factor in the effect, albeit small, that our provisioning of a new instance would have on the overall market supply and demand.

4.2.3 Results

Using this simulation framework, we can evaluate the cost effectiveness of spot instances under new and old pricing. Further, we can use this expected cost information to highlight cases where one pricing scheme may be more beneficial to certain use cases.

First, when comparing spot instance cost effectiveness with on-demand usage, we see that, on average, the old pricing system allowed spot instances to be more cost-effective than on-demand instances for any workload shorter than about 340 hours, or roughly two weeks. However, there were availability zone-instance type combinations that would become more expensive if they had to avoid preemption for even as little as half an hour. Looking at the new pricing scheme, we observe that in all combinations, spot instances are cheaper than on-demand instances for at least one month, and in many cases indefinitely. (This is simply due to the fact that the new time-stabilized spot prices effectively never exceed the on-demand price. However, this is where our assumption of the old termination algorithm begins to fail as this shows that user-facing pricing is insufficient to predict terminations. Otherwise, in such a case where a max bid of the on-demand price would never be met, all rational users would use the spot instances instead.)

Next, comparing the expected costs of the pricing systems to each other, we see a benefit

on the side of the new pricing scheme in that, for workloads of uniformly distributed durations up to one month, it is generally much cheaper than under the old pricing system. However, the generality begins to break down when we look at the lower end of the workload duration as, for durations under an hour, the old pricing system is more cost-effective due to the presence of lower “dips” in pricing, albeit for short durations. Therefore, we can see that the end-user cost-effectiveness with respect to on-demand pricing has generally increased but, for users interested in securing spot instances for only a short time, the costs have increased as there are no longer the pricing “valleys” that we used to see within the spot market (though, conversely, there are also no pricing “peaks”).

4.2.4 Suitability for Scientific Computing

The spot market presents opportunities for acquiring low-cost resources to meet the sporadic and non-deadline-constrained needs of scientific applications. We have previously investigated the use of spot resources to cost-effectively [Chard et al. \[2015\]](#) and reliably [Wolski et al. \[2017\]](#) provision computing resources for various scientific applications, including medical imaging [Chard et al. \[2018\]](#), genomics [Madduri et al. \[2015\]](#), and workflows [Babuji et al. \[2019b\]](#). Here we discuss the implications of the market changes for such workloads.

We showed above that the new pricing algorithm generally increases spot instance prices, while decreasing price variance. While these changes may make short-duration tasks more expensive (e.g., those that can be completed within a pricing valley), for longer running jobs the new market dynamics may benefit scientific computing. Most notably, in previous work we showed that naïve use of the spot market may result in significant costs for scientific computing, upwards of an order of magnitude more than when using on-demand instances [Chard et al. \[2016b\]](#). This increase was primarily as a result of pricing “peaks” and an inability to adapt accordingly. The reduction in price variance alleviates these concerns and ultimately reduces the associated risk of using the spot market. Furthermore, more stable prices reduce

the need for elaborate provisioning tools designed to dynamically select suitable, yet low-cost instances and to exchange risk with reliability via bidding strategies.

Conversely, the fact that termination decisions are no longer associated with bid price creates new challenges. Without understanding termination dynamics, it is difficult for users to predict total cost when using spot instances or to have any certainty regarding instance durability. Under the old model, researchers could trade off risk (potential total cost) against durability by bidding well above the market price. Such strategies are no longer applicable.

The first step in corralling the computing continuum is figuring out what the cost will be. Prior to the 2017 change in AWS spot instance pricing, the AWS spot market was an ideal place to access cost-effective, highly distributed compute.

To utilize this feature, we determined that the ability to reasonably predict the spot market prices would allow users to both use compute while its cheap and determine what maximum price they should be willing to pay. We used ARIMA models along with LSTM neural networks to accomplish this, to a relatively high degree of accuracy. Following the spot market pricing change, however, the prediction task was trivially easy and the majority of spot instance terminations were no longer incurred due to price.

4.3 Spot Price Prediction

We aim to predict the spot price for a particular instance type, availability zone, and region for a specific time in the future. Specifically, we aim to predict the price using prior histories from all regions, availability zones, and instance types. Given the relative complexity of this task and the potential for many latent features to influence the spot price (e.g., seasonality, relationships between instance families, resource supply, etc.) we investigate if recurrent neural networks (RNNs) can provide better accuracy than standard statistical approaches. RNNs are becoming increasingly popular as a means of modeling time series data such as meteorological data and stock prices. For our task, we use long/short-term memory (LSTM)

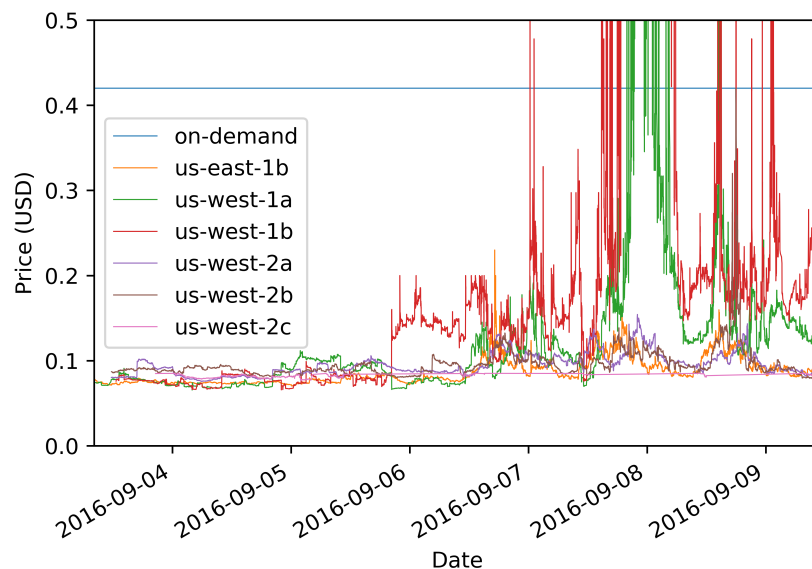


Figure 4.9: Historical AWS Spot Market price for a given instance over multiple availability zones prior to the pricing change.

as the main building blocks of the RNN. LSTMs are able to identify and remember latent features over an unspecified number of time periods, making them a versatile tool in time series prediction.

RNNs, just like other neural networks, are specified as a combination of various parameters such as number and type of layers as well as activation functions. Hence, a necessary step in building RNNs is a systematic evaluation of these parameters to find the best combination for the given task. In addition to specifying the RNN parameters, the input data for RNNs (and neural networks in general) is often pre-processed into a format that makes it easier for RNNs to generate good predictions. Hence, in the remainder of this section we describe these model specification steps: our hyperparameter selection, the resulting network structure, and the preprocessing steps considered to transform our input data.

LSTM Hyperparameter Selection

The number and type of layers is a fundamental architecture decision for a neural network. To select our network architecture for the given task, we performed a simple grid search over

a range of potential network architectures.

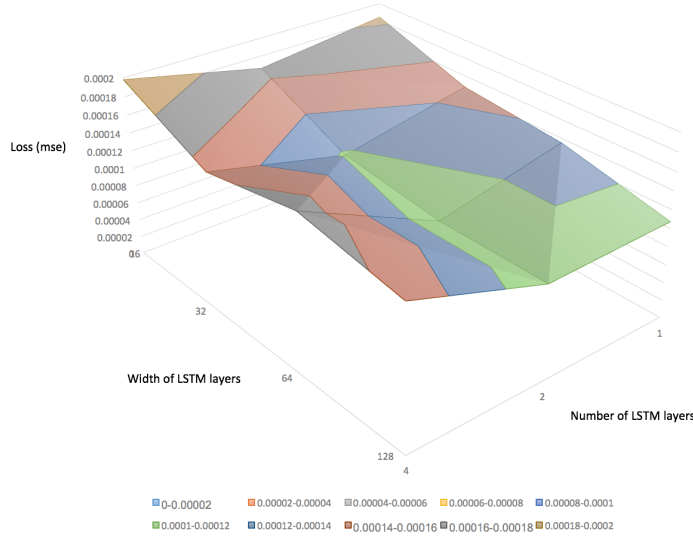


Figure 4.10: Training MSE (USD squared) after ten epochs of training each combination of LSTM width and depth.

Specifically, we systematically varied the number of LSTM layers in the network (from 1 to 4), and the width of each LSTM layer (from 16 to 128 nodes). Each neural network had an additional dense layer for consolidation after the given number of LSTM layers.

Figure 4.10 shows the loss function for these combinations of LSTM layers and widths of each LSTM layer. Based on this analysis we identified the relatively small, three-layer solution with two LSTM layers and a dense layer for consolidation to show the most promise. Note: for this analysis, we did not use a mini-batch training method such as to reduce variation between training and validation error, given the computational requirements to do so.

Network Architecture

Based on our previous hyperparameter selection, our final RNN incorporates elements from both LSTM and dense neural networks. We use a simple, three-layer network composed of two LSTM layers—each 32 units wide—and one dense node to consolidate input from the second LSTM layer to a final predicted value. We chose the LSTM unit to comprise the primary layers of our neural network model due to the temporal element of our data and regression problem, as well as the benefits of LSTM over other types of RNNs. Like

other RNNs, the LSTM units can be assembled into layers of the neural network that are able to accept data sequentially (i.e., the data is processed through each unit such that the previous data point is able to be related to the last). The advantage held by LSTM over traditional RNNs is that it possesses three internal “gates” that allow it to explicitly forward or forget data, allowing it to relate data from point to point and “forget” earlier relations over a certain period within the sequence, if these earlier relations turn out to be irrelevant. This functionality provides a mechanism to use information from surrounding inputs to affect a given input while still remaining unaffected by inputs more distant.

To train our neural network, we used the ADAM optimization algorithm [Kingma and Ba \[2014\]](#) with Nesterov momentum [Dozat \[2016\]](#) and Mean Square Error as our loss function.

Time Series Data Preprocessing

Our neural network is designed to process tick-level data (as given by the spot price history), rather than points for every time period. We choose to use tick-level data as it is less computationally intensive to train while still providing a strong pricing signal to our network. To further minimize the computational overhead of training we investigate two forms of preprocessing to our input data. One method regularizes pricing data relative to the on-demand price; the other applies exponential smoothing to regularize data for seasonal patterns.

In order to allow for greater generalizability of the model to different spot price series, we have attempted to regularize our data by transforming the raw data into a percentage of the on-demand price. This allows us to better expand one optimal solution from one instance type in one region to all instance types in all regions. While the price of a spot instance does not vary directly with on demand price (i.e., some instance types are in higher demand than others), this allows the network to work with data that are much more similar in terms of actual value. For example, imagine one instance type that is in low demand but very

expensive such that the on demand price is \$10/hr and the mean spot price is around 20% of this value (\$2/hr). Imagine another less expensive instance type in high demand such that the on demand price is \$0.50/hr and the mean spot price is 60% of this value (\$0.30/hr). It is much less computationally expensive to retrain a network from values between 0.1 and 0.3 to values between 0.5 and 0.7 than it is to retrain a network from values around 10 to values around 0.5. Additionally, in the retraining process, we are much more likely to be marginally altering the specific weighting than we are to be completely reorganizing the function of the network.

The second preprocessing step we explore looks at seasonal regularization at different levels (day, week, year) using Holt-Winters exponential smoothing. This preprocessing step represents a well-tested approach for removing regular and easily quantifiable seasonal trends so that the network itself can focus on predicting more irregular trends of the time series. This seasonal regularization also reduces the computational requirements of the network as it does not have to learn these broad trends.

4.3.1 Pricing Prediction Evaluation

We used historical Amazon spot pricing data to evaluate our neural network and to compare its performance against that of a baseline statistical model—Autoregressive Integrated Moving Average (ARIMA)—which has been shown to perform well for spot price prediction [Agmon Ben-Yehuda et al. \[2013\]](#), [Wolski et al. \[2017\]](#). We first describe our input dataset and then investigate the training performance of our network and the comparative accuracy of both ARIMA and our network on our historical, not preprocessed, spot pricing data. Finally, we explore the effect of data preprocessing on accuracy.

When reporting on forecast accuracy, we use the Mean Square Error (MSE), defined as:

$$mean \left(\sum_t (actualValue_t - predictedValue_t)^2 \right)$$

We use MSE as our primary metric for evaluation as it is the loss metric we used to train our network. We also report on the Mean Absolute Percentage Error (MAPE), Root Mean Square Error (RMSE), and Mean Absolute Error (MAE).

Dataset

In order to validate the use of neural networks for spot price prediction, we arbitrarily picked a single instance type from our historical pricing data: the **c3.2xlarge** Linux instance type from the **us-east-1b** region between September 3, 2016 and September 10, 2016 (as shown in Figure 2.1). Note: we used historical data as, at the time of this study, there was not sufficient pricing data available from the new spot market. To make the network computationally tractable, so that we could explore different network architectures, we used a subset of 10,000 points (roughly six days) for training and held back the following 2,000 points (roughly two days) for validation. While this data represents a relatively small sample, and only considers a single instance type, available data is sufficient to provide initial validation of the model.

We define our prediction task as follows: given 50 input observations, forecast the spot price for the subsequent 50 periods (i.e., predict the next 50 observations). We chose 50 observations for this analysis as it represents a window of approximately one hour.

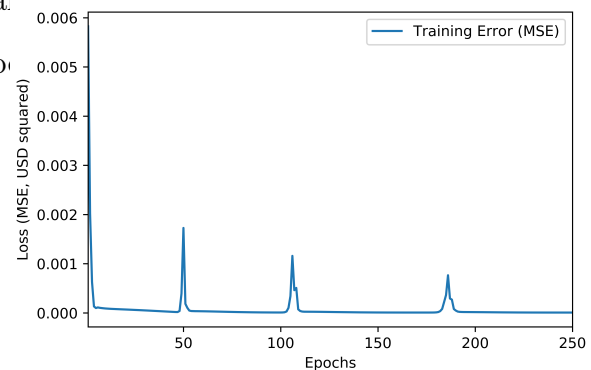


Figure 4.11: Training MSE (USD squared) over 250 epochs.

For each of these 50 input points, the ARIMA method creates and trains a new model to generate the output of the next 50 predictions; for our neural network, we trained on the 10,000 points for 250 epochs and then applied the model to sequences of 50 input points in the validation dataset.

ARIMA Baseline

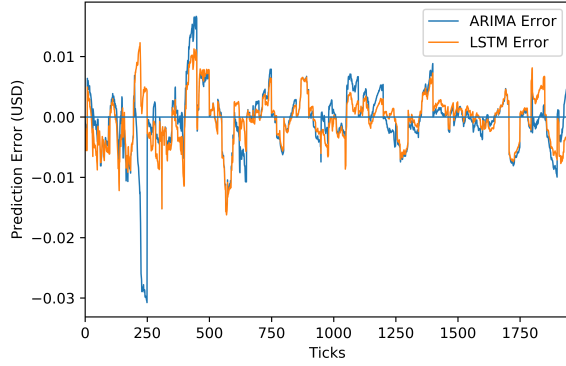
We use the ARIMA method to produce a baseline prediction. An ARIMA model for forecasting univariate time series data can be parameterized through three attributes, (p, d, q) , where p is the number of autoregressive terms, d the number of differentiation, and q the order of the moving average. Selecting appropriate values for p , d , and q depends on the particular time series. For the subsequent evaluation, we used the `auto.arima` functionality of R to automatically compare several parameterizations and select the parameter combination with the lowest forecast error. We configured the ARIMA model to use a training window of historical data (here: 50 observations) and forecast the next time periods (here: 50 predictions). In general, longer training windows and shorter forecast windows lead to a more accurate forecast.

Training and Validation of LSTM

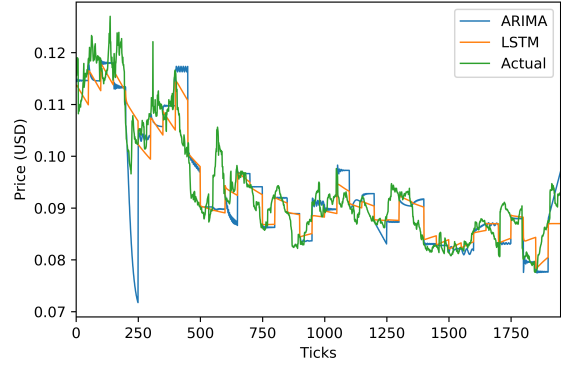
After exploring various network architectures, we trained our three-layer LSTM network on the 10,000 input data points for 250 epochs. Figure 4.11 shows the training loss function for this dataset. The disconcerting spikes are caused by the NADAM optimization function we used. Spikes are common with ADAM and are potentially exaggerated via NADAM's Nesterov (N) momentum component. We achieve an in-data training MSE of 7×10^{-6} . (A validation error of 9×10^{-6} was achieved when training on the same data split by Tensorflow with one-third of the data used for validation.) These results were achieved without preprocessing.

Forecast Accuracy

We now compare the accuracy of our neural network with the ARIMA model. Figure 4.12b shows the predictions of both models as well as the actual `c3.2xlarge` series. Both models track the actual data fairly accurately. Figure 4.12a shows the MSE for each model on each



(a) MSE of both ARIMA and LSTM models where a value of zero would represent a perfect prediction.



(b) Prediction of the 2,000 point validation set using the ARIMA and LSTM models. Actual data included for reference.

point.

While the errors look similar, the neural network is clearly more accurate than the ARIMA model. The MSE over our 2,000 point validation set was 1.7×10^{-5} for the neural network and 4.2×10^{-5} for ARIMA. Additionally, the Mean Absolute Percentage Error (MAPE) for the ARIMA and LSTM models were 3.76% and 3.35%, respectively. In Table 4.3 we report Root Mean Square Error (RMSE) and Mean Absolute Error (MAE). In both cases the LSTM model is more accurate than the ARIMA model.

Table 4.3: Root Mean Square Error and Mean Absolute Error for the ARIMA and LSTM models in cents (USD).

	ARIMA	LSTM
Root Mean Square Error (RMSE)	0.557	0.423
Mean Absolute Error (MAE)	0.362	0.320

LSTM with Preprocessing

Our preprocessing steps aim primarily to reduce the computational requirements for training; however, they might

also affect accuracy. To evaluate their effect on accuracy we retrained the neural network twice, each time using one preprocessing method. The price regularization method provided no increase (and no decrease) in accuracy. This was expected as this method was intended

to reduce training cost by allowing generalizability across many instance types. The seasonal regulation, however, did affect accuracy. In this case, using Holt-Winters exponential smoothing, we were able to achieve a training MSE of 6.98×10^{-6} —a small improvement over the same network without this preprocessing. This result indicates that the neural network is able to detect a small seasonal trend in the data.

4.4 Changes in AWS Spot Market Forecastability

We begin by presenting a summary of the spot history data (Section 4.4.1) that we use for our analyses, and describe general statistics of those data. We then briefly discuss how we prepared the data for time series modelling (Section 4.4.2).

4.4.1 Data Description

AWS continuously makes available 90 days of historical spot price data, broken down by operating system, region, availability zone, and instance type. These data are *event-level*, meaning that a price is only recorded when it changes. These fine-grained data allow us to examine the volatility of the spot market through the lens of the pricing algorithm.

We have aggregated three years of price data leading up to the November 2017 pricing-scheme change and several years of data produced thereafter. In order to focus on the transition period and to minimize irregularities due to the addition of new instance types over time, we focus our analysis on the three months preceding the change (Jul–Sep 2017: Period 1) and three months after all changes had taken place and the market had stabilized for all instance types (Jan–Mar 2018: Period 2). We also consider a third period, (Dec’21–Feb’22: Period 3), to validate that our findings apply today.

Table 4.4 shows the number of price events that correspond to a specific feature of the data. In total, we analyze 23.7 million price change events over the three three-month periods of interest.

Geography: Our data set comprises four regions in the US: *us-east-1*, *us-east-2*, *us-west-1*, and *us-west-2*. Each is further divided into a number of availability zones denoted with a letter, for a total of 14 distinct *availability zones* (Table 4.4a).

Instance Types: AWS offers many different virtual machine instance types, each with a certain processing and memory capabilities, and each typically named by a one- or two-letter

Table 4.4: Features of the dataset used in this study, in numbers of price events.

(a) Price events by zone

Zone	No. of events
us-east-1a	1,208,666
us-east-1b	2,695,610
us-east-1c	1,417,741
us-east-1d	1,300,888
us-east-1e	889,053
us-east-1f	263,531
us-east-2a	1,769,067
us-east-2b	1,015,611
us-east-2c	904,997
us-west-1a	1,749,246
us-west-1b	1,506,119
us-west-2a	2,999,299
us-west-2b	3,181,022
us-west-2c	2,788,133

(b) Price events by instance family

Grouping	Family	No. of events
General purpose	m	4,765,344
	t	36,785
Compute-optimised	c	6,997,794
	cc	141,574
	cr	27,042
Memory-optimised	r	6,800,019
	x	146,803
Storage-optimised	d	573,924
	i	1,885,286
Accelerated computing	f	197,686
	g	1,466,569
	p	665,519

(c) Price events by generation

Generation	No. of events
1	838,403
1e	1,183
2	3,951,492
3	9,186,700
4	9,726,567

(d) Price events by instance size

Size	No. of events
micro	36,785
small	49,874
medium	363,939
large	1,310,605
xlarge	3,686,178
2xlarge	6,505,655
4xlarge	5,204,867
8xlarge	5,054,800
10xlarge	519,614
16xlarge	922,910
32xlarge	49,118

family, a *generation*, and a *size* (amount of resources) [AWS](#). For example, `m5.xlarge` is a m-family (m), fifth-generation (5), extra-large (xlarge) instance type. The family indicates the general class of application for which the instance is intended; the generation is updated as new hardware replaces old; and the size the number of cores, amount of memory, etc. Across the data collected for the three time periods, 10 optimization families, 5 generations, and 11 different instance size classes were present in all three. AWS users can also select different operating systems, but we consider only Linux/UNIX instance types here as other operating systems are not well represented in the data.

Price: All prices are captured in USD charged per hour of availability. The empirical quantiles of price (USD) for the nine months of data across all instance types and geographies are (rounded to four decimal places) 0% (min) 0.0000, 25% 0.0938, 50% (median) 0.2103, 75% 0.4103, and 100% (max) 266.8800, with a mean of 0.5572 and standard deviation 4.8471. The presence of six decimal places in Amazon’s raw pricing data indicates that AWS have extremely finely grained pricing models that operate to the fraction of a cent.

Time: Price events are date-time stamped and represented in UTC to the nearest second.

In prior work [Baughman et al. \[2019\]](#), we explored the effects of the change in pricing strategy and observed substantial changes in terms of the frequency of price events, the time between events, the price change represented by an event, and the price distribution (in terms of mean and variance) between the old and new pricing schemes. Seasonal effects (in terms of the day of the week) have changed, previously there were fewer price events at weekends, and outside business hours. This has now been normalized across the week, yet still with some amount of reduced activity at weekends. We also noted that the new system offers significantly increased duration (thereby benefiting users with longer running tasks) at the expense of providing extremely low prices for short durations.

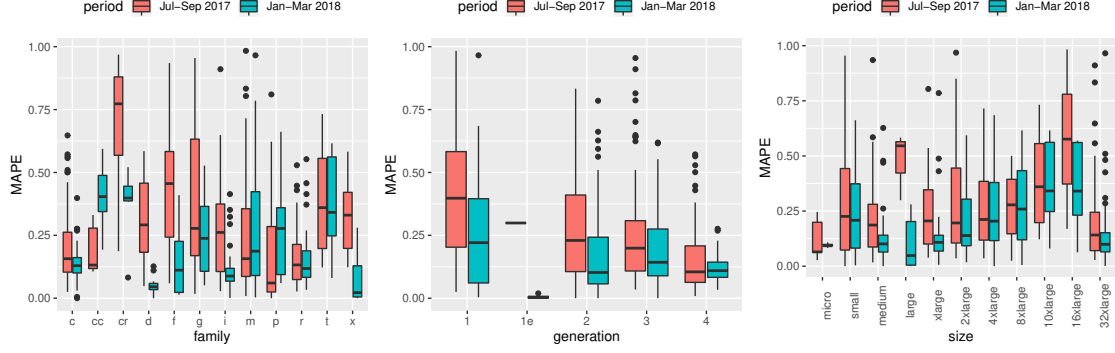
4.4.2 Data Preparation

A challenge when analyzing AWS price event data is that the time between price updates is non-uniform. The price signal is composed of ad hoc changes in price. This is unsuitable for price forecasting as price needs to be expressed over standard periods of time with a consistent time granularity (e.g., hours). Therefore, we transformed the price data for each instance type into an hourly price signal.

For Period 1, this transformation involves averaging all price events that occur within a one-hour time period, i.e., the price signal is smoothed. This is a methodological assumption that may appear problematic: it may make the forecasting of this time period easier as smoothing reduces volatility. However, we note from our prior study of this data [Baughman et al. \[2019\]](#) that while the frequency of price changes is high, the degree of change in price is low. Consequently, the smoothing effect of averaging prices within one time period should be minimal.

For Periods 2 and 3, we needed to fill in gaps between price events. In these time periods the time between price updates can be significant. In some extreme cases, this meant mapping back prior to the period to derive a starting price for a given instance. As price updates are for some instance types so infrequent we average the price for an instance across all 14 availability subzones noted in Table 4.4a. Again, this may appear as methodologically naïve, however, without doing so, we would have some instances with so few price updates that time series modelling would be pointless. In aggregating in this manner, we can engineer sufficient variance into the time series to justify modelling the series. As we apply this decision to the Period 2 and Period 3 data, we do the same to the Period 1 data for consistency. Consequently, this should make modelling easier for Period 1 (data are smoother) but harder for Periods 2 and 3 (there is more variation).

Transforming the data in this manner means we have a univariate price observation for each hour of each day in Periods 1 and 2 for 70 different AWS spot instance types and 68



(a) MAPE by Family (All Models) (b) MAPE by Generation (All Models) (c) MAPE by Size (All Models)

Figure 4.13: MAPE range for all models across all instances aggregated by family, generation, and size.

different types from Dec 2021-Feb 2022 (due to data availability). In order to fit a time series model to the data, we perform a standard holdout evaluation strategy: retaining the last two weeks of data for each period or ($2 \times 7 \times 24 = 336$ hourly price observations). Other evaluation strategies for time series models, such as cross-validation [Bergmeir et al. \[2018\]](#), are too computationally expensive for our evaluation strategy. Instead, we blind the model to the last two weeks of price data, thus the model must predict all 336 hours of price data. The rationale for a two week testing window is predicated to identify models that can forecast more than just a number of hours in advance. Therefore our aim is to find models more appropriate to realistic settings.

4.4.3 Evaluation

Table 4.5: Count of the best model type for each period.

Model	Period 1	Period 2
AR	0	0
(S)ARIMA	63	68
HW	0	0
LR	2	1
LR+AR	5	1

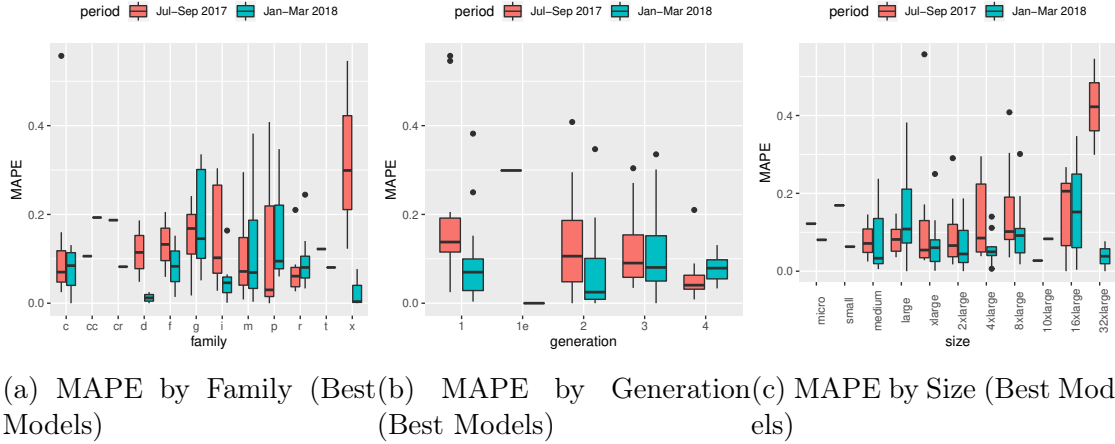


Figure 4.14: MAPE range for the best model produced for an instance type aggregated by family, generation, and size.

To assess the impact that the change in pricing scheme has had on AWS price forecasting, we fit models to every instance type that was available in both Period 1 and Period 2, then validate our results on the more recent Period 3. To retain comparability, we evaluate all models using the mean absolute percentage error (MAPE). This choice of metric is motivated by the degree of variation that spot instances can have. Other common evaluation metrics such as the root mean squared error (RMSE) and mean absolute error (MAE) are suitable when all time series are in the same value range. MAPE is scale and unit free: it is a percentage of the true value. This means that model performance of an expensive instance type, can be easily compared to one of a cheap instance type. This is not the case for RMSE and MAE as the degree of error is somewhat tied to the value range of the series itself.

To determine the “forecastability” of AWS prices, we compare the performance of the best model we were able to fit in terms of its MAPE error on the 336 hour price observations. To draw some intuition of the effect on “forecastability” we compare models produced on the same instance type before and after the change in pricing scheme (this equates to roughly 70 instances present in the data both pre- and post-change).

Specific to the discussion of model performance, it is worth noting the contextual considerations of MAPE. A MAPE of 0.2 implies that, on average over the 336 hourly prices

forecast, a model was incorrect by \$0.2 if the instance cost was \$1. Summed over all 336 hours, such a MAPE would represent a total budget estimate (if the price was US\$1 in each hour) of $\$336 \pm 67.5$. If instead the MAPE was 0.02, this would be $\$336 \pm 6.75$.

4.4.4 *Forecastability using ARIMA Models*

Available methods for automating the fitting of an (S)ARIMA model tend to overfit. Thus, we create a search space for the p, d, q, P, D, Q parameters and execute this as a parameter sweep. For each of the six parameters, we explore values in the range $[0, 4]$, for a total of 5^6 models trained and tested per instance type in the first two periods and validate on non-seasonal ARIMA models in Period 3: $5^6 \times 2 \times 70 + 5^3 \times 68 \approx 2.2$ million models. (In practice, 64% of models are actually evaluated, as some of these configurations correspond to (S)ARIMA models that cannot converge due to stationarity constraints.) By performing a sweep across all reasonable parameter values, we can be certain that we uncover all likely performance outcomes of (S)ARIMA models.

Modelling Approach and Rationale

For each AWS instance type that was available as a spot instance in Period 1 or 2, we fit each of the models noted here. Fitting so many different model types is motivated around exploring how the change in pricing strategy affected how the price of different instance types can be forecast. Intuitively, forecasting should be easier: prices are updated less frequently (meaning that price autocorrelation should be stronger), but that changes in price when a price change occurs is larger (implying that smaller lags are considered by models). Both effects would initially suggest that simple(r) should be sufficient to model and therefore forecast AWS prices.

Table 4.6: Mean instance prices (USD) aggregated by family, generation, and size to provide context of MAPE errors

(a) Average hourly price by family

Size	Period 1	Period 2	Period 3
c	0.26	0.20	0.21
cc	0.46	0.39	NA
cr	1.09	0.90	NA
d	0.70	0.80	0.78
f	25.95	3.43	2.98
g	0.55	0.96	0.71
i	0.76	0.79	0.84
m	0.24	0.24	0.19
p	1.11	3.03	3.13
r	0.29	0.35	0.34
t	0.01	0.00	0.00
x	8.13	6.17	4.67

(b) Average hourly price by generation

Size	Period 1	Period 2	Period 3
1	7.46	1.48	1.69
1e	226.14	26.69	8.01
2	0.65	1.32	1.14
3	0.34	0.48	0.38
4	0.27	0.33	0.38

(c) Average hourly price by size

Size	Period 1	Period 2	Period 3
micro	0.01	0.00	0.00
small	0.02	0.01	0.01
medium	0.01	0.02	0.01
large	0.03	0.04	0.03
xlarge	0.09	0.13	0.13
2xlarge	0.15	0.24	0.26
4xlarge	0.30	0.43	0.45
8xlarge	0.76	1.04	1.17
10xlarge	0.66	0.60	0.78
16xlarge	7.13	3.04	2.83
32xlarge	22.36	11.91	6.00

4.4.5 ARIMA Model Results

Table 4.5 shows the distribution of models that had the lowest MAPE of all models attempted for Periods 1 & 2. This suggests only that ARIMA was better able to find a low error model in most cases: it does not state categorically that ARIMA is “best.”

Figures 4.13 and 4.14 shed more light on this part of the discussion. Models fit to Period 1 (red) are compared to models fit to Period 2 (green) and aggregated according to family, generation, and size. To be able to suggest that forecasting is “easier” after the change in price structure, we would want to see a discernible reduction in MAPE errors in green box and whisker plots.

Figure 4.13 shows MAPE across all models. We see reduced MAPE in Period 2 for the **c**, **cr**, **d**, **f**, **g**, **i**, **r**, and **x** families, but not for the **cc**, **m**, **p**, and **t** families. Similarly, most generations except generation **4** seem easier to model on average. For size, as the instance increases in size, so too does the challenge to forecast its price. Note that we have trimmed wildly inaccurate models: those with a MAPE greater than 1, i.e. a model that is more than 100% incorrect on average. These assertions hold only in the sense of somewhat blindly trying an array of modelling techniques.

If we consider only the best model (usually an (S)ARIMA as noted in Table 4.5), this provides a slightly different view as shown in Figure 4.14 (note that the range of the y-axis is different between Figure 4.13 and Figure 4.14). Families **cr**, **d**, **f**, **i**, **t**, and **x** are notably “better” post change. Families **c**, **g**, and **m** have on average a lower MAPE in Period 2, but a higher variance as well. Families **c**, **cc**, **p**, and **r** are illustrating that the change in pricing scheme has negatively affected the ease of forecasting their price. For generations, generations **1**, **1e**, and **2** are notably better in Period 2. Generation **3** is similar in both Period 1 and Period 2, and generation **4** seems slightly negatively affected. For instance size, the larger instance types are no longer discernibly “more difficult” yet there does seem to be occurrences of higher variance in the MAPE for models fit to Period 2 data.

We use three metrics to compare ARIMA model performance pre- and post-change: difference in mean accuracy between periods, difference in accuracy across periods for a given parameterization, and difference in accuracy between mean and optimal parameterizations for each instance in each period.

4.4.6 ARIMA Convergence

We look at the distribution of differences between periods of each parameterization across all instance types. To do this, we compare the accuracies of a given parameterization between Periods 1 and 2. This inherently requires a value for both periods. However, likely due to extreme volatility in Period 1 and nearly no volatility in Period 2, 36% of all parameterizations fail to converge. Of these, 18.6% are due to lack of convergence in the first period and the other 81.4% are due to lack of convergence in the second period.

This model instability is likely due to the fact that AWS’s artificial pricing scheme post change abstracts away meaningful dynamics, forcing the models to learn not only the market trends, but also AWS’s pricing method. Additionally, it is difficult for ARIMA models to learn changes in time series when those changes are so few and far between. This yields a prediction task more akin to asking when a change will occur rather than when the next change is.

4.4.7 Distribution of ARIMA Accuracy

We evaluate overall performance by showing the 95% confidence interval of mean performance of all ARIMA models for each instance type in Table 4.7. Overall, the relative lack of variability in Period 2 and Period 3 makes it much easier to predict than in Period 1. The ease of prediction enables the same models to reduce the mean MAPE across all instance types by nearly 90%.

Table 4.7: Mean accuracy (MAPE) across instance types.

Value	Period 1	Period 2	Period 3
2.5%	0.14	0.05	0.01
Mean	2.55	0.90	0.61
97.5%	13.52	4.39	4.94

4.4.8 ARIMA Parameterization Accuracy

To test whether parameter selection has a significant effect on model accuracy, we evaluated the performance of our ARIMA models by comparing the mean of all the models against the best model for each instance type. Table 4.8, shows the 95% confidence intervals and mean difference between optimal and average model performance across all instance types. These values demonstrate the time series is significantly more predictable on average and show order of magnitude improvement for the worst models.

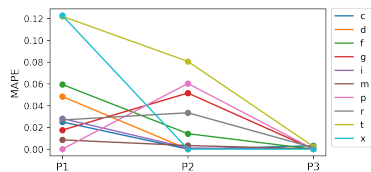
Table 4.8: MAPE difference (%) between mean and optimal ARIMA parameters over all instance types.

Value	Period 1	Period 2	Period 3
2.5%	0.08	0.03	0.01
Mean	2.41	0.81	0.60
97.5%	13.27	4.24	4.93

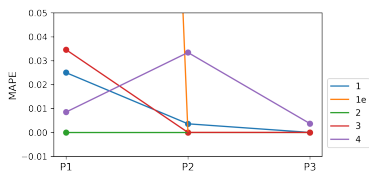
4.4.9 ARIMA Performance Trends

To demonstrate the generalization of our claims surrounding ease of predictability, we present in Figure 4.15 trends over time, broken down by best and mean model parameterizations for optimization family, compute generation, and instance size, respectively. In these figures, we show the best and mean predictive powers of ARIMA models across our entire parameter sweep for spot pricing in all three periods. Through this illustration, it is clear that instance pricing has become and remained significantly easier to predict based on any metric.

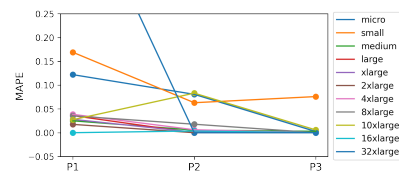
Figure 4.15: Trends in predictability over time—broken down by family, generation, and size—for average and top performing models respectively.



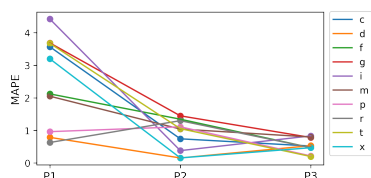
(a) Mean best model performance by family.



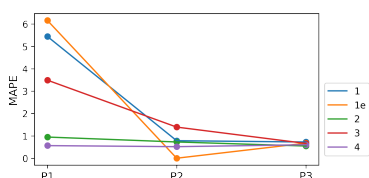
(b) Mean best model performance by generation.



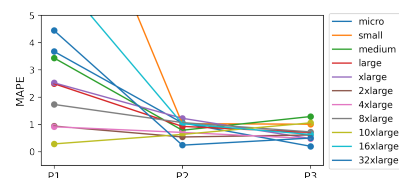
(c) Mean best model performance by instance size.



(d) Mean model performance by family.



(e) Mean model performance by generation.



(f) Mean model performance by instance size.

4.4.10 Practical Implications

Given a general-purpose instance type like `m4.16xlarge`, which has an average spot price of \$1.15 and an on-demand price of \$3.20, the expected error is only \$0.30 per instance per hour. Keeping the error rate in mind, this savings would net an end user \$1.75 per hour from the on-demand price. By comparison, the mean error of an average model pre-change would require a \$4.08 bid, making using naive ARIMA probabilistically more expensive than on-demand. With a model performing in the top 2.5% of all models, the prices drop to \$1.31 in Period 1 and \$1.17 in Period 3, respectively.

Effectively speaking, a random model post-change is more than 4 times more accurate than one pre-change. Additionally, even when considering large-scale applications, which may use an automated resource management strategy, the performance of the top models now means that at **1,000,000 core-hours**, the user will have spent **less than \$360 over optimal** or about 2% of total cost. This lack of waste means that any workload easily adapted to the cloud can utilize spot instances with high certainty of the price, making spot instances perfect for cloud-based scientific computing.

CHAPTER 5

WORKLOAD PROFILING

WITH AUTOMATED EXPERIMENTAL DESIGN

Since the price to execute a given workload on a given resource became more stable and more predictable, we turned to profiling workloads in order to characterize how they might run on resources not previously used. This work led to two primary contributions: 1) we demonstrated that small regressive models could be used to predict how a workload would change with each changing feature of the compute resource, as well as the ability to aggregate these models into a unified, multidimensional model (see Fig. 7.5); and 2) we developed an automated experimental design system that was able to learn execution characteristics to within 10% of optimal after just four executions across multiple resources (see Fig. 5.9)

5.1 Role of Data in Profiling

The ability to create, and tune, machine learning models is dependent on sufficient training data. To obtain this training data we explore two possible approaches. The first relies on exploiting information obtained from executions on external resources (e.g., a local cluster or supercomputer); the second uses an automated profiling approach to conduct experiments to measure performance on different instance types, with different configurations, and different input data.

5.1.1 Profiling Using External Histories

Scientific workflows are often developed and executed in a variety of environments before transitioning to the cloud. This prior execution information can be used to develop predictive models that can later be mapped to cloud environments using *transfer learning*. In the case of SwiftSeq we have obtained nearly one million task executions on Beagle2, a Cray XE6 supercomputer. Specifically, we have collected detailed execution traces and input metadata for variant calling workflows used to process tumor-normal pairs for over 2,100 exome sequencing samples. The input exome datasets range in size between 2 GB and 40 GB and have between 1 and 12 distinct readgroups. This dataset provides a basis to

investigate how the applications behave when processing different input data.

5.1.2 *Profiling Using Automated Experiments*

In the absence of prior executions and to gather cloud-specific execution data we also explore proactive experiments to collect execution information. To do so, we have developed a parallel profiling system that is able to provision arbitrary cloud instances and then parameterize, execute, and monitor executions on those instances. We build upon our prior work creating an automated profiling service [Chard et al. \[2016a, 2017\]](#) to develop a simplified parallel profiling system.

Our parallel profiling system is built using Parsl [Babuji et al. \[2019a\]](#), a parallel scripting library for Python. Parsl allows for the simple construction of parallel scripts that are comprised of both Python and external command line applications. Parsl manages the execution of the script on arbitrary execution environments including laptops, supercomputers, and clouds. We use Parsl to construct simple scripts to execute a given SwiftSeq application on a set of cloud instances using various input datasets and settings. Using Parsl’s AWS executor, we define the set of instance types and a preconfigured Amazon Machine Image (AMI) with all applications preinstalled. Parsl uses these settings to provision cloud instances from the spot market. When the spot request is granted, Parsl deploys an IPyParallel engine on the instance. This engine connects back to the Parsl executor and begins accepting profiling jobs. Our profiling system then dispatches jobs in the form of Python scripts to the instance. Each Python script, dynamically configures the execution environment by first downloading the input data from S3, installing the Python Resmon tool to record resource usage, configuring the execution environment, and customizing the SwiftSeq execution script. This process enables us to dynamically set execution parameters, such as the memory allocated to a specific application or the number of SwiftSeq processes to deploy. We customize these values for each instance type to reflect its capabilities and available resources.

Immediately prior to invoking the SwiftSeq workflow the profiling script installs Resmon to capture resource statistics and utilization counters for CPU, memory, disk, and network utilization. Resmon is configured to run as a daemon, collecting data every second and appending it to a local file. Once the SwiftSeq workflow completes the profiling script renames both the SwiftSeq and Resmon logs and publishes them to an S3 bucket.

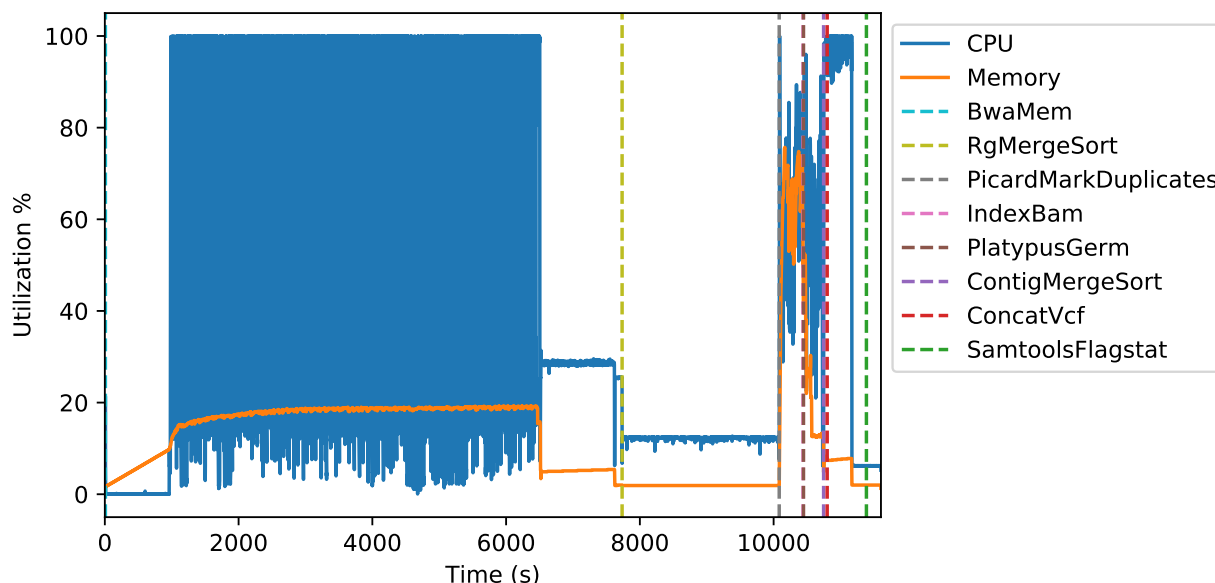


Figure 5.1: The execution profile of a Platypus genotyping workflow executed on an `m4.4xlarge` instance. The starting time of each application is denoted by a vertical dotted line.

We have used the profiling system to execute the genotyping workflows on 14 distinct AWS EC2 instance types with varying resource capabilities. We focus specifically on the M4, R4, and C5 instance families, ranging from 2 to 72 vCPUs and from 2 GB to 488 GB of memory. The CPU and Memory utilization from one execution of the Platypus genotyping workflow is shown in Figure 5.1. The figure highlights the distinct applications that comprise the workflow and exemplifies their different execution characteristics.

5.2 Profiling Application Performance

The performance of an application can depend on a wide range of features, from input data size to the application’s ability to take advantage of multiple cores. In this section we briefly review the performance of our nine genomics applications when executed on EC2 and the Beagle2 supercomputer.

5.2.1 *Application Performance on Cloud Instances*

We used the profiling system described in Section 5.1.2 to deploy and monitor the two genotyping workflows across 14 different instance types from three instance families. Each experiment was repeated at least three times. The profiling system captures both Resmon resource utilization counter values and, via the SwiftSeq execution log, the start and end time for each process executed by a workflow. As some applications are parallelized and start multiple processes, we aggregate processes by name to compute the total execution time of a given application.

Figure 5.2 shows the range of execution times for each application across the 14 instance types. Execution times range from less than five seconds (ConcatVcf) to more than nine hours (HaplotypeCaller). We see that some applications are more sensitive to instance type than others. For example, IndexBam has a maximum-to-minimum execution time ratio of 19.4 (i.e., the longest execution time was 19.4 times more than the shortest), whereas SamtoolsFlagstat’s ratio is only 1.23.

Figure 7.7 illustrates the relationship between execution time, CPU, and memory for several applications, highlighting the different instance families. We note several observations. First, while some applications are CPU and/or memory bound (e.g., BwaMem, ConcatVcf), some display constant performance irrespective of resources (e.g., RgMergeSort), and others seem to be unrelated to available resources (e.g., PlatypusGerm). Second, scaling relationships are most obvious within an instance family and performance is often proportional to

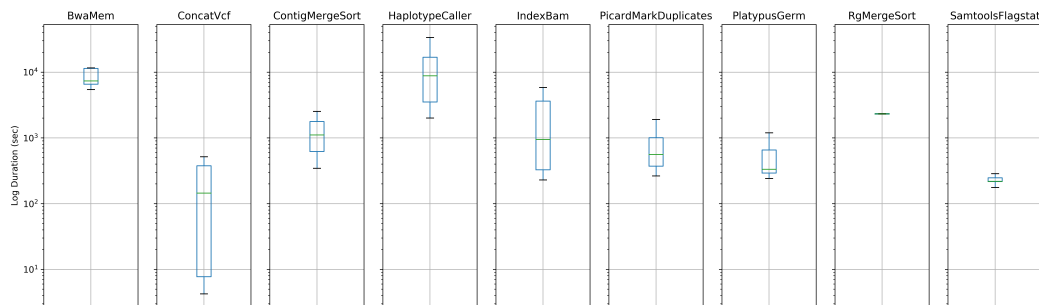


Figure 5.2: Average application performance across instance types.

the price of the instance (e.g., as shown in Figure 7.7 for the HaplotypeCaller genotyping workflow). And third, there are significant performance differences between instance families as evidenced by different execution times when using instance types with relatively similar resources.

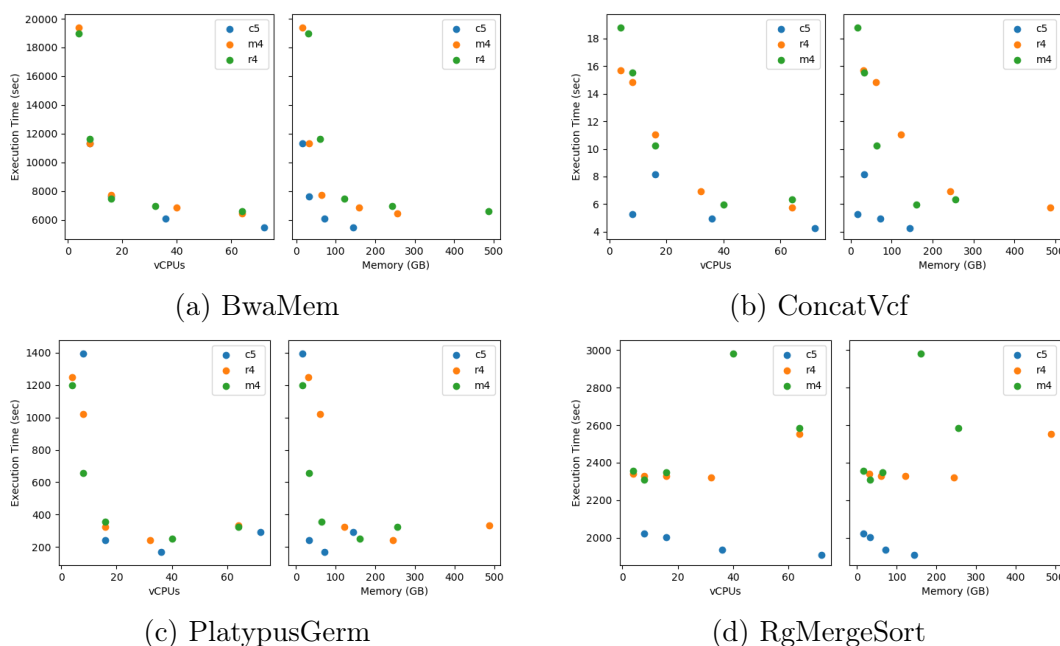


Figure 5.3: Execution time for four genomics applications with identical input data, on instance types with varying numbers of vCPUs and varying amounts of memory.

Figure 5.4 shows how the total workflow performance differs across instance types and instance families as a result of the cumulative application performance in the workflow, in this case Platypus. We see that performance typically increases within an instance family

up until a point at which additional resources do not improve performance. Note also that the workflow performs best on the compute-optimized (C5) family of instances. It is these relationships that we aim to capture in the models presented in Section 5.3.

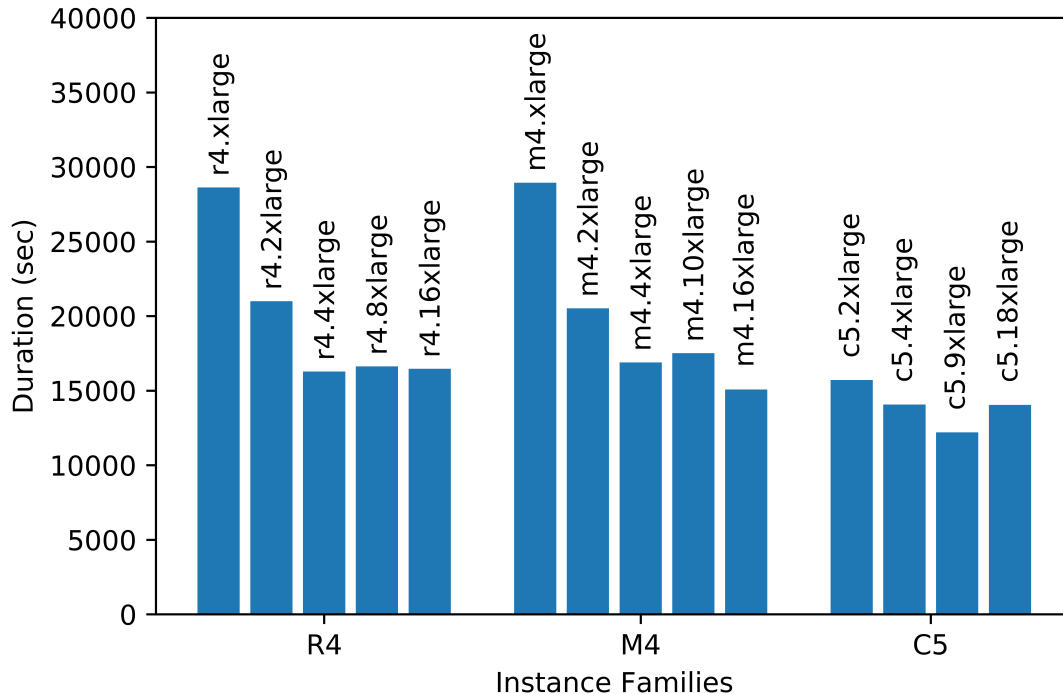


Figure 5.4: Execution time for the Platypus workflow across instance type, grouped by instance family.

5.2.2 Application Performance on HPC

We have analyzed the SwiftSeq execution logs from over 2,100 exome tumor-normal variant calling workflows, consisting of over 900,000 successfully executed tasks on homogeneous resources. We have aggregated the execution data to extract each application’s execution history and combined these with the workflow metadata to associate information regarding the input data size.

Figure 5.5 shows the relationship between execution time and input data size for several applications. These figures highlight the highly variable performance irrespective of input data size. While we expected to see some differences, the observed results differ by several

orders of magnitude in some cases. It is possible that resource contention affected performance. For example, execution of thousands of concurrent applications, many of which have not been designed for parallel execution and that access shared reference genomes or inefficiently access files stored on a shared file system is one potential reason for this contention. Furthermore, Beagle2 is a production, multi-tenant supercomputer, and therefore execution may be affected not only from contention among SwiftSeq workflows, but also from external, unknown usage.

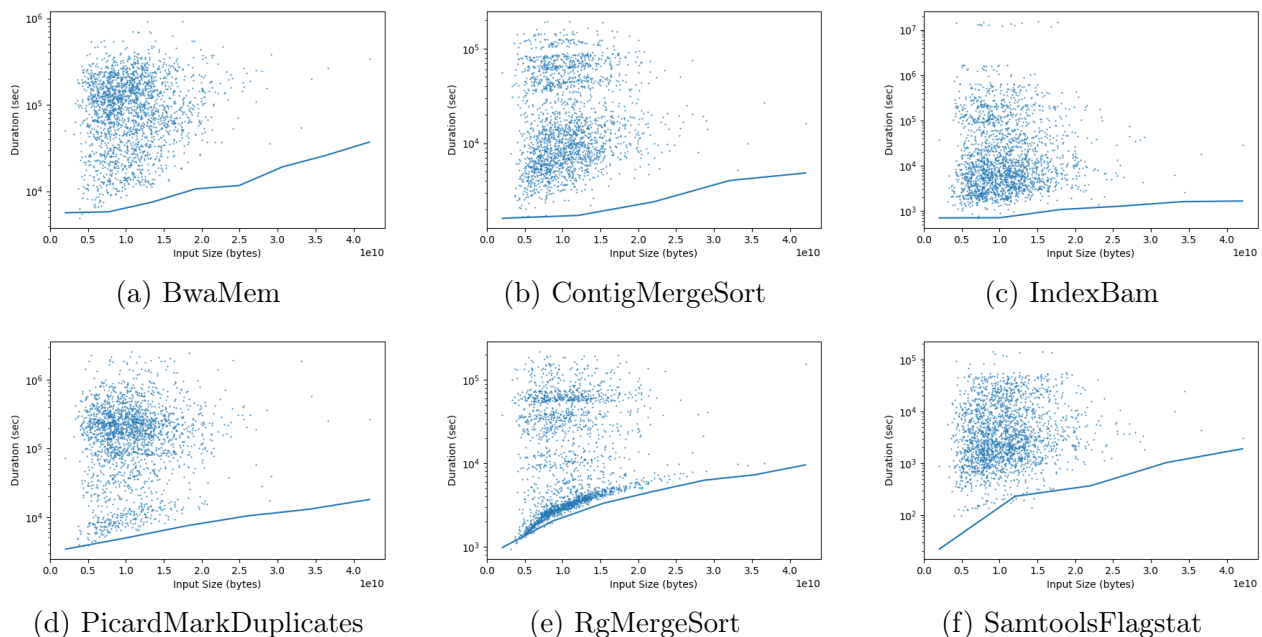


Figure 5.5: Execution time vs. input data size for six common applications executed on Beagle2. Each dot indicates an execution of the application. The line shows a fitted lower bound.

5.3 Predictive Performance Models

Based on the data collected in the previous section we now develop models to predict execution time given instance type and input data size. We present three distinct models: first, using data generated by the profiling system we create a model to predict application execution time on a given instance type; second, we use execution traces from the Beagle2

supercomputer to model the execution time of an application given different input data size; finally, we combine these two models to construct a composite model to predict execution time given arbitrary input data size and instance type. The respective model parameters are reinitialized and then fitted with unique values in every run to ensure independence between evaluations. We evaluate these models by using the deviation of the actual execution time from the predicted value from the respective regressions. We then calculate the mean absolute percent error (MAPE) for each model respectively.

5.3.1 *Instance Type Model*

We see in Figure 7.7 that some applications scale well with respect to resources, while others do not. Of those that do scale well, they tend to scale well within instance families, where the amount of memory per core is constant between instance types. Based on the observations in the previous section, we developed a regression-based model to predict execution time for each application and instance type.

To profile each of these applications, we implemented a regression using SciPy’s `curvefit` function which uses nonlinear least squares. Based on the observed nature of how these applications scale, we selected the formula below to capture and fit scaling from both available memory and vCPUs.

$$f(vCPUs, RAM) = \frac{1}{a * vCPUs + b * RAM} + c$$

Regressing this equation using parameters a , b , and c over our data, we can represent the portion and magnitude of the scaling that is dependent on vCPUs and memory, respectively. The structure of this regression also allows us to create composite functions, as discussed in Section 5.3.3. Finally, fitting only three parameters allows an initial fit with limited data and can be processed in milliseconds on a commodity laptop.

(a) Model accuracy for Platypus workflow

Application	MAPE	SD	Baseline MAPE
BwaMem	8.0	6.8	33.7
RgMergeSort	10.6	6.3	10.6
PicardMarkDuplicates	10.3	7.9	68.2
IndexBam	34.7	21.3	57.5
PlatypusGerm	33.3	26.2	76.0
ContigMergeSort	13.5	16.1	56.5
ConcatVcf	30.5	34.9	51.2
SamtoolsFlagStat	8.8	3.2	9.4

(b) Model accuracy for HaplotypeCaller workflow

Application	MAPE	SD	Baseline MAPE
BwaMem	12.0	16.1	40.8
RgMergeSort	8.4	6.6	8.4
PicardMarkDuplicates	11.8	9.9	64.9
IndexBam	44.0	50.0	142.4
ContigMergeSort	26.1	21.9	31.0
ConcatVcf	8.2	5.8	10.4
SamtoolsFlagStat	9.8	8.5	14.5
HaplotypeCaller	5.7	4.8	144.1

Evaluation of our model

on the two genotyping workflows, containing nine individual applications, we were able to achieve a fit with a mean absolute percent error (MAPE) of 17.2%. Individual accuracy for each of the applications is reported in Tables 5.1a and 5.1b for the Platypus and

HaplotypeCaller workflows, respectively. In these tables, we also provide a baseline MAPE that is calculated in reference to the application’s mean performance. We use this as a baseline as it functions as a random selection analogue given that the expected performance of a randomly se-

lected instance will be the mean performance. An example of how our model fits the BwaMem application is shown in Figure 5.6. Most applications achieve a mean validation error of less than 20% and those that do not achieve such accuracy exhibit the highest dependence on instance family qualities, like compute optimization and memory per core.

Such a fit shows that the execution of these workflows on AWS is both dependable and predictable and that our model is suitable for providing accurate performance predictions across a diverse range of applications and instance types.

5.3.2 *Scaling Prediction*

Given the varied performance measurements from execution on Beagle2 we choose to model execution behavior based on only the highest performance executions. This is appropriate as the highest performance successful executions can be assumed to be those with the least contention and thus the most representative of an execution performed on a dedicated resource. We thus fitted a lower bound to capture the “best case” runs (as shown in Figure 5.5).

We use a similar regression model to that defined above, but with an additional variable exponent, l , in the denominator because larger input data sizes typically take longer to run (i.e., this is a growth model) and different applications scale with different computational complexities:

$$\begin{aligned} g(size) &= \frac{k}{size^l} + m \\ &= k \times size^{-l} + m \end{aligned}$$

To fit the lower bound, we binned the data and retained the smallest values within each bin. The exact number of bins and values retained were calculated dynamically on an application by application basis to maximize accuracy. To avoid overfitting, a condition was added that we needed to retain at least twice the number of observations as we have parameters in our model. This quantity was selected as it forced binning optimization away from selecting increasingly larger bins while being low enough to allow the data binning process to still include data from each bin, thereby ensuring a level of representativeness. Due to the much larger quantity of data and the increased complexity of the regression

equation, the mean fitting time for this model was 0.032 seconds running on a commodity laptop.

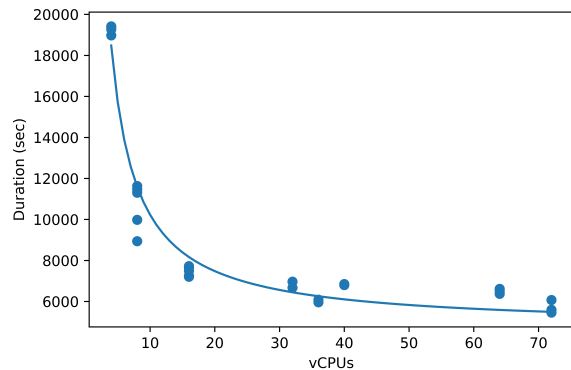


Figure 5.6: The fit for the BwaMem application given different resource capabilities.

Using this model, we were able to achieve relatively good lower bound fits. Table 5.2 shows the accuracy for each of the applications. Note: we include only applications executed before the variant callers as these workflows used alternative variant calling algorithms. In each case, the mean absolute percent error is below 15% and in two cases (RgMergeSort and ContigMergeSort) is below 5%. We believe this represents an adequate lower bound fit and, as such, allows for fair representation of optimal runtime.

5.3.3 Composite Performance Models

We developed the two models f and g defined above by using manual profiling on cloud instances and historical data from Beagle2, respectively. We now seek to combine these models so that we can use them to predict performance on arbitrary instance types given arbitrary input data sizes.

To do so, we apply a transfer learning approach in which we first define a composite function such that the input for one model is the output from another and then transfer the data size model to the cloud by training on a single observation (i.e., instance type). While this approach requires the addition of an internal regularization constant (within the exterior function to allow us to fit the models at the point of known overlap), the composite function

allows us to easily relate scaling factors resonating from both of the respective models:

$$G(size, vCPUs, RAM) = f(vCPUs, RAM) \times g(size) \times r$$

To create this three-dimensional model, we retain the crucial scaling parameters from both models and solve for r , which is our new regularization factor. This is possible as we used a single 10 GB input file to train our function $f()$. The new regularization factor can be trained using only one additional runtime observation. We selected one instance at a time from our validation set and solved for r using that observation while keeping all other parameters constant.

We validated our model using the BwaMem application and found a mean MAPE of 38% and a minimum of 22.9%. Our combined model performs best for small and compute optimized instance types, with only 16xlarge or larger instance types having a MAPE greater than 32%. This shows that we can successfully use the data type model (trained on executions in a non-cloud environment) by adding a single observation from cloud execution to construct a multi-dimensional model with useful accuracy. An example of the combined model for the BwaMem application is shown in Figure 7.5.

Table 5.2: Model accuracy for lower bound fit

Application	MAPE	SD
BwaMem	5.4	5.1
RgMergeSort	1.4	1.0
PicardMarkDuplicates	8.0	5.4
IndexBam	14.5	8.7
ContigMergeSort	4.3	4.0
SamtoolsFlagStat	12.6	10.0

5.4 Automated Experimental Design

Our profiling approaches to this point assumes significant knowledge of application performance. However, gaining this knowledge can be expensive in the case of an unknown

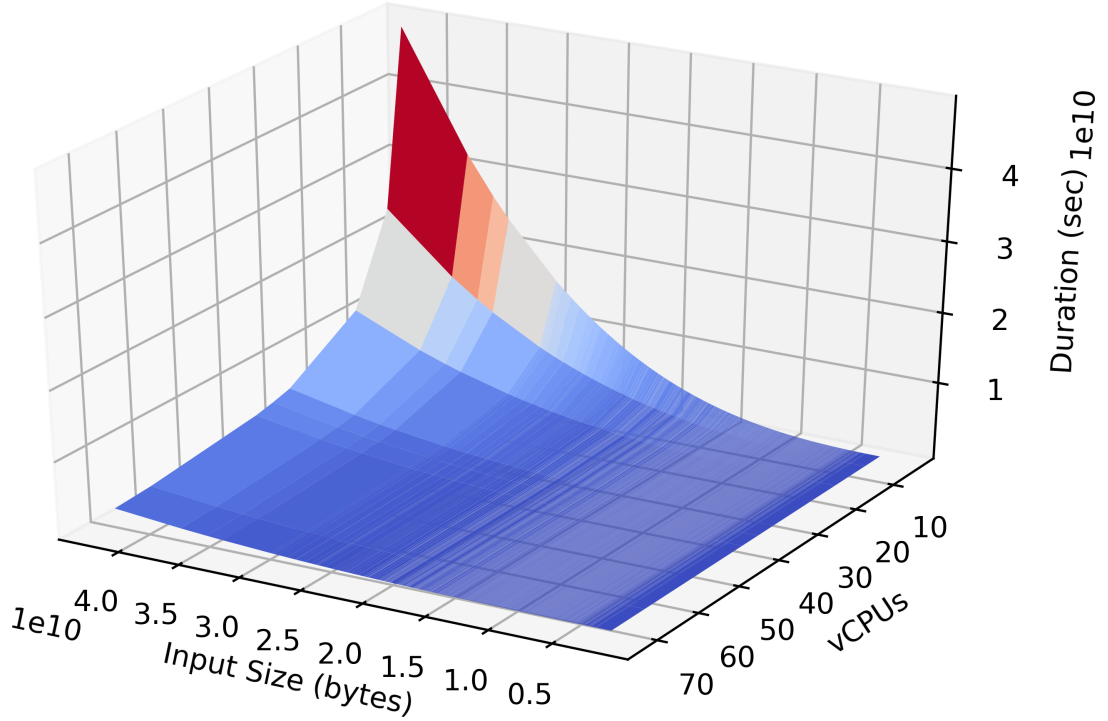


Figure 5.7: Combining both the predictive models.

application. Ideally, we would like to obtain just a few measurements for a new application, for example by considering only some representative instance types. Such information could then be used to guide additional profiling, or potentially as part of an online scheduling tool that could explore the search space by selectively assigning tasks to instance types and learning from execution performance. To minimize the number of instance types required to generate an application profile we explore an active experiment design approach [Settles \[2012\]](#).

In order to intelligently select the “best” instance types (i.e., those that would provide the highest marginal reduction in error across the model search space), we have implemented a simple approach that calculates where the search space has the least knowledge—where *density* of observations is the least. By selecting computational resources strategically to increase our knowledge of the solution space, we can then achieve a better model fit not only

for that space with little prior information but also for the rest of the model as this new observation can act as a new “anchor” to help validate and improve the model.

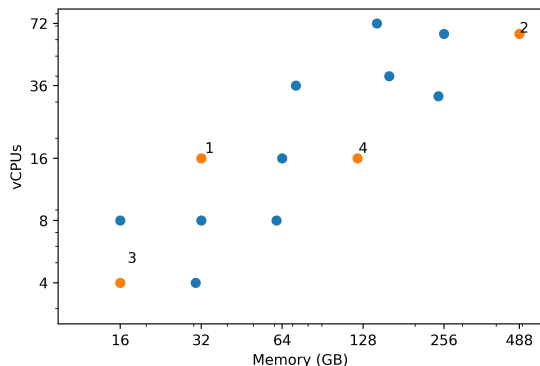


Figure 5.8: An example of the experiment selection algorithm at work, showing the first four instance types selected as it explores the search space.

Figure 5.8 shows how instance types are selected for a new application. The algorithm is simple—it first selects the most central instance type and then progressively selects instance types in the most *remote* part of the search space (calculated using a distance measure). More specifically, given a set of existing data points (given by prior history), we will then launch the application on an instance type that is the most remote in terms of instance CPU and memory. As there are few (relatively speaking)

instance types, we can compute relative remoteness by calculating the distance between each instance type for which we have data and for which we do not. Though this is inefficient, the small scale of this problem makes it relatively inconsequential. Other methods such as dynamic programming and prior state retention could be explored to improve performance.

We have applied our experiment design approach to model each of the applications in the Haplotype genotyping workflow. We assume no prior knowledge of the applications and therefore begin by selecting an instance type from the center of the search space, `c5.4xlarge`.

Table 5.3: Difference in MAPE for models trained with varying numbers of observations from the MAPE obtained by the model trained with all observations.

Observations	Difference
3	79.3
4	29.9
5	19.8
6	2.3
7	1.9
8	0.9
9	0.6
10	0.1
11	1.9
12	1.8
13	1.6
14	0.0

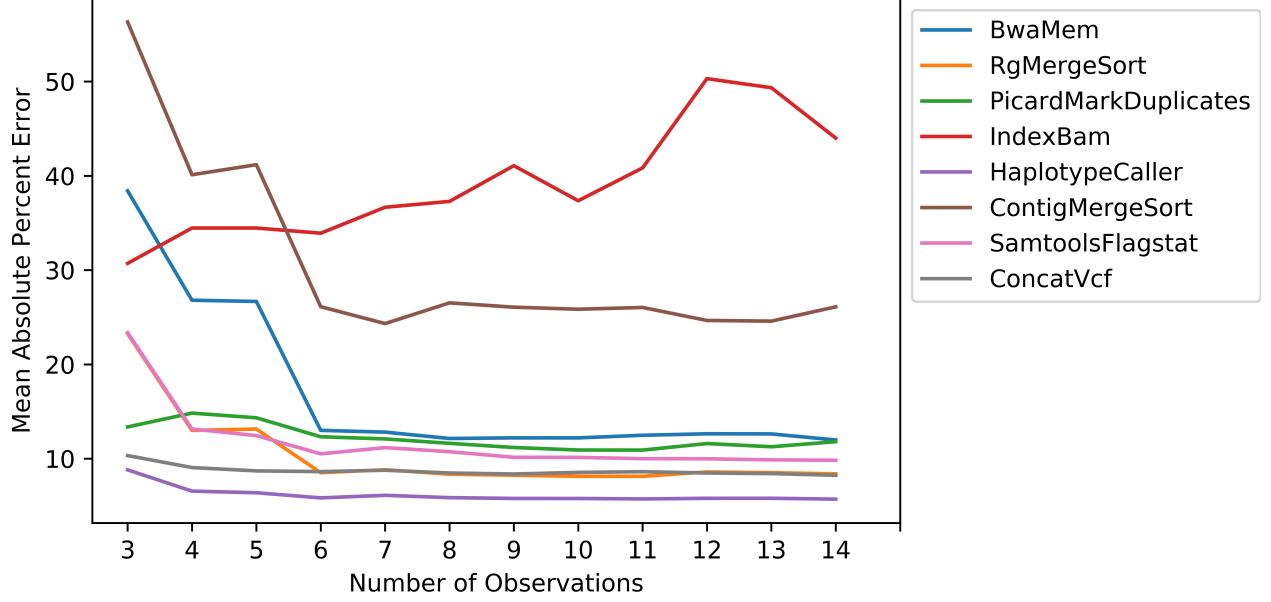


Figure 5.9: Validation of the active experiment design approach.

As depicted in Figure 5.8, we select the second observation as a `r4.16xlarge` instance type. Subsequent observations then select the most distant instance type before gradually filling in the search space.

Figure 5.9 shows the MAPE achieved for each application as the number of observations increases and Table 5.3 provides the average values across task types. We see that accuracy generally increases with additional observations, except in the case of `IndexBam`, where accuracy declines as more observations are made. This is due to the tool being tightly CPU bound; it performs roughly 25% better on the compute-optimized C5 family of instances than on any other family. Thus, the first instance type selected, `c5.4xlarge`, is in fact an accurate representation of performance on the C5 family. Subsequent observations on other instance types only increase the prediction error. As the experiment selection approach encourages the diverse selection of instance families, the scaling properties of the model are not efficiently found between instance families. We aim to address this problem in later work by extending our experiment selection techniques to explore scaling both within and across instance families to further optimize the selection of observation points.

When we have three observations on which to train our model, we achieve a validation

MAPE that deviates by 79.3% from our final MAPE value once experiments are conducted on all 14 instance types. The table also shows that our approach allows accurate prediction to within 30% of the final MAPE on average with only four observations and within 2.3% after six observations. We conclude that there is little benefit to be gained by conducting more than seven experiments.

CHAPTER 6

DISTRIBUTED EXECUTION OF LAMBDAS

WITH TRADEOFF ANALYSIS

6.1 Challenges and Problem Formulation

We discuss challenges associated with scheduling functions for execution across diverse and distributed resources, formalize the problem, describe how we model function execution time, and present assumptions we make.

6.1.1 *Challenges*

Choosing where to execute a function in a heterogeneous and distributed system depends on myriad factors ranging from the performance of a particular function on a particular resource through to the time to provision its execution environment and transfer input data to that environment. We describe the factors we consider here.

Function Runtime

Different applications perform differently on different resources, as illustrated in Figure 6.1, which shows the performance achieved on seven resource types for three applications: a parallel counter (ParCount), a matrix multiplication (MatMul), and an I/O heavy application (FileIO).

Data Transfer

A large contributor to execution duration is the time required to move task-execution information to a remote endpoint. This involves not only the latency of communicating with the endpoint, but also the cost of transferring data in cases where the function needs to act upon large input data. Data transfer rates can vary enormously, depending on endpoint capabilities and available network bandwidth.

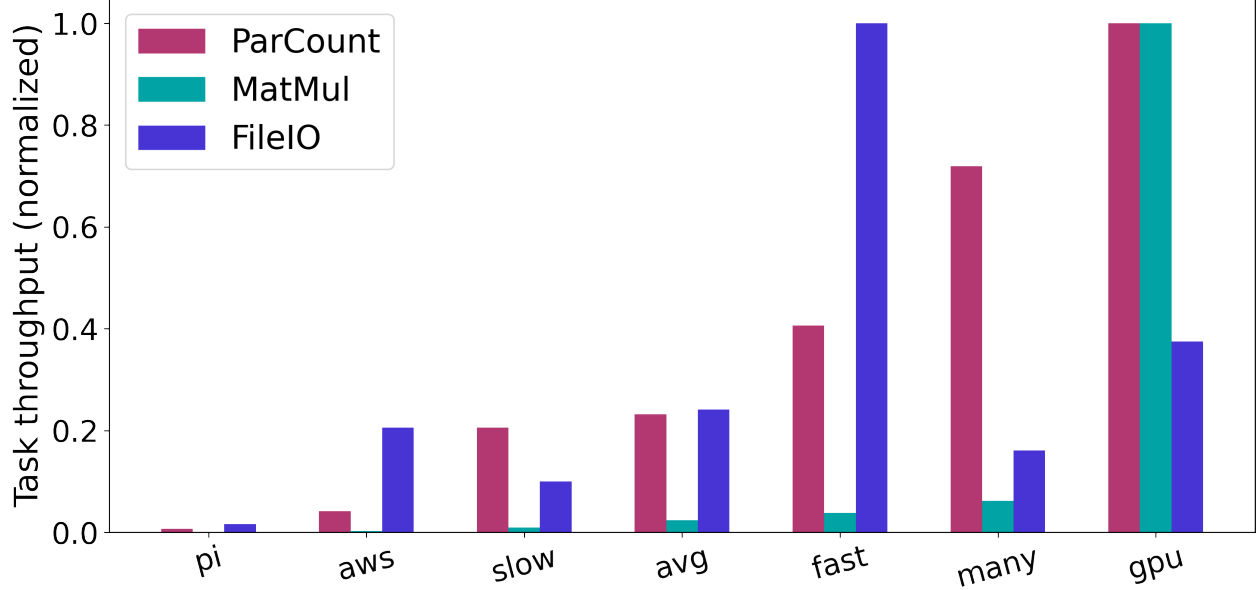


Figure 6.1: Relative throughputs (tasks/sec) for three tasks on the resource types of Table 6.1, with values normalized to the maximum observed on any resource.

Cold Starts

The *cold start problem* is a well-known and well-documented issue in FaaS literature [Oakes et al. \[2018\]](#), [Chard et al. \[2020b\]](#), [Wang et al. \[2018\]](#), and represents an important scheduling difference between serverless computing and conventional HPC. Broadly, the problem is that the first invocation of a function typically incurs the time to provision and configure infrastructure, which may include time to request resources, configure virtual machines, deploy containers, and load function dependencies. Contemporary FaaS systems have taken varied approaches to this problem, from keeping “pre-warmed” containers [Amazon Lambda](#), [Apache OpenWhisk](#), [Wang et al. \[2018\]](#) to applying optimized container technologies [Oakes et al. \[2018\]](#). While prior work has focused primarily on cloud environments, we focus on the more general case where resources include both cloud and HPC resources. Acquisition of nodes on HPC resources is typically the most significant (and variable) factor.

6.1.2 Problem Formulation

We define an *endpoint* as a FaaS-enabled computing resource where users can execute functions: for example, a cloud instance, supercomputer, desktop, or edge device.

We have a heterogeneous set of n such endpoints, $\{\text{EP}_1, \dots, \text{EP}_n\}$. We receive a stream of *tasks* of the form (f, x, data) where f is a function, data is a list of files (on the requesting machine or elsewhere) needed by the computation, and x is function input (e.g., paths to data). Neither the timing nor nature of these tasks is known ahead of time. We must decide where to send each task when it is received.

Objective

We aim to select an endpoint for each task as it is received in order to minimize overall *time-to-completion*. (Other objectives that could be considered include minimizing data movement, maximizing resource utilization).

Modeling the Continuum

We decompose the time taken to execute a task $f(x, \text{data})$ on an endpoint EP into:

- *Scheduling overhead* (t_{sched}): Time to make a scheduling decision and to queue the task for execution.
- *Cold-start latency* ($t_{\text{cold}}(\text{EP}, f)$): Time to allocate a node, start a container, and load package dependencies for f , if the endpoint is not already warm.
- *Pending-tasks delay* ($t_{\text{prev}}(\text{EP})$): Time for all previously scheduled tasks on EP to finish. (As noted below, we assume FIFO execution on an endpoint.)
- *Function transfer time* ($t_{\text{trans}}(\text{EP}, f, x)$): Time to transfer f and x from submit side to EP .

- *Data transfer time* ($t_{\text{trans}}(\text{EP}, \text{data})$): Time to transfer each input file in **data** from its source to EP.
- *Runtime* ($t_{\text{run}}(\text{EP}, \text{f}, \text{x}, \text{data})$): Time for EP to run **f** on input **x** and **data**, producing output **res**.
- *Result transfer time* ($t_{\text{trans}}(\text{EP}, \text{res})$): Time to transfer **res** from EP to submit site.

Thus, we determine the endpoint EP_i that minimizes a function's expected execution time (EET) as follows:

$$\underset{1 \leq i \leq n}{\text{argmin}} \left(\begin{array}{l} t_{\text{sched}} + t_{\text{cold}}(\text{EP}_i, \text{f}) + t_{\text{prev}}(\text{EP}_i) + \\ t_{\text{trans}}(\text{EP}_i, \text{data}) + t_{\text{trans}}(\text{EP}_i, \text{f}, \text{x}) + \\ t_{\text{run}}(\text{EP}_i, \text{f}, \text{x}, \text{data}) + t_{\text{trans}}(\text{EP}_i, \text{res}) \end{array} \right) \quad (6.1)$$

Assumptions

We make the following assumptions to restrict the scope of the problem. While unrealistic, they allow us to explore aspects of the general case.

- (A1) $t_{\text{run}}(\text{EP}, \text{f}, \text{x}, \text{data})$ is constant for a particular EP, **f**, **x**, and **data**.
- (A2) The time to schedule a task, t_{sched} , is a constant.
- (A3) The time to communicate a function to, and results from, different endpoints, $t_{\text{trans}}(\text{EP}_i, \text{f}, \text{x}) + t_{\text{trans}}(\text{EP}_i, \text{res})$, is constant for all EP, **f**, **x**, **res**. (These times were small relative to other costs in the studies reported here.)
- (A4) Each endpoint runs one task at a time, in FIFO order.
- (A5) Each task runs on a single endpoint (i.e., no task can be distributed over multiple endpoints).
- (A6) There are no inter-task execution dependencies managed by Delta (these can occur freely from within a function).

(A7) We seek only to minimize task time-to-completion.

6.2 DELTA Design and Implementation

DELTA (Distributed Execution of Lambdas with Tradeoff Analysis) is a framework that provides function profiling, prediction, and scheduling over existing FaaS frameworks (see Fig. 6.2). Users can use DELTA to manage task execution across one or more function-serving endpoints. To do so, they use the DELTA Python client to invoke tasks directly from within a programming environment.

DELTA follows a standard FaaS model in which users register a function before invocation, providing self-contained function code, a clearly stated function signature, and any dependencies. DELTA wraps the function and registers it in turn with the associated FaaS platform. When a DELTA client requests that a task $f(x, \text{data})$ (i.e., a function plus arguments and data) be executed, DELTA selects an endpoint, manages data transfers to that endpoint, waits for transfers to complete, and then forwards the request to the FaaS service to execute the task on that endpoint. Users can subsequently retrieve results asynchronously via Delta.

6.2.1 Core Components

The DELTA service maintains various components to enable task scheduling and performance prediction, task tracking, data transfers, and resource monitoring. We describe these components in the following subsections.

Performance Predictors and Task Scheduling

DELTA includes methods for training predictive models on historical data to estimate the various aspects of function execution time listed in Section 6.1.2. DELTA’s `TaskScheduler`

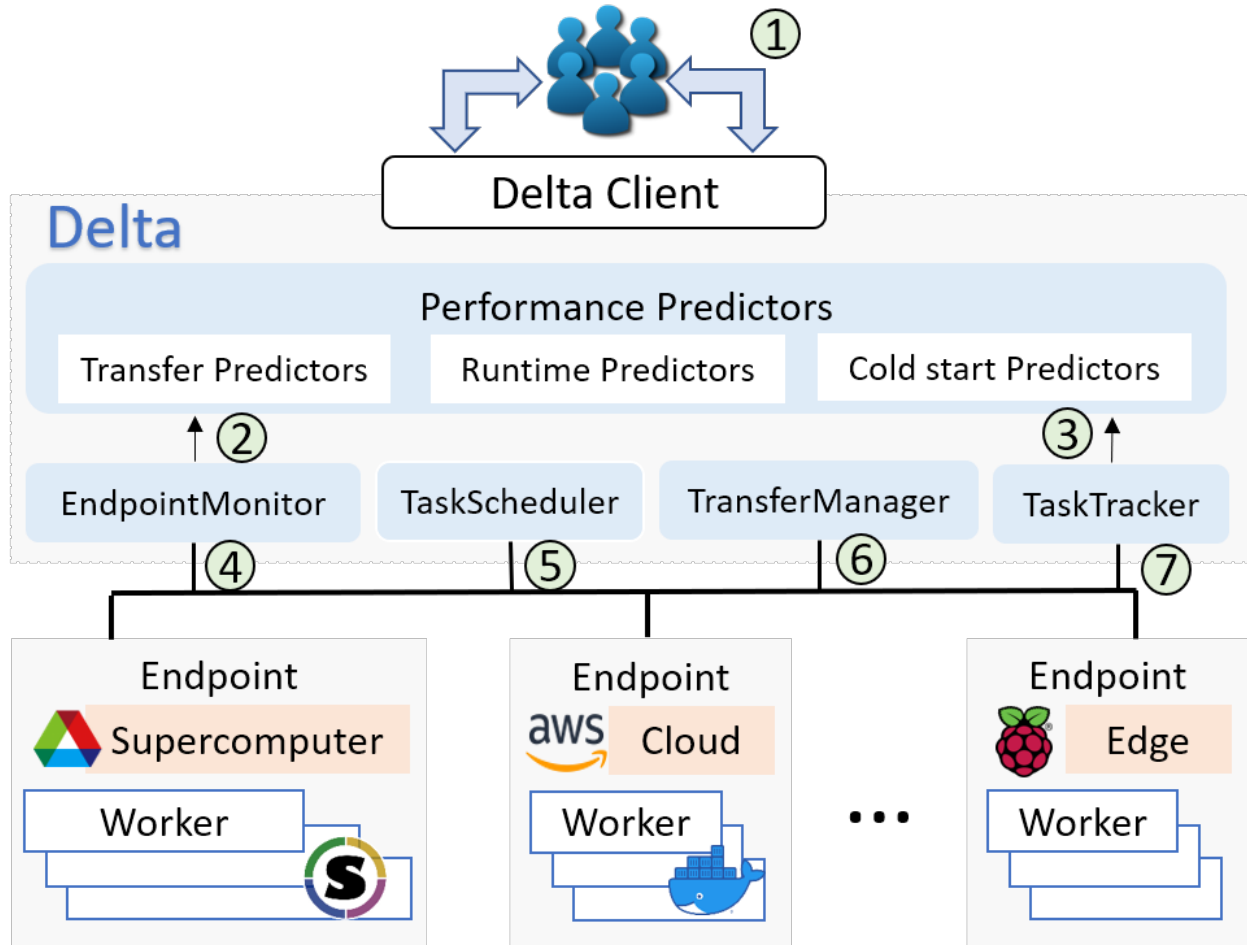


Figure 6.2: DELTA architecture, showing components of the scheduling service and the communications required to execute tasks. (1) Submit tasks and fetch results; (2) and (3) supply endpoint and task information to predictors; (4) regularly provide endpoint status; (5) distribute tasks across endpoints; (6) schedule and track data transfers; (7) regularly monitor each task’s status.

uses trained predictors to determine the most appropriate endpoint for a task’s execution. When given an incoming task $t = (f, x, \text{data})$, it applies a scheduling strategy to determine which endpoint should be used, retrieving predictions from predictors as needed. The **TaskScheduler** is extensible, allowing for arbitrary predictors and scheduling strategies.

Task Tracking

We assume that function execution endpoints execute tasks in FIFO order. An essential metric for execution time prediction is an estimate of when all tasks scheduled on each end-

point will finish running—the endpoint’s *pending time*. For this, the **TaskTracker** maintains a FIFO queue of scheduled tasks for each endpoint that it updates whenever a new task is sent to the endpoint.

Data Transfer

The **TransferManager** is responsible for managing data transfers between endpoints. If input data is not available on the selected endpoint, the **TransferManager** starts data transfer(s) between the source(s) and the destination endpoint, tracks transfer status, and dispatches the task to the endpoint only when it is notified that the transfer has completed.

Resource Monitoring

The **EndpointMonitor** component tracks the status of each execution endpoint. It relies on a heartbeat message from the endpoint to the Delta service, which provide information about the endpoint’s state, available capacity, queued workload, and utilization of its resources.

6.2.2 External Services

Delta leverages funcX [Chard et al. \[2020b\]](#) for function execution and Globus [Chard et al. \[2014b\]](#) for data transfers.

The **funcX** federated and distributed FaaS platform allows users to register any computing resource as an *endpoint* on which they can then execute functions via the funcX service. funcX is an attractive choice for an execution fabric upon which Delta can be implemented. All aspects of function execution happen via the **FuncXClient**. Functions are registered by the **FuncXClient** and assigned a *function-id* by the funcX service. Functions can then be executed, by specifying the function-id, endpoint-id, and any input parameters through the **FuncXClient**. The resulting task is assigned a unique *task-id* via which users can monitor task status and retrieve the results asynchronously.

A funcX endpoint uses Parsl [Babuji et al. \[2019a\]](#) to provision and manage compute resources. Each endpoint hosts a *manager* for receiving incoming tasks from the funcX service. The endpoint manages a collection of *workers* deployed on the various nodes available for execution. For example, in an HPC setting, nodes are provisioned via the HPC scheduler, while in a cloud setting nodes are provisioned via the cloud API. The manager routes tasks to registered workers for execution. When the function completes, serialized results are returned to the funcX service and stored in a task database, awaiting a pull from the client.

Globus [Allen et al. \[2012\]](#) provides managed high-performance data transfers between remote endpoints. It implements a third-party transfer model via which a user, or in this case the Delta service, can request a data transfer between pairs of data transfer endpoints. (More than 30,000 are active, and others can easily be deployed.) The cloud service offers a REST API for programmatic management of transfers.

Globus uses the GridFTP protocol for high-performance data transfer, and implements advanced features including parallel data streams for performance, automated integrity validation via checksums, automated restarts in the case of errors and file corruption, and a comprehensive security model that enables authentication at both source and destination as well as encryption of the data channel.

6.2.3 *Function Execution*

When a user registers a function, DELTA wraps it with Delta-specific code to allow DELTA to track task execution. DELTA then registers the augmented function with funcX.

When making a task request, the user need not specify an endpoint for execution, as DELTA determines this before sending the request to funcX. The user may optionally provide data specifications and the source Globus endpoint(s), which are included as part of the function execution request, and data will be transferred prior to function execution by the

DELTA service. The DELTA service tracks pending tasks via a background `TaskWatchdog` thread which regularly polls the `funcX` service to inquire about the status of each pending task. This polling enables DELTA to receive results as soon as they are ready and allows for more accurate runtime predictions.

DELTA provides for resiliency by generating backup tasks when the `EndpointMonitor` detects that an endpoint has failed, or if a task takes significantly longer than expected. A backup task is always sent to an endpoint other than those chosen previously for a task. Multiple backups can be sent, up to a configurable `max_backups` limit. DELTA does not send backup tasks during the exploration phase, as runtime predictions are unlikely to be accurate.

6.2.4 *Endpoint Monitor Implementation*

The DELTA `EndpointMonitor` is responsible for determining the current state of an endpoint and reports information such as task queue depth, worker status, and resource utilization. This information allows DELTA’s predictors to estimate wait time for specific endpoints. For example, the `EndpointMonitor` can track if warm nodes are available by checking if the `funcX` endpoint has active managers and workers. If no active managers are reported, the `EndpointMonitor` reports the endpoint as *cold*. This fact is used for subsequent scheduling decisions. If a cold endpoint is chosen for execution, Delta marks it as *warming*, and once it regains an active manager, it is marked as *warm*. The `EndpointMonitor` also keeps track of which packages have been imported on the endpoint worker’s environment. The required packages for a function are determined at the time of function registration by the DELTA service.

6.2.5 *Predictors*

To validate the DELTA architecture we implemented several proof-of-principle predictors.

A **runtime predictor** uses timings $(f, x_i, \text{data}_i, t_{\text{run}}(\text{EP}, f, x_i, \text{data}_i))$ from previous runs of f on EP to estimate $t_{\text{run}}(\text{EP}, f, x, \text{data})$. As function runtime often depends heavily on its inputs, we implement for the studies reported here an *input size predictor* that uses online polynomial regression to fit observed $\{\text{input-size}, \text{runtime}\}$ pairs for each function. To reduce computational cost, we group endpoints with similar execution capabilities (e.g., CPU architecture, memory and disk capabilities) and create runtime models for these groups. We rely on our previous works [Baughman et al. \[2018\]](#), [Chard et al. \[2017\]](#) to provide the necessary predictive abilities as this task is difficult and crucial.

Transfer predictors estimate the data transfer time between two locations. As with runtime predictors, we group endpoints by the features that affect transfer times, namely physical location, network connectivity, system memory, and network hardware. We can then measure for each *transfer group* how transfer rates scale with data size when explicit data scaling information is available through the function declaration.

Figure 6.3 shows that transfer rates vary greatly with source and destination. Transfers between two desktop computers on a local network are much faster than transfers from a local network to the AWS instance; on the other hand, transfers between a commercial and research cloud are incredibly fast.

We apply simple regression models to predict the relation between size and transfer rate for each pair of source and destination transfer groups. Our model predictions are shown in the dotted lines in Figure 6.3. We see that we can predict transfer times in our testbed reasonably well.

Cold start predictors estimate one-off startup costs. As mentioned in Section 6.1.1, we treat the cold start process as a sequence of three consecutive steps: node acquisition, container instantiation, and package loading.

Node acquisition times are frequently both long and hard to predict, particularly on HPC clusters where queue times can depend on many factors—capacity, resources requested, and

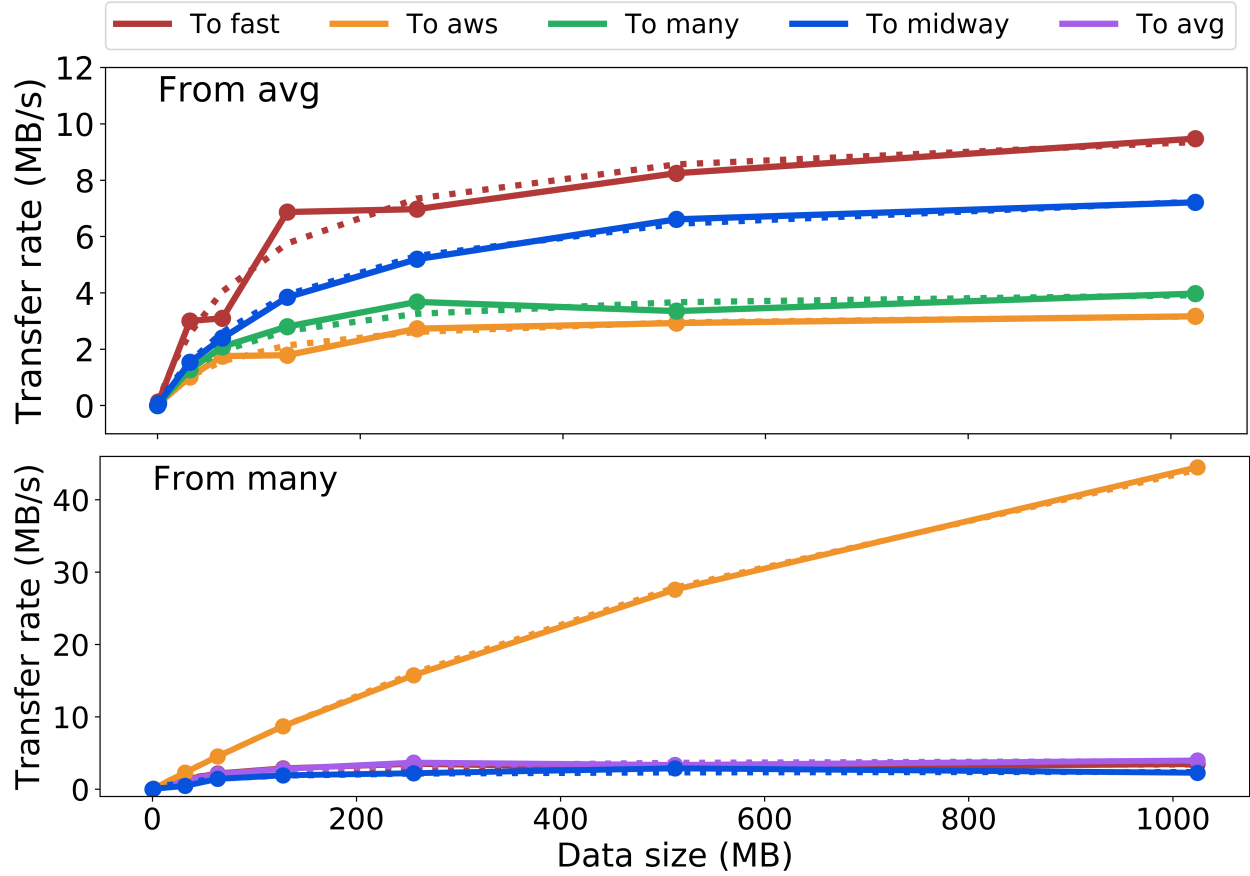


Figure 6.3: Data-transfer rates between Globus endpoints in our testbed (see Section 6.3.1): specifically, from endpoints **avg** (a desktop on the departmental network) and **many** (at Argonne National Laboratory) to those endpoints and also **fast** (another department desktop), **aws** (an AWS instance in Virginia), and **midway** (a cluster in UChicago’s Research Computing Center). The circles represent medians of measured values and our model’s predictions are shown by the dotted lines.

job-scheduling policies—and differ by several orders of magnitude [Smith et al. \[2004b\]](#). One solution is to maintain a collection of warm nodes at all times. For our work, we assume that our endpoints either need not wait for node allocation (e.g., they are dedicated edge devices or desktop computers) or, if in HPC clusters, they are already running on warm nodes.

Dependency loading involves downloading, installing, and importing packages. Figure 6.4 shows the times to import common Python packages on the different devices in our testbed. These are easily predictable quantities, so accounting for these package-loading latencies is a matter of calculating the tradeoff between matching installed packages with application

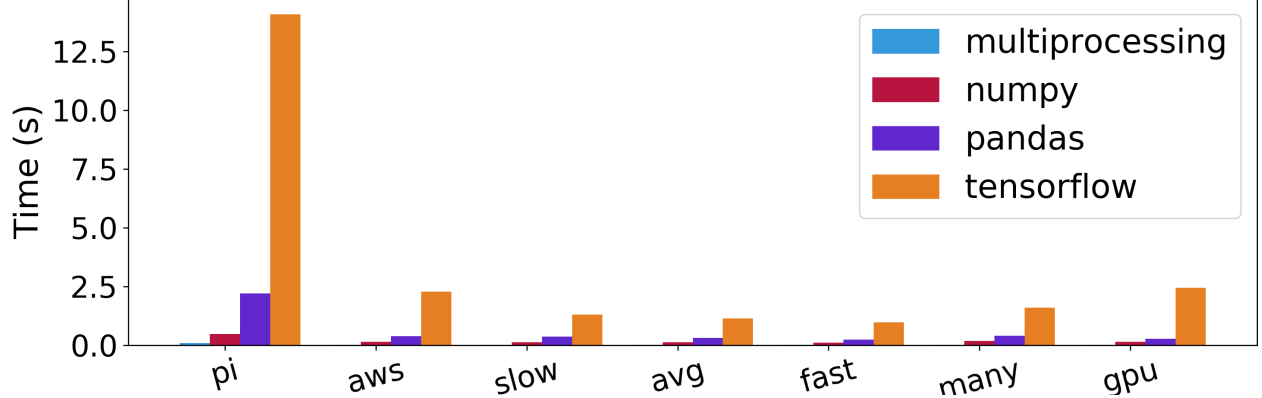


Figure 6.4: Import latencies for four common Python packages on different devices. Download and installation latencies vary similarly.

dependencies and incurring package loading overheads where a dependency is missing.

6.2.6 Transfer Manager Implementation

A user specifies, when registering a funcX endpoint, its associated Globus endpoint. When a new task is submitted to the DELTA service with a list of files required for execution, the `TransferManager` uses the Globus API (with delegated Globus Auth tokens for the requesting user) to orchestrate the transfer. The `TransferManager` monitors the transfer and dispatches the task to the endpoint when the transfer has completed. This just-in-time submission allows the endpoint to execute other tasks while waiting for transfer overheads.

6.2.7 Scheduling

We evaluate three scheduling algorithms in this paper:

- **round-robin** routes each task to the next endpoint in a list of endpoints.
- **fastest-endpoint** routes each task to the endpoint for which our runtime predictor yields the shortest estimated function runtime, $t_{\text{run}}(\text{EP}, \text{f}, \text{x}, \text{data})$.
- **smallest-EET** routes each task to an endpoint selected according to Equation 6.1.

Resource selection in a heterogeneous environment requires knowledge of how different tasks will perform on different machines. If such knowledge is not initially available, then exploratory runs must be performed: a classic explore-exploit tradeoff [Maes et al. \[1996\]](#). We have explored such tradeoffs elsewhere [Wu et al. \[2019\]](#). Here, we implement a first exploration phase in **fastest-endpoint** and **smallest-EET** in which one task is sent to each of the seven endpoint groups to collect performance data, after which tasks are dispatched based on predictions from models trained with those data. enabled automatic detection of errant task execution requests for some cases in which an endpoint should be blocked. An early version of this capability is performed by parsing returned content for common error messages (e.g., a Python `MemoryError` or a `ModuleNotFoundError`). Exceptions like these lead to DELTA’s automatic function-wise blacklisting of endpoints.

6.3 DELTA Evaluation

We use **micro experiments** to evaluate to what extent the various components (e.g., ability to learn function runtime and data transfer time) work for holistic computing ecosystems, and **macro experiments** to evaluate to what extent , as a whole, yields satisfactory performance when put under load with realistic workloads.

6.3.1 Experimental Setting

We performed experiments on a heterogeneous testbed with three Raspberry Pis, four nodes in the Chameleon research cloud (two CPU-optimized, two GPU-optimized), an AWS EC2 instance, and three desktop computers: see Table 6.1.

For both micro and macro experiments, we used four benchmark applications, each with a parameter n (and for MatMul, also a parameter m) that can be set in an experiment: 1) **Counter**, which uses a single thread to increment an integer value n times; 2) **ParCount** (parallel counter), a happily parallel computation that runs c instances of Counter, one on

each of a resource’s c cores, each counting to n/c ; 3) **MatMul**, which uses TensorFlow to perform m multiplications of different $n \times n$ matrices; and 4) **FileIO**, which writes and reads n bytes to/from disk.

Table 6.1: The heterogeneous testbed used for experiments included 11 devices.

Description	Hardware	Name	Count
Edge device (Raspberry Pi 3B)	ARM Cortex, 4-core, 1GB	pi	3
Slow desktop (UChicago CS)	Intel Core, 8-core, 8GB	slow	1
Average desktop (UChicago CS)	Intel Core, 8-core, 8GB	avg	1
Fast desktop (UChicago CS)	Intel Core, 12-core, 16GB	fast	1
Cloud (AWS EC2 T3a.medium)	AMD EPYC, 2-core, 4GB	aws	1
Manycore CPU (Chameleon)	Intel Xeon, 48-core, 125GB	many	2
GPU node (Chameleon)	Intel Xeon, 48-core, 187GB	gpu	2

6.3.2 Micro Experiments

We first present three experiments to highlight different features of and verify that can transparently route tasks to different endpoints across the computing continuum.

The first experiment demonstrates that, when asked to run a workload that favors a particular endpoint, quickly learns to send this workload to this endpoint. The second experiment demonstrates that, when tasks involve data transfers, accounts for the cost of data movement, thus making smarter decisions than baseline strategies. The third experiment demonstrates that, when faced with endpoint failures and slowdowns, quickly recovers and completes tasks.

Learning Under a Small Load

A simple situation in which to observe the benefits of heterogeneity is when tasks arrive infrequently enough that the only factor affecting EET is the function’s runtime, t_{run} . Figure 6.5 shows the execution times observed when 100 MatMul tasks, each with $m = 50$ and

$n = 1000$, arrive at 0.5 Hz: an approximation to an application that regularly requests a neural-network inference.

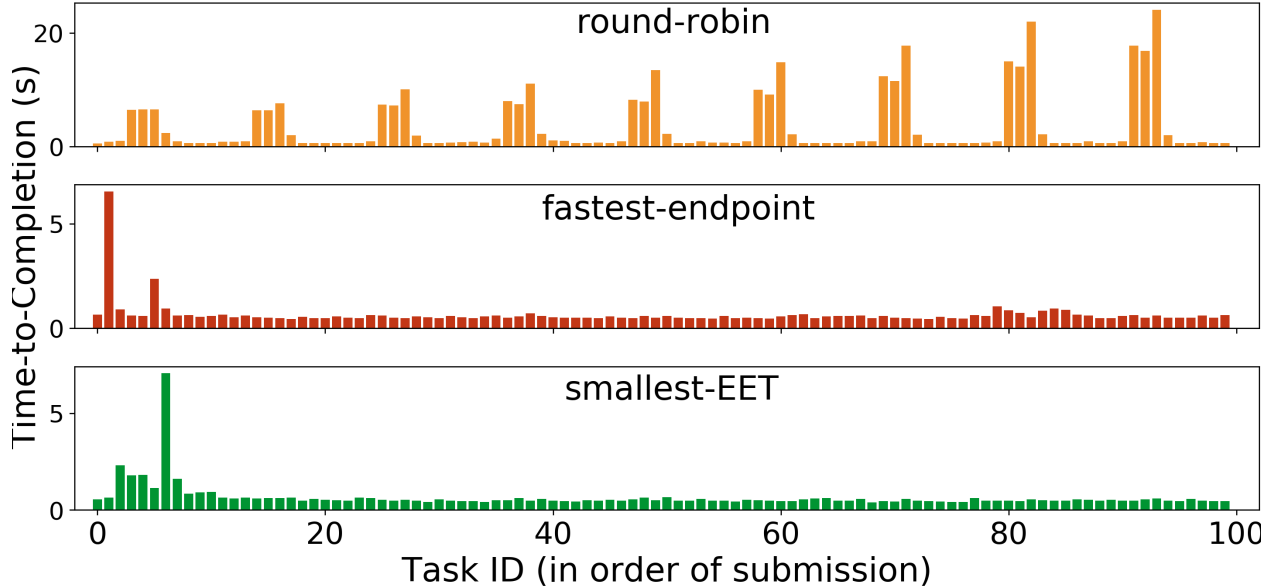


Figure 6.5: Learning function performance behavior with different strategies under a small load. Note different scale for `round-robin`.

Both the `fastest-endpoint` and `smallest-EET` strategies use the first few tasks to explore the different endpoint groups. Both strategies then favor the most suitable endpoints for MatMul tasks, which unsurprisingly are the GPU endpoints. The naive `round-robin` strategy performs noticeably worse, and as the slowest endpoints cannot keep up with the arrival rate of tasks, `round-robin` execution times become increasingly slow. While the `fastest-endpoint` strategy performs optimally in this simple case, we see below that it does not in others.

Data Transfer Trade-offs

To evaluate the effects of data transfers, we conducted an experiment involving a series of Counter tasks, each with n selected randomly from $\{2^{24}, 2^{26}, 2^{28}\}$, and with each also specified to require two 1KB files located on the `avg` endpoint. As does not cache files, both files must be transferred once for each task that is run on an endpoint other than `avg`; as this

requires a request to the cloud-hosted Globus service, there is a non-trivial cost. Thus, for every execution, the service must consider both execution and transfer times to predict the lowest overall EET. We did not use the three `pi` endpoints in this experiment, as they lacked Globus support; thus, we had eight endpoints and six endpoint groups. After 18 exploration tasks (one per size for each endpoint group) used by to learn function performance behavior, we requested a total of 100 tasks at 0.5 Hz.

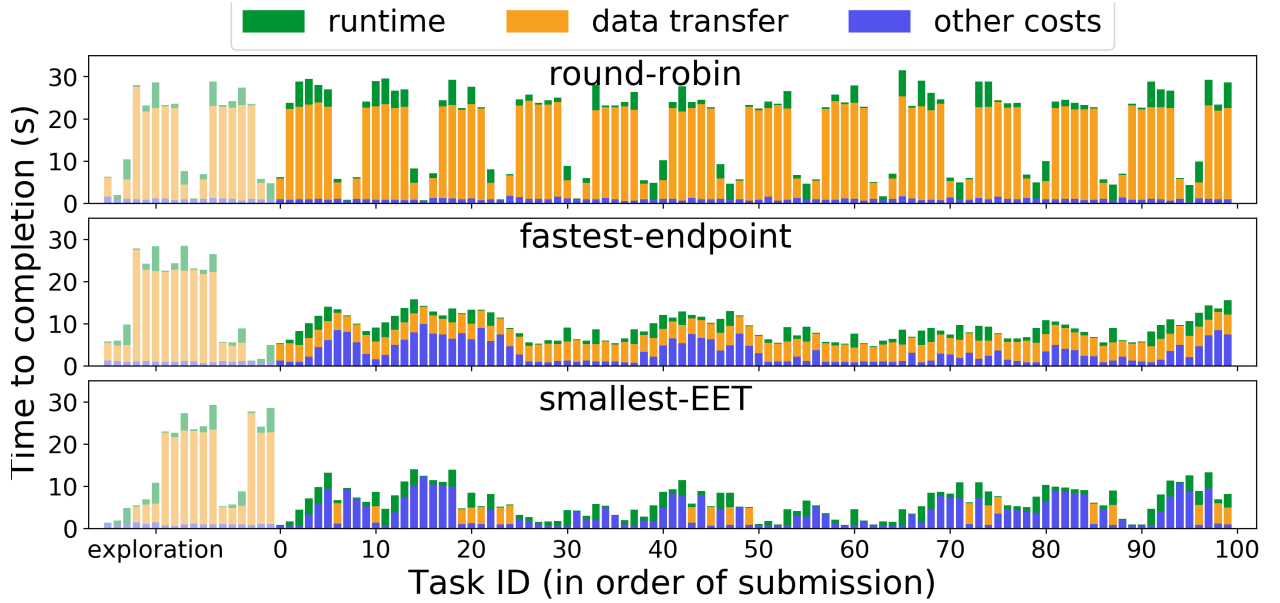


Figure 6.6: Accounting for data transfer cost as part of function execution. The workload is 100 Counter tasks; the figure distinguishes between runtime costs, data transfer costs, and all other costs, with the lattermost including time spent waiting for resources.

Figure 6.6 shows that the `round-robin` strategy naively cycles through the available endpoints, incurring a data transfer cost for every endpoint except the one that holds the required files. The `fastest-endpoint` strategy also ignores transfer costs and thus simply chooses the endpoint that has historically run the task the fastest. Since the files were located on the `avg` endpoint, which has slower CPU cores than three of the other endpoints, the `fastest-endpoint` strategy always chooses to offload the computation to a different endpoint, incurring non-negligible transfer costs. Finally, the `smallest-EET` strategy, which considers both transfer costs and runtime costs, only offloads the computation to a different

endpoint (incurring transfer costs) some of the time. Closer analysis of the figure reveals the following pattern: the `avg` endpoint was chosen for execution several times, as indicated by the lack of transfer cost, until enough pending tasks accumulated (shown by the growing size of the blue bars, which include wait time), at which point it was best to offload the next task to a different endpoint and incur transfer costs.

Tolerating Failures and Slowdowns

Here a stream of MatMul tasks with $m = 50$ and $n = 2500$ arrives at 0.33 Hz. To simulate a failure, we removed the `gpu-1` endpoint after the 30th task and restarted it after the 60th task.

Figure 6.7 shows how after a first exploratory phase, execution times are consistently small. Since the period from task 30 to task 60 is small, does *not* observe a missed heartbeat from the `gpu-1` endpoint. However, it does see delays, and when it has not received results in more than twice the expected execution time, it sends backup tasks to other (slower) endpoints. After task 60, `gpu-1` resumes responding to task requests as expected and so execution returns to normal.

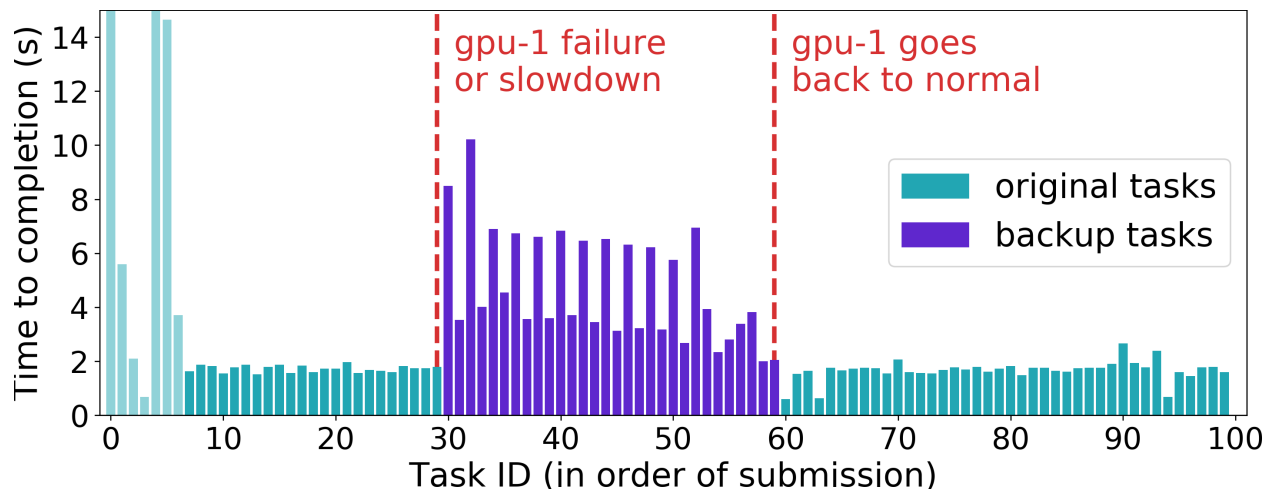


Figure 6.7: Tolerating failure of endpoints and slowdown of tasks with automatic delay-detection and backup tasks.

6.3.3 Macro Experiments

These experiments evaluate 's performance when subjected to high loads with tasks being scheduled in large batches (frequent on FaaS platforms). The first experiment shows how throughput and time-to-completion vary when many copies of the same task are run. The second shows how input size must be taken into account when making scheduling decisions. The third experiment shows that performs well when subjected to multiple task types at the same time.

Overloading Tasks

To test if 's scheduling system can distribute tasks across endpoints in a way that maximizes task throughput while maintaining low execution times, we ran 500 ParCount tasks with $n = 10^8$, in batches of 50.

Figure 6.8 shows that the **round-robin** strategy struggles to keep the average execution time of tasks low, since it simply cycles through endpoints and inevitably hits slowdowns when it sends tasks to the slower endpoints. While the **fastest-endpoint** strategy provides a slightly better task throughput than **round-robin**, its median task execution time is, in fact, *higher* than that of **round-robin**. This is because whereas the **round-robin** strategy naively balances load amongst the different endpoints, the **fastest-endpoint** strategy sends *all* tasks to the endpoint which provides the fastest runtime for the function in concern. As tasks are executed FIFO, this leads to tasks quickly piling up on this endpoint, which explains the higher median time-to-completion. The **smallest-EET** strategy does not suffer from such ailments since it takes into account pending-task predictions for each endpoint. We see that, after the first batch (in which it explores all endpoint groups), the **smallest-EET** strategy consistently keeps the time-to-completion for tasks low and demonstrates a task throughput that is several times higher than the other strategies.

Given a goal of explainability, we repeated the experiment above but with the simple

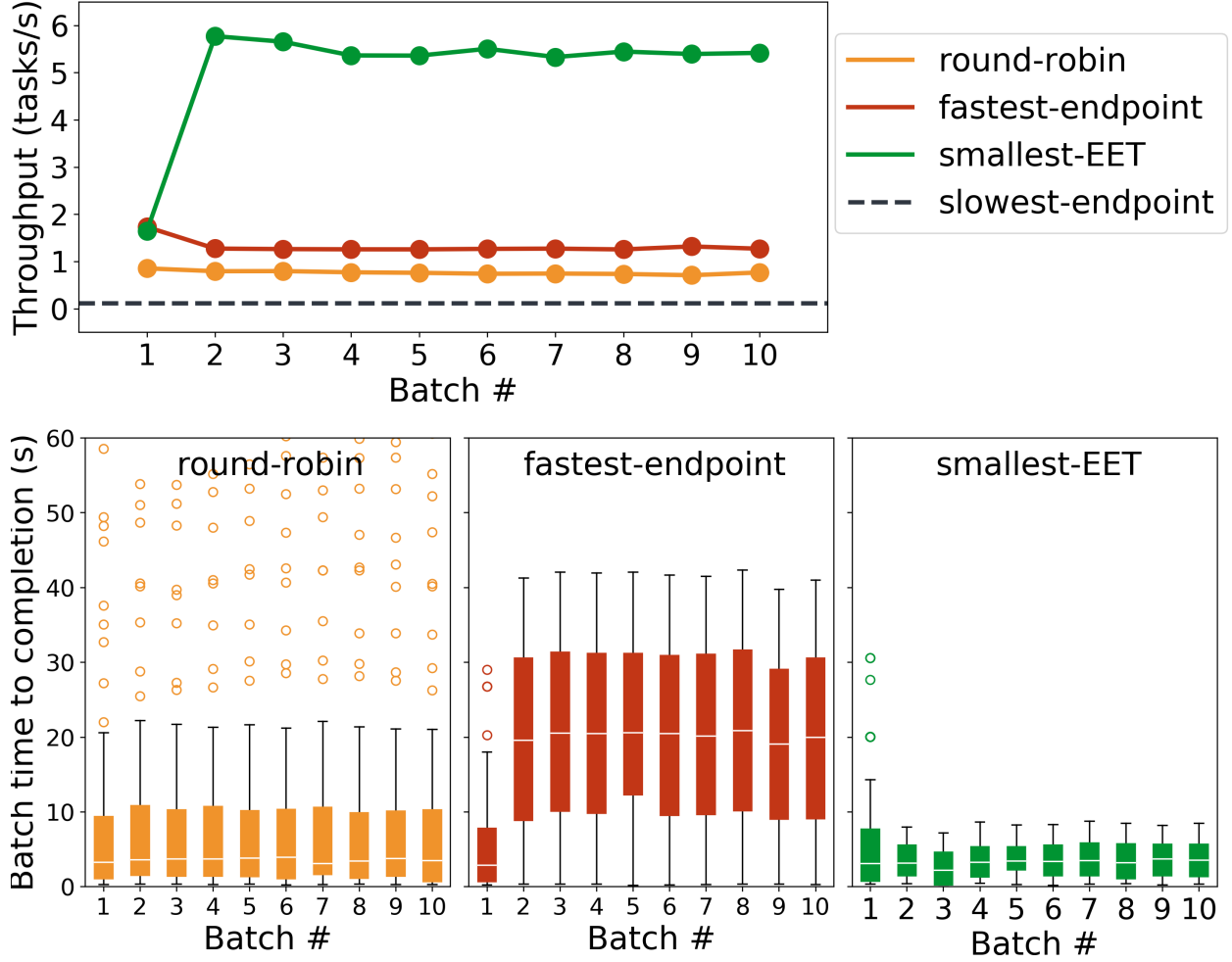


Figure 6.8: Overloading with ParCount tasks shows that, compared to baseline strategies, EET prediction significantly reduces time-to-completion and boosts throughput. The top figure reports the throughput of each batch. The bottom figure reports the box plot of task time-to-completion within each batch for each strategy. The box shows the first and last quartile; whiskers are shown at 1.5 interquartile range.

Counter function with $n = 10^8$. Figure 6.9 shows the distribution of tasks per endpoint produced by the **smallest-EET** strategy in this case. The figure also shows the relative speeds with which each endpoint can run this function—this is essentially just a measure of CPU clock-speed. Ignoring other execution costs, the optimal distribution for this task should match the relative speeds of the different endpoints. Figure 6.9 demonstrates that the **smallest-EET** strategy achieves a close approximation of this distribution, suggesting that its scheduling decisions are close to optimal.

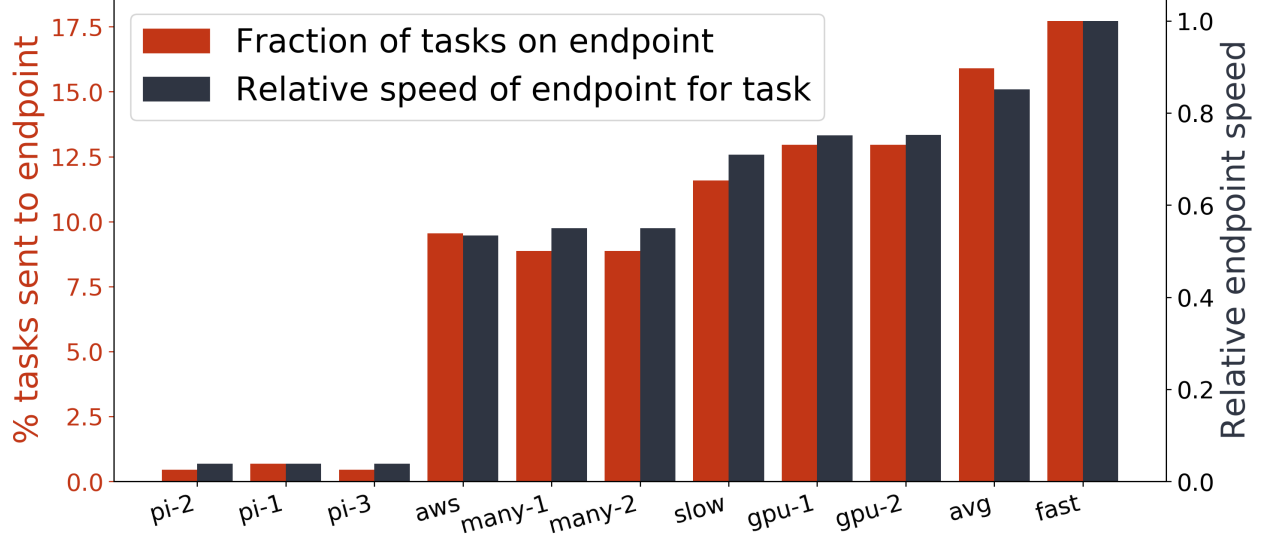
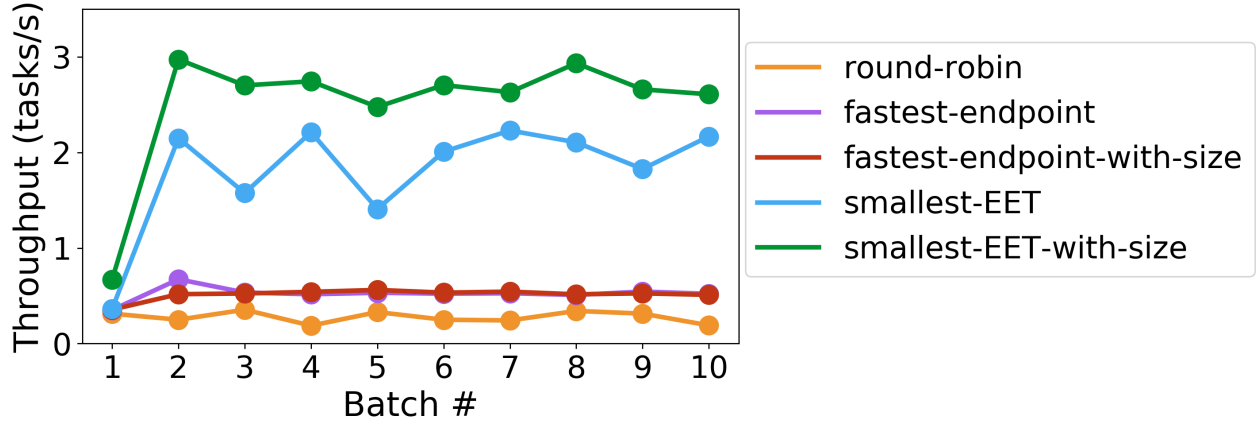


Figure 6.9: In red, fraction of tasks sent to each endpoint when Delta is overloaded with Counter tasks; in black, relative endpoint speed for a Counter function.

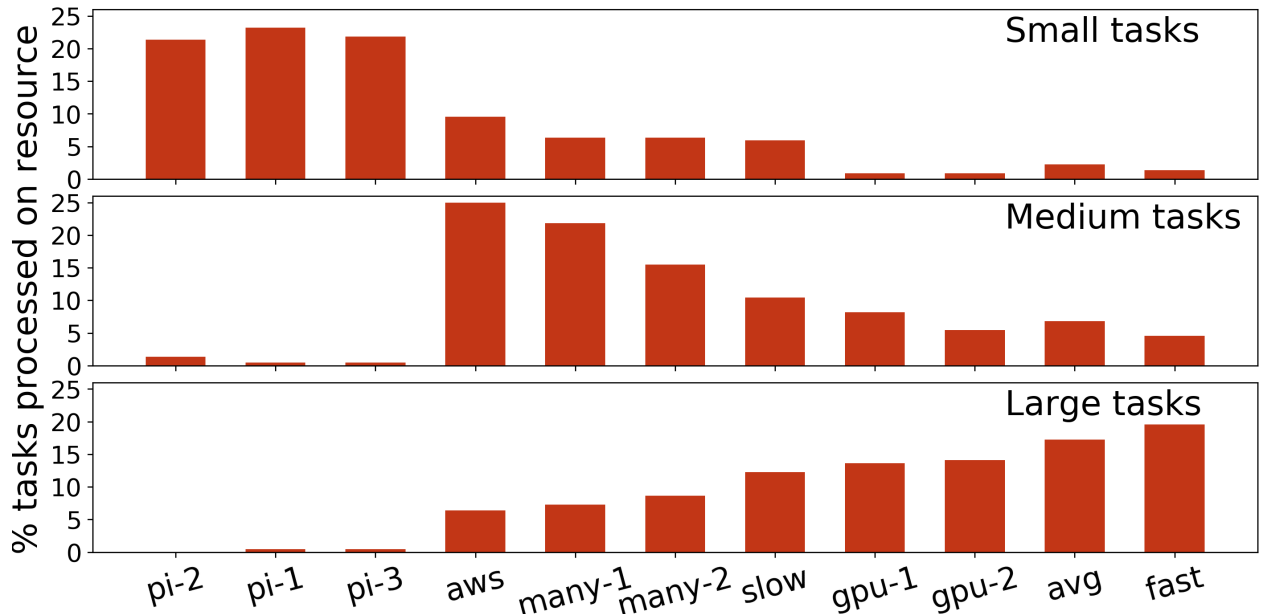
Input Size Trade-offs

The previous experiment demonstrated how tasks should be scheduled across heterogeneous endpoints when all tasks are uniform. We now consider the case where tasks are non-uniform, specifically, when different function inputs result in different runtimes: a common occurrence seen in many workloads. We use Counter tasks with $n \in \{2^{22}, 2^{25}, 2^{28}\}$ and schedule a total of 600 tasks, divided into 10 batches of 60 tasks each. Each batch consisted of 20 tasks of each of the three input sizes, randomly ordered.

The results in Figure 6.10 show that the **round-robin** strategy, by ignoring all features of incoming tasks and available endpoints, performs poorly and yields low throughput. The variance in the observed throughput between batches is because of the randomness in the order of tasks. The **fastest-endpoint** strategy, similar to the previous experiment, overwhelms the one endpoint which has the fastest runtime for the function. This is because this fastest endpoint, unsurprisingly, is the best for *each* of the three sizes. The figure also shows how the **smallest-EET** strategy performs, with and without taking into account input size in its runtime predictions. Without considering input size, the **smallest-EET** strategy has a high variance in the observed throughput. This is because it indiscriminately sends



(a)



(b)

Figure 6.10: Running tasks with multiple input sizes shows that treating different input sizes differently yields increases in performance. When under load, it is beneficial to send smaller tasks to slower endpoints, even Raspberry Pis. Figure a reports throughput when the **fastest-endpoint** and **smallest-EET** consider (“with-size”) and do not consider input size in runtime predictions. Figure b shows how the **smallest-EET** strategy allocates different sizes of tasks to endpoints.

tasks to where the function has been performing well recently. Occasionally, it sends tasks to the correct endpoints but, other times, it sends tasks to endpoints which have previously offered short runtimes due to the fact they were allocated tasks with small input sizes. When the **smallest-EET** strategy takes into account task sizes, its throughput remains consistently

high.

Figure 6.10 also shows how tasks are allocated to endpoints under the **smallest-EET** strategy. We see that tasks are distributed across the endpoints: small tasks are sent to the slowest endpoints, medium tasks to the mediocre endpoints, and the largest tasks to the fastest endpoints. We glean from this experiment that when put under load with tasks of different sizes, we can and must take advantage of even the slowest endpoints for improved performance.

Multiple Heterogeneous Tasks

This final experiment seeks to emulate real-world FaaS behavior via a workload involving three functions and, for each, three problem sizes: MatMul with $m = 20$ and $n \in \{2^8, 2^9, 2^{10}\}$; ParCount for $n \in \{2^{24}, 2^{26}, 2^{28}\}$; and FileI/O for $n \in \{2^{20}, 2^{22}, 2^{24}\}$. We assembled 111 of each of these nine {function, size} pairs, for a total of 999 tasks. These tasks were randomly shuffled and then dispatched in batches of 100, with each batch starting only when the previous batch had completed. Figure 6.11 shows the throughput observed when using different scheduling strategies. We used the input size predictor described in Section 6.2.5 in all cases.

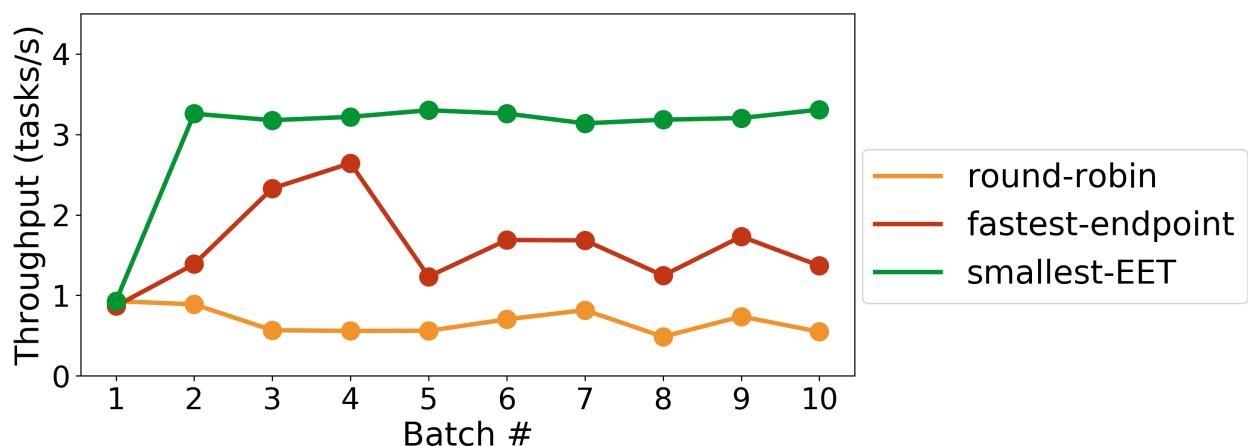


Figure 6.11: Throughput achieved by different scheduling strategies for the Multiple Heterogeneous Tasks experiment.

Table 6.2: Statistics for Multiple Heterogeneous Tasks experiment: average completion time for each task type, and average makespan both across all batches and across only non-exploration phases. All times are in seconds.

Strategy	ParCount			MatMul			FileI/O			Avg Makespan	Avg Makespan Post-Exploration
	Small	Medium	Large	Small	Medium	Large	Small	Medium	Large		
Round-Robin	28.7	30.2	41.8	30.6	26.8	32.6	30.1	32.6	42.4	154.2	159.4
Fastest-Endpoint	19.4	34.6	31.6	20.7	19.5	19.9	20.2	34.2	40.3	67.9	62.6
Smallest-EET	23.4	24.9	23.9	25.3	23.7	24.6	24.3	24.4	27.0	38.7	31.0

Unsurprisingly, **round-robin** does not distribute tasks efficiently to different endpoints. **fastest-endpoint** learns to send each function to the endpoint on which it runs fastest, but as it fails to account for any other factors, it shows high variance in throughput, getting lucky in some batches and overwhelming a few endpoints in others. **smallest-EET** quickly learns to allocate tasks to suitable endpoints and maintains a consistently high throughput. Figure 6.12 shows how each strategy allocates tasks to endpoints. **round-robin** of course distributes tasks evenly across the endpoints (the slightly uneven peaks are due to the random order of tasks submitted). **fastest-endpoint** overwhelms the handful of endpoints that it determines to be the fastest for each task type, sending most FileIO tasks to **fast-desktop** (fastest CPU and fast disk) and most MatMul tasks to the GPUs. On the other hand, the distribution observed for **smallest-EET** is more nuanced, depending on both task types and sizes. The smallest MatMul tasks are sent almost exclusively to the Raspberry Pis; longer-running task are sent to the GPU and manycore endpoints, where Tensorflow can exploit massive parallelism. Similarly, larger ParCount tasks are sent to the manycore endpoints and larger FileIO tasks to endpoints with fast disks and fast CPUs, whereas the smaller tasks are distributed amongst slower endpoints.

Table 6.2 shows the time to complete each task as an average across batches. It also shows the average makespan per batch, both across all batches and when ignoring batches in the exploratory phase. We see that **smallest-EET** achieves the fastest average completion time for only four of the nine {function, size} pairs, while **fastest-endpoint** gives the fastest average completion time for the other five. This result is not surprising as **smallest-EET** takes a more holistic view and spreads workload across the testbed. This distribution is

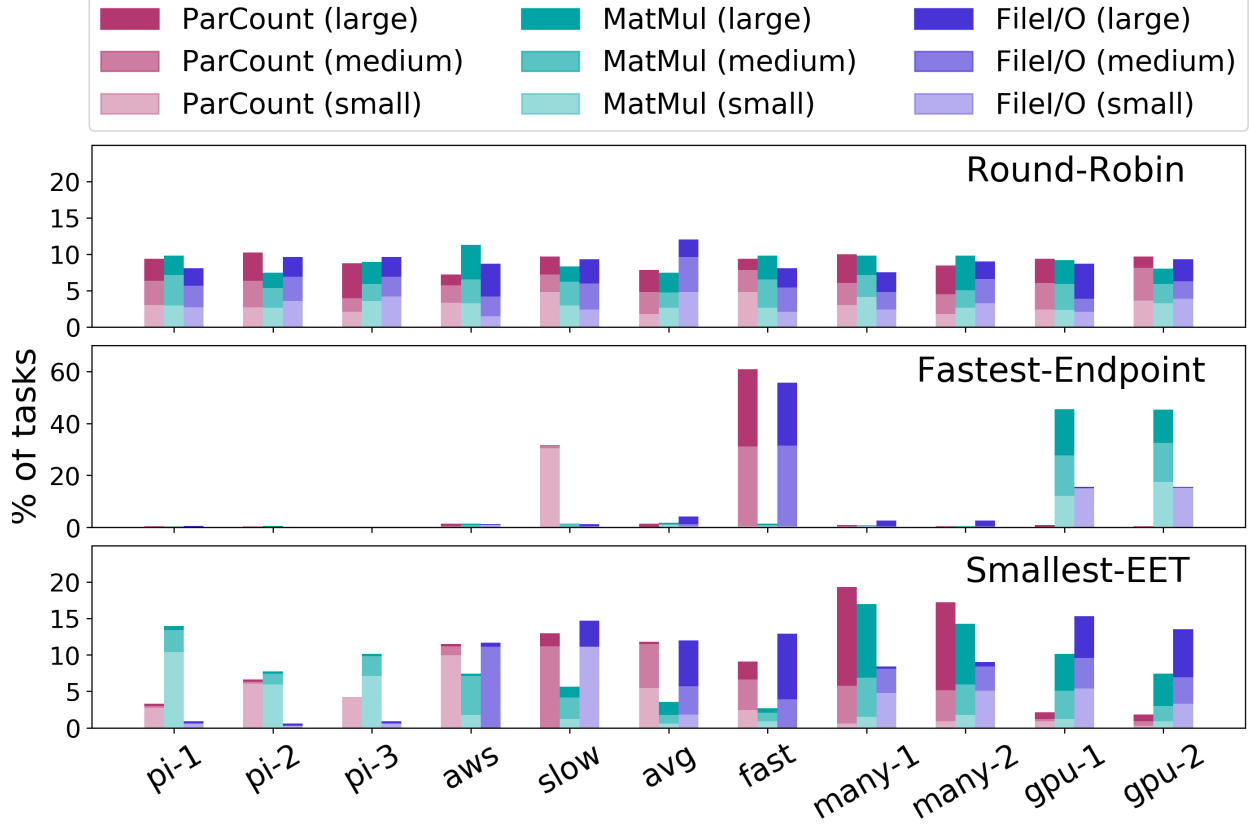


Figure 6.12: Distribution of tasks across endpoints for the Multiple Heterogeneous Tasks experiment. Note different scale for fastest-endpoint.

further highlighted by the fact that the average makespan for **smallest-EET** is less than half that achieved as **fastest-endpoint** and five times better than **round-robin**, when the exploration phase is excluded.

These results underscore our claim that scheduling task executions in a heterogeneous environment is a non-trivial undertaking that requires modeling the many complexities involved in running a computation remotely—and that doing so can yield considerable benefits.

6.4 DELTA+: Addressing Key Shortcomings

We extended this original work several times to include initial investigation into bursty workloads in the cloud and multidimensional notions of cost (see Fig. 6.13). While these additional efforts greatly complimented the original DELTA work, the system had to be

implicitly extended for scalability and less defined and granular notions of predictions. In other words, we had to recreate DELTA with a global scope that focused more on end user performance than specific predictable metrics.

DELTA+ builds off of a existing framework, DELTA [Kumar et al. \[2021b\]](#), that automated the resource selection process but only used traditional, time-based optimization metrics to make its resource selection choices. DELTA accounts for several limitations in the original DELTA architecture. Specifically:

- DELTA+ implements an n -dimensional notion of cost rather than simply accounting for time-based components;
- DELTA+ introduces the concept of budget to the task allocation architecture, allowing for bounds on optimizations that may have non-ideal asymptotic behaviors;
- DELTA+ switches to a fully probabilistic scheduling model, enabling better exploration of task allocation under a given optimization metric.

Computing cost tradeoffs

DELTA+'s primary purpose and extension from the original DELTA service is its altered cost-metric. While DELTA possessed configurable scheduling—demonstrated in the original paper [Kumar et al. \[2021b\]](#)—metrics had to be exclusively time-based.

The pervasive use of the cloud and with heterogeneity of modern computing in general makes the use of time alone without respect to any other metric untenable. For example, using DELTA's original system, optimizing for fastest function return time on massively parallel workloads would lead to the provisioning of as many cloud instances as possible. The allocation of such a large quantity of resources would conceivably be very expensive.

DELTA+'s solution allows for the user to express their optimization metric as an equation of factors, their valuation of the relationship between the different factors in the metric (i.e.,

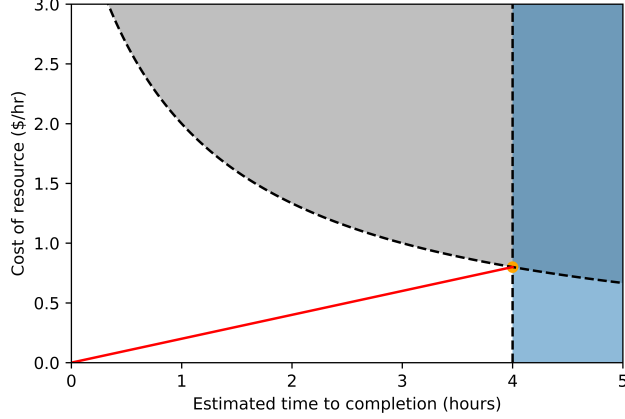


Figure 6.13: Illustration of optimizing across the tradeoff of resource performance versus cost. In this case, we have constraints of 4hr. maximum time and a \$4 budget (including a \$1 fixed overhead cost). Optimizing for likelihood of constraint satisfaction, the best resources lie along the red line (formed between the origin and the orange dot). Shading illustrates where time (blue) or cost (gray) budgets would be violated.

an hour of computation time may be worth \$5 to the end-user), and their ultimate constraints (both cost and time; see Fig. 6.13 for greater detail). In this paper, we used $\frac{\text{performance}}{\text{cost}}$ as our optimizer.

Once these metrics are established, DELTA+ functions identically to the original DELTA service, except that tasks submitted in batches are distributed probabilistically, the probabilities weighted by the calculated *utility* of a given task-endpoint combination. This enables consistent exploration of available resources and does not depend on human-defined determinations in the exploration-exploitation tradeoff.

Simulation of task allocation

To act as a pathfinder for our approach and to demonstrate the potential of using more complex cost relationships, we developed a simulation that demonstrates the allocation of batches of tasks across a number of computing endpoints, each with their own computational power and cost. The simulation begins with DELTA only assuming knowledge of resource cost, not resource performance. The simulation will demonstrate how we learn the tradeoff with respect to demonstrated cost.

Fig. 6.14 illustrates the simulation in which tasks were distributed equally at first, with respect to cost only, and then probabilistically allocated in subsequent batches to the instances with the greatest performance-per-cost. We can see that within the first 5 batches, the allocation of tasks comes within 10% of the ideal distribution. The target distribution represents allocating the majority of tasks to the two endpoints, 6 and 7, that have the highest performance to price value while minimizing those that have the worst performance and price—4 and 5. The other simulated endpoints had equal performance to price ratios.

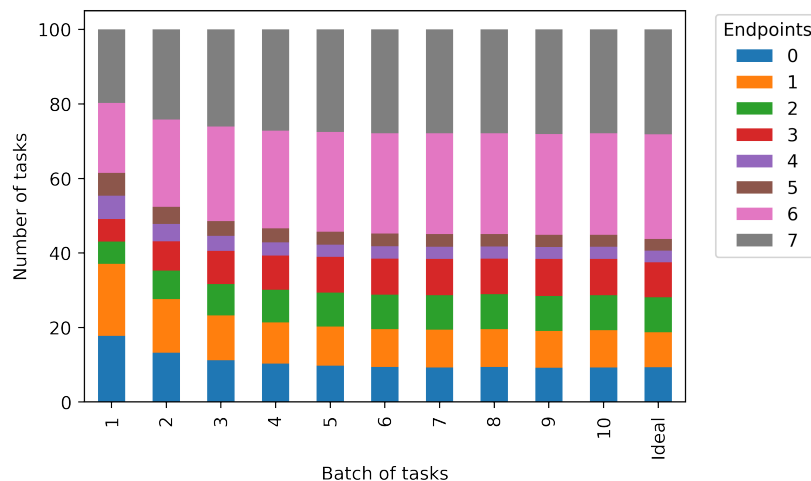


Figure 6.14: Simulated task distribution across 8 different endpoints for 10 successive batches of different tasks. Each batch illustrates how the 100 tasks comprising that batch are distributed between the 8 endpoints and this figure demonstrates DELTA’s learning capabilities. This simulation was run 1,000 times and the values presented are the mean of those runs.

This simulation demonstrates that simple multidimensional metrics hold at least some value for the potential allocations of tasks. The largest problems highlighted by this pathfinder are a) increasing confidence in our probabilistic allocations as time progresses to enable better fine-tuning during asymptotic executions; and b) the lack of automation between the computation relations of each metric (e.g., $\frac{\text{performance}}{\text{time}}$) and their relative values (e.g., $\$5 \approx 1 \text{ hr.}$).

CHAPTER 7

ADAPTIVE TASK MANAGEMENT

FOR MULTI-SITE SCIENTIFIC WORKFLOWS

Finally, we introduce Adaptive Task Management (ATM) as the basis of our current and final work. ATM is a natural successor from the DELTA system with several notable improvements. By taking into account all tasks that need to be scheduled and all resources available simultaneously, we overcome the key limitations of DELTA regarding scalability.

The core innovations of ATM lie in its multi-faceted approach to resource management:

- **Workload Execution Prediction:** ATM employs self-adaptive experimental design to continuously infer execution characteristics of diverse compute resources and workloads in real-time, dynamically adapting to changes in the workloads and the computational environment.
- **Cost Tradeoff Optimization:** Using probabilistic scheduling, ATM can make cost-aware task placement decisions with multidimensional costs, including time, monetary costs, and service allocation budgets.
- **Resource Shaping:** ATM introduces an automated configuration tool for a high-performance function-as-a-service (FaaS) framework that enables fine-grained control over parameterization and provisioning of resources, allowing dynamic resource reshaping and reconfiguration to match workload characteristics.

7.1 System Architecture

ATM is comprised of two primary components—the task placement algorithm and the supporting system (see Fig. 7.1). The system itself leverages the Globus Compute platform along with a control interface, a task tracker, submission handler, and a record of historical executions. The placement algorithm interfaces with the system by converting historical execution data into actionable decisions when the task handler receives a new task. We describe the system components in greater detail in the sections below.

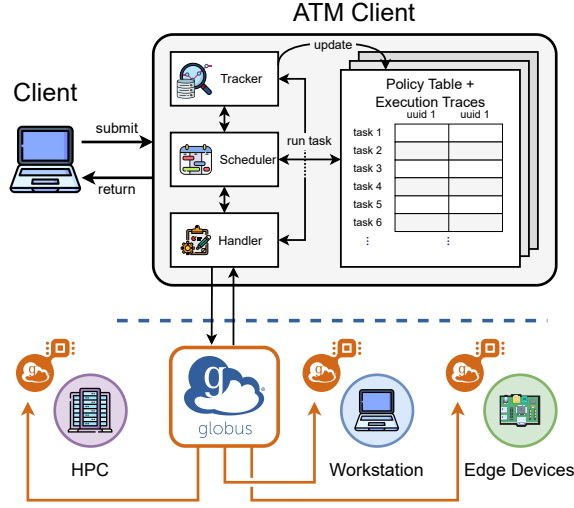


Figure 7.1: Overview of ATM’s client and operation within a computing environment using Globus Compute.

7.1.1 Infrastructure Control

Task-based execution frameworks, such as workflow systems and FaaS frameworks, provide an ideal execution model for exploiting the computing continuum. These frameworks break up applications into a set of *tasks* that contain both the necessary application logic, as well as information about the required environment, input arguments, and dependencies. They are designed to separate the task from the hardware which allows for flexibility in moving tasks around the continuum. We designed ATM to use Globus Compute (GC) due to its open-source SDK and endpoints, the ability to execute functions on arbitrary remote resources, and the published configurations for various HPC systems around the world. Our architecture is generally applicable to other task-based execution frameworks.

ATM interfaces with GC via the task handler. This component of the system tracks all task movement into or out of the system. Specifically, the handler can be thought of as the primary ATM service as it both initially receives the task from the client and interfaces with GC to forward the task to the necessary compute resource.

Upon receiving a new task or batch of tasks, the handler queries the scheduler for place-

Algorithm 1: Update Policy Table Values

Input: quants, throughputs, set of tasks T , set of machines M **Output:** quants

```
1 quants'  $\leftarrow$  copy(quants);
2 foreach task ID  $t = 1, \dots, |T|$  do
    // Extract the throughputs for the current task across all machines.
3     start  $\leftarrow (t - 1) \times |M| + 1$ ;
4     end  $\leftarrow t \times |M|$ ;
5     task_throughputs $_i \leftarrow$  throughputs $_i$  ( $\forall i = \text{start}, \dots, \text{end}$ );
6     avg  $\leftarrow |M|^{-1} \cdot \sum_{i=\text{start}}^{\text{end}}$  throughputs $_i$ ;
7     foreach machine ID  $m = 1, \dots, |M|$  do
8          $i \leftarrow (t - 1) \times |M| + m$ ;
9         current  $\leftarrow$  throughputs $_i$ ;
10        if avg > 0 then
11             $\delta \leftarrow \sqrt{\frac{\text{current}}{\text{avg}}}$ ;
12            quants' $_{t,m} \leftarrow$  quants' $_{t,m} \times \delta$ ;
13 return NormalizeQuants(quants',  $T, M$ );
```

ment. It then uses this information to wrap the task, provide the GC configuration necessary for the destination resource, and then submit the task to the GC service. The handler passes the task ID returned from GC to the tracker that then monitors for task completion. When a task is complete, the handler polls for results and returns them to the client.

7.1.2 Placement Algorithm

The placement algorithm used by ATM (we will refer to this as the ATM algorithm, for simplicity) is implemented as a policy table and control logic within the scheduler. Specifically, the *policy table* acts as the primary source of information to make task placement decisions. The policy table is updated using information from the historical execution traces (see Sec. 7.1.2 below) and contains an entry for every task-machine pair.

As executing every task across every machine is computationally untenable, we must infer execution profiles from historical data. Additionally, as task makespan is stochastic in nature due to machine and task input variability, we cannot simply use the most recent

execution time as ground truth. Moreover, to reduce system complexity, we do not try to predict contention between task types. Therefore, we must adapt to any incurred conflict over time, motivating a learned mapping between historic traces and an online policy table.

We characterize these two gaps as *workload profiling* and *cost estimation* and will address them individually.

Workload Profiling

We define workload profiling as the process of predicting task execution characteristics on a given resource based on the task’s prior executions on the same or other resources. While there is a substantial prior work in this space [Juve et al. \[2013\]](#), [Baughman et al. \[2018\]](#), [Gao et al. \[2018\]](#), [Calzarossa and Serazzi \[1993\]](#), [Ren et al. \[2010\]](#), [Hsia et al. \[2020\]](#), we opted for a novel generalizable method to ensure scalability and interoperability.

For each function, a profile is estimated by establishing a synthetic control [Abadie et al. \[2010\]](#) of the recorded execution of other functions weighted based on their respective likeness of execution characteristics, measured by cosine similarity and weighted by the overall magnitude of n -dimensional vectors corresponding to the n available endpoints (see Eq. 7.1; $\hat{T}_{i,j}$ is the estimated runtime of task i on machine j , $\mathcal{M}_i$ are the machines, $\cos(j, k)$ are the cosine similarities between the full performance vectors of machines j and k).

$$\hat{T}_{i,j} = \frac{\sum_{k \in \mathcal{M}_i} \cos(j, k) T_{i,k}}{\sum_{k \in \mathcal{M}_i} \cos(j, k)} \quad (7.1)$$

This methodology assumes that the execution time of a given workload on a given resource is consistent within some error bound. Second, we assume that workloads that perform similarly across $n - 1$ of n computing resources will also perform similarly on the n -th resource [Kamatar et al. \[2024\]](#). Finally, we assume that there are many functions that are functionally identical (i.e., any change that would cause the function to be registered as a

new function may not affect performance, such as renaming a variable). Prior studies have shown that this is a common occurrence with FaaS systems [Bauer et al. \[2024\]](#). An area of future work would consider methods to determine functionally similar functions based on their code, dependencies, historical performance, and other characteristics.

Cost Estimation

ATM’s notion of cost is based on the penalty incurred when executing a given task on a given resource. This cost could be time (i.e., makespan) or a monetary or allocation cost (e.g., ACCESS credits). ATM infers these costs based on the historical traces of given tasks and machines, subject to profile estimates performed following the method above. The ATM algorithm computes the policy table from historical information by taking a time-decayed average execution cost (i.e., the SUs used, makespan, etc.). If the intended value is supposed to be minimized, we take the inverse. From this information, we construct an intermediary policy table for each value. We aggregate these values into our final policy table via an element-wise product across the individual tables.

Bag-of-Task Placement

The primary use case for which ATM is designed is for large embarrassingly parallel batched compute workloads. Sec. 7.2 presents real-world science use cases that rely on this distributed scatter-gather pattern and which can be managed with ATM to meet some goal, such as reducing runtime or service units consumed.

ATM learns how to place tasks over a series of batches. As such, bag-of-tasks workloads must be broken up into batches. At present, we assume that users define these batches; however, in future work we will investigate intelligent auto-batching. The use of batches presents a tradeoff between smaller batches that may not saturate all available resources and larger batches which may slow the system’s learning of task execution characteristics.

Algorithm 2: The ATM task placement method.

Input: Batch of tasks B , table H where each entry $H(f, e)$ gives the predicted runtime for task function f on endpoint e

Output: A mapping M of tasks to endpoints

```
1  $M \leftarrow \emptyset$ ;  
2 foreach unique function  $f$  in  $B$  do  
3    $r \leftarrow H[f, :]$ ;  
4    $p \leftarrow \frac{r - \min(r)}{\max(r) - \min(r)}$ ;  
5   foreach task  $t$  in  $B$  with  $t.\text{function} = f$  do  
6      $e \leftarrow \text{Sample}(p)$ ;  
7      $M[t] \leftarrow e$ ;  
8 return  $M$ ;
```

Once batches are established, ATM can ingest each batch and schedule the component tasks based on the current state of the policy table. Specifically, ATM uses the policy table, along with the batch task types and available machines, and probabilistically allocates those tasks to machines. Using the values in the policy table, ATM calculates a normalized vector of weights for each task type across all available machines. For each machine-task type pair, ATM uses this normalized value as a weight to probabilistically send a proportion of tasks of that type to the specific machine. This can be seen more explicitly in Alg. 2.

Under this execution regime, assuming time is the only optimization factor, this algorithm acts as a dynamic load balancer, allocating tasks such that batch makespan across all machines is equal. By ensuring that all tasks return at nearly the same time, system utilization is increased due to a reduction in time spent waiting for the next round of tasks.

DAG-based Placement

ATM also supports placing tasks in a non-batchable workflow. The primary difference is the need to consider a queue for each machine based on the number of tasks that have already been sent to each machine. To accomplish this, we compute the relative policy value between two machines, adjust for the number of tasks, and then recompute the normalization for the

Algorithm 3: Schedule a single task in a non-batchable workflow

Input: policy table Q , historic traces E , scheduled tasks T , new task t , set of machines M

Output: $m \in M$

```
/* Compute current queue length and adjust policy values */
1 for  $m \in M$  do
2    $c_m \leftarrow |\{t' \in T \mid \text{assigned}(t') = m\}|$ ;
3    $\tilde{Q}_m \leftarrow \frac{Q_m}{c_m + 1}$ ;
/* Normalize adjusted policy values */
4 for  $m \in M$  do
5    $\pi_m \leftarrow \frac{\tilde{Q}_m}{\sum_{k \in M} \tilde{Q}_k}$ ;
/* Sample machine for the new task */
6  $m^* \sim \pi$ ;
7 return  $m^*$ 
```

policy table based on the new task addition. In other words, if one machine has a value in the policy table that is $4\times$ another machine and the first four tasks have been split evenly between the two machines, the next task is $8\times$ more likely to be placed on the first machine than the second. This algorithm is shown in Alg. 3.

7.1.3 Resource Shaping

We use Globus Compute’s endpoint templating to remotely configure and shape compute resources for tasks. For example, in some scenarios it may be best to execute as many single-threaded tasks as one has logical threads on a CPU. However, many applications and modern Python libraries (e.g., `numpy`) natively vectorize or parallelize operations. Because of this, co-tenancy and context switching can reduce overall throughput if tasks are executed on each thread but throughput will also be reduced if each endpoint receives only a single task. Beyond CPU utilization, other resources, such as memory, accelerator, and I/O bandwidth contention, can lead to not just diminishing returns but performance reversions by packing too many tasks to run concurrently. Therefore, a balance must be struck in the execution of

tasks between over and underloading a node. This balance shifts with each unique function and so must be updated dynamically. Globus Compute endpoints allow for such on-demand endpoint shaping by allowing updates to the number of tasks per endpoint without requiring an interruption to that endpoint’s service.

To perform endpoint shaping, we form a simple probability distribution around suitable configurations and narrow this distribution over time to balance the exploration-exploitation tradeoffs. This type of process has been proposed previously for configuration of cloud instance sizing [Wu et al. \[2019\]](#), [Vashishth et al. \[2017\]](#), [Meyer et al. \[2021\]](#) but the application of this type of methodology is less well explored for mapping tasks to fixed hardware. Some initial investigation in this area has focused on heterogeneous environments [Yang et al. \[2011\]](#), job submission parameter tuning [Robert et al. \[2020\]](#), [Menon et al. \[2020\]](#), and balancing multiple optimization criteria [Zhu et al. \[2022\]](#), like energy [Miyazaki et al. \[2018\]](#).

Additionally, we form the probabilities within this system very similarly to how we assign probabilities to different endpoints used for execution. In this case, however, we can use the relationship between endpoint configurations to make better guesses. To do this, we sample from an aggregate distribution created by a weighted mean of all previous runs. This allows for better exploration of the sample space and more informative sampling across it as we will tend to sample from midpoints between popular configurations, resulting in sampling from an undersampled portion of the search space.

7.1.4 Error Handling and Retries

ATM extends the error handling and retry logic of the underlying infrastructure through a result polling process. Specifically, ATM polls tasks at a configurable time multiplier with respect to the predicted execution time for a given task. For the experiments presented in this paper, we set the multiplier to $3\times$. If no result is ready, ATM retries the task on a resource with a shorter predicted execution time, should one be available and not violate

user-specified QoS. Finally, ATM waits for *both* tasks, dropping the straggler once the other has completed. If user-defined QoS is about to be violated, ATM will cease this execution loop for a straggling task and return an error to the user.

7.1.5 *Sample Execution*

We now describe a sample execution of a bag-of-tasks workload to illustrate how ATM works as a complete system.

Startup

At startup ATM loads any saved policy table; otherwise it prompts for the target Globus Compute Endpoint UUIDs. With this information, ATM iterates through the policy table and sends a simple no-op task to each endpoint in order to obtain its status. This serves two purposes: 1) it ensures that the relevant endpoints are online and 2) it initiates the creation of a batch job (and Globus Compute pilot job) for any batch scheduled system. Additionally, this initial status task serves to query system information for any previously unseen resource, allowing for a more intelligent, heuristic-based starting point with which to make scheduling decisions.

Initial Task allocation

Once the table is initialized and the no-op tasks have returned (or the initialization timeout has been reached in the case of a queued system), then the first batch of tasks provided by the user can be submitted for placement. To do this, ATM will infer the initial number of tasks to be sent to each endpoint based on the inferred capabilities of the machine obtained from the status message (i.e., it will send a number of tasks proportional to the number of cores of a machine, assuming no previous information is available regarding the tasks or endpoints). For each task type and endpoint pairing, the relevant tasks and data will

be packaged into a Globus Compute batch to be submitted to the endpoint with an ATM wrapper that provides the necessary monitoring information upon function completion or return.

System Feedback Mechanism

After all tasks are submitted in the first batch, ATM then begins to poll for results from each endpoint, logging the relevant completions and loading task results or errors as they become available. Based on the timing information tracked by the ATM wrapper, the policy table is updated according to the process described in Sub. 7.1.2. Following the completion of the entire batch, the new batch splits and task placements are calculated and are submitted to Globus Compute for execution again. Additionally, if more endpoints have come online during the execution of the first batch, they will be incorporated into the new placement decision. This process repeats until the bag-of-tasks is complete. Following completion, the global table is saved for later use.

7.2 Methods

In this section, we describe our heterogeneous testbed and the workloads and applications that we use to evaluate ATM.

7.2.1 Testbed Resources

We use a diverse testbed of four desktop computers and eight HPC systems. The extreme heterogeneity of this testbed allows us to explore ATM’s ability to balance across diverse, real-world resources, including scheduled systems. The desktops are four department-provided workstations (see Table 7.1). The HPC systems are Purdue’s Anvil [Song et al. \[2022\]](#), Indiana University’s Jetstream2 [Hancock et al. \[2021\]](#), TACC’s Stampede3 [tac \[2023\]](#), SDSC’s

Expanse of California San Diego [2025], PSC’s Bridges2 Brown et al. [2021], TAMU’s Aces and Faster, and the university and department clusters associated with the authors (see Tab. 7.1 for details). These machines are not only geographically distributed, but cover a wide breadth of compute resources.

Resource	CPU	Memory
Hawfinch (desktop)	6-core Intel Core	16GB
Ryerson (desktop)	4-core Intel Core	8GB
Kestrel (desktop)	4-core Intel Core	8GB
Roomba (desktop)	4-core Intel Core	16GB
Uni. Cluster (node)	2×14-core Intel Xeon	256GB
Dept. Cluster (node)	2×12-core Intel Xeon	128GB

Table 7.1: List of non-public resources used for ATM evaluation.

For each of these systems, we deploy a Globus Compute Multi-user Endpoint (MEP) on each machine (or, in the case of the scheduled systems, on a login node). The MEPs depend on a recent version of Python as well as the `globus-compute-endpoint` package being installed. Additionally, for some of the tests, we assume that the relevant data is accessible to each machine, specifically those using the SEBS tasks. We consider data movement to be outside the scope of this paper and refer to other methods for managed transfer for task-based applications between machines Pauloski et al. [2023].

7.2.2 Simulator

To disentangle machine and network variability from ATM’s performance, we developed a simple simulation tool that allows us to directly compare the performance of different algorithms across a diverse set of environments. We elected to build a simulator rather than use an existing one as the entire purpose is to realistically generate execution traces for the system to process. Additionally, ATM’s role as client-side software eases the disentanglement of execution, allowing us to completely control the execution rather than forcing the framework to “live” inside a simulated environment.

This simulator models the dynamics of two values—parallelizability and cost. For machines, we capture the parallelization available (e.g., number of cores or threads) and the processing speed (i.e., core-GHz). For tasks, we define a maximum amount of parallelization and assume perfect runtime scaling. These values are not visible to ATM.

In our experiments using this simulator, we place tasks on different machines and compute task runtime algebraically. This allows a more direct comparison between machines and an infinitely larger search space. Moreover, we can use these values and the simulated workloads in order to calculate the optimal task placement. Finally, we can vary the amount of noise within each simulated task and machine to add to the stochasticity of the simulation.

This simulator is integrated in ATM by replacing the execution infrastructure and task handler. This allows it to be modularly substituted without affecting the rest of an experimental workflow. Additionally, this change requires the use of preregistered tasks—the lack of true task execution limits the ability of the system to handle unseen tasks as the simulator must have ground truth knowledge of task execution characteristics.

7.2.3 Workloads

We evaluate ATM using various workloads, including both synthetic benchmarks and real-world science applications. These workloads were derived from two benchmark suites: the Serverless Benchmark Suite (SeBS) [Copik et al. \[2021\]](#) and the Task Performance Suite (TaPS) [Pauloski et al. \[2024\]](#). We provide an overview of these two sets of benchmarks below and describe the applications we used from each. We generate bags-of-tasks from all the applications and non-batched task submissions from the synthetic applications.

Synthetic Tasks

To isolate the performance of ATM, we use matrix multiplication and sleep tasks with various input arguments to represent compute-bound and concurrency-bound applications

Serverless Benchmarks

SeBS [Copik et al. \[2021\]](#) represents a wide range of traditional fine-grained serverless tasks that are intended to execute quickly and maintain little to no state. In many of the SeBS applications, data is downloaded from, or uploaded to, the Internet. As ATM only addresses compute capabilities, we allow data to be transferred by the application outside of the compute control band.

From SeBS we use the following applications: HTML rendering, thumbnail generation, video processing, image compression, image recognition, pagerank, graph minimum spanning tree calculation, graph breadth first search, and DNA processing and visualization. These represent traditional serverless workloads and are included to demonstrate a worst-case scenario for ATM.

Science Workflows

TaPS [Pauloski et al. \[2024\]](#) includes several task-based applications, most distilled from domain science workflows, to serve as a real-world benchmarking suite for task-based execution frameworks. For our evaluation, we use the Molecular Design and Protein Docking applications. These science applications utilize a general scatter-gather task pattern and can easily be batched, scaled, and evaluated.

To use ATM with these applications, we removed the wrapping provided by TaPS and used only the core portion of these applications. This ensures that our evaluation best demonstrates system performance on real workflows rather than showing its interplay with other task management systems. Moreover, for the Molecular Design application, we scale up the molecular search space to include the entirety of QM9 [Ramakrishnan et al. \[2014\]](#), a dataset of more than 130k organic molecules, to ensure that we can utilize our large testbeds.

For each application, we outline within the experiment the configuration used to generate the tasks, as well as provide the size of the individual batches. Additionally, we present the

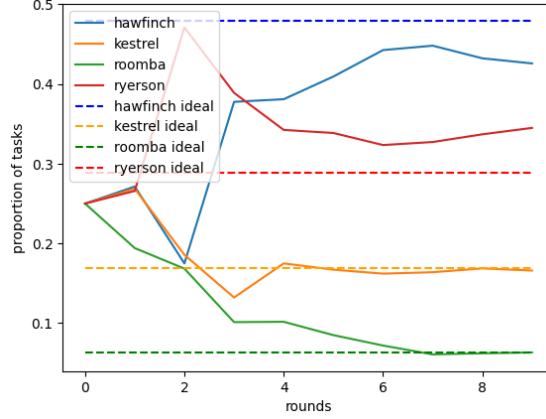


Figure 7.2: Demonstration of how proportions of batched tasks become reallocated to different machines over time with respect to ideal machine performance.

science results (i.e., the direct results of the design and docking codes) where appropriate.

7.2.4 Baseline Algorithms

To adequately evaluate the novel placement algorithm developed for the ATM system, we compare against seven well-known heuristic-based methods. These are as follows: round robin (equivalent to proportional allocation by machine), least-loaded (equivalent to proportional allocation by worker with updates based on completion rates), Multi-Heuristic Resource Allocation (MHRA) [Juarez et al. \[2018\]](#) (ranks tasks using and assigns them to resources by balancing factors like energy consumption and makespan), Coasters [Hategan et al. \[2011\]](#) (a provisioning algorithm for highly distributed applications), Swift/K [Wilde et al. \[2011\]](#) (a feedback-based strategy using a points system to track performance), a shortest-task-first strategy (we schedule the shortest task-machine pair first and so on), and a mini-max strategy (where we schedule the task-machine pair where that optimal machine selection is the longest duration task).

7.3 Evaluation

To demonstrate the novelty and impact of our framework, we study the performance of ATM both as an aggregate system as well as through evaluation of its component parts. These studies demonstrate the contribution of each novel functionality presented by ATM and motivate the use of ATM as a single cohesive framework. This section evaluates several of the key capabilities of ATM individually before demonstrating its abilities as a framework through running a large, multi-site science workflow.

7.3.1 *Simple Load-Balancing*

We first explore the performance of ATM with a simple use case of homogeneous matrix multiplication tasks (see Sec. 7.2) executed on the desktop testbed. Using simple $4,096 \times 4,096$ square matmuls demonstrates the ability of ATM to learn the ideal distribution of tasks—effectively, demonstrating the *load-balancing* capabilities of ATM. We execute 144 tasks in each batch over 10 batches for 1,440 total tasks and assume no prior knowledge of the machines. Fig. 7.2 illustrates the proportion of tasks assigned to each machine for each round. It also includes the ideal proportion of each batch determined through locally executing the target function and measuring task throughput on the endpoint without any systems overheads. Our experiments show ATM is able to find near-ideal balancing of tasks within 5 batches, with increasing convergence to within 2.5% of the ideal proportions on average.

7.3.2 *Task-Machine Affinity*

We use our simulation (see Sec. 7.2.2) to demonstrate how different tasks are better suited to different types of machines and, as such, have greater affinity for specific machines. Simulation is used rather than real machines as it allows a more diversity and more precision in

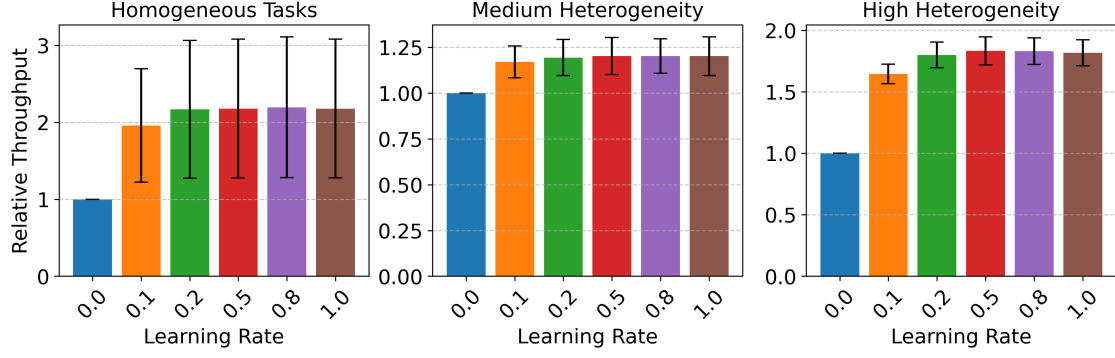


Figure 7.3: Sensitivity analysis of performance impact of learning rate with different levels of task heterogeneity.

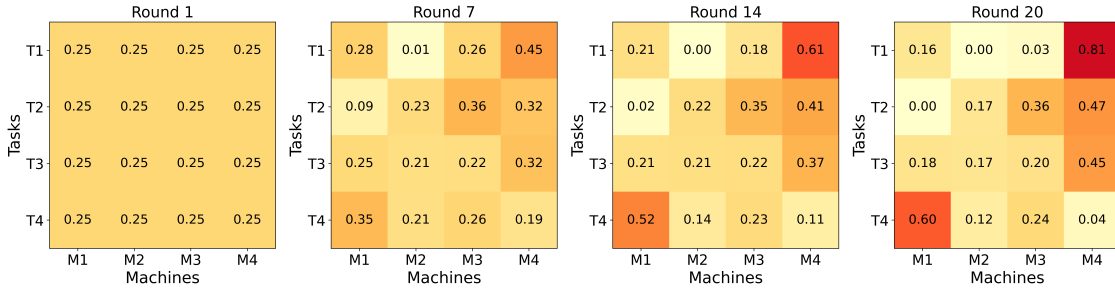


Figure 7.4: Demonstrates how tasks migrate over time to show affinity to the resources for which they are most compatible within an experiment.

machine and task types than using real-world tasks and machines.

We use four base machine configurations (what is their relative differences, insert table: cores/speed) and four task types (define for a single core the runtime and max parallelism). In each run of the experiment, we slightly noise the machine configurations ($\pm 10\%$). In each batch we pick at random 40 tasks and noise each task runtime ($\pm 10\%$).

We ran 100 experiments to ensure consistent convergence. Fig. 7.4 shows how task migration occurs over time and becomes self-reinforcing. This self-reinforcement occurs as, when a task is particularly well suited for a specific machine, a greater proportion of that task will be placed on that machine, leading to increasing slowdowns for other types of tasks on that resource, which will be increasingly offloaded.

7.3.3 Learning Rate

Due to the overheads in the system, many heterogeneous but very short running tasks (i.e., <1 second) represent the worst case scenario for ATM and are generally representative of more traditional serverless tasks. . One such characteristic of these types of tasks is that they are highly heterogeneous but highly consistent [Copik et al. \[2021\]](#) (i.e., many tasks that vary greatly from each other are run many times each). Under such conditions, we must consider how altering the learning rate of the system affects the results.

To evaluate ATM’s performance in this worst case and to analyze the effect of learning rate on the system, we run an experiment of 500 SEBS [Copik et al. \[2021\]](#)(see Sec. 7.2) tasks. To generate our homogeneous runs and medium and high heterogeneous runs, we take a random subset of SEBS tasks that can be used for each experimental run, selecting one, four, and all respectively. We randomly sample at every batch these tasks from the different SEBS subsets. These are run on the desktop testbed 100 times.

Fig. 7.3 shows how ATM’s learning rate (see Sec. 7.1) affects performance with both heterogeneous or homogeneous workloads. The sweep in learning rate from 0 (no adaptation) to 1 (making decisions based only on the last batch) shows a moderate increase for all task configurations up until a 0.5 learning rate. As these experiments are performed using extremely short running tasks, we only see an increase of 20-110% using 0.5 rather than 0 for learning rate. Given these results, we chose a learning rate of 0.5 for all of our tests, unless otherwise stated.

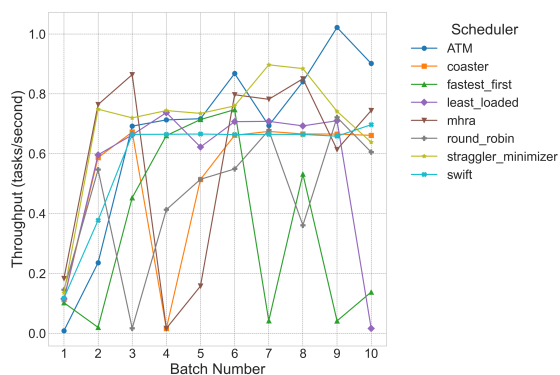
7.3.4 Scheduling Algorithm Comparison

As discussed in Sec. 7.2, we have implemented seven scheduling algorithms to compare with our novel task placement strategy. We evaluate the performance of our algorithm against the others first using the SEBS benchmark suite as it offers the widest range of tasks. We also perform our evaluations on longer running matrix multiplication (matmul) tasks to more

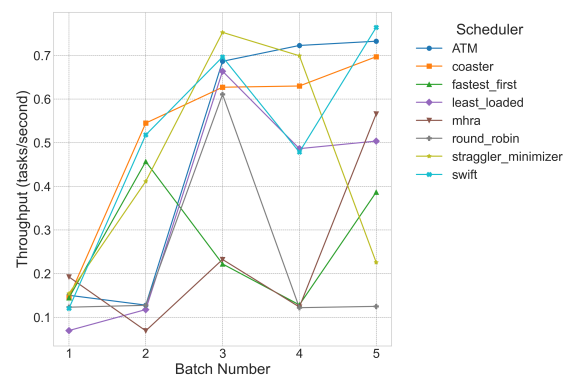
closely demonstrate the performance of our algorithm in compute-intensive workloads. For all of these experiments, we analyze the results and compare the algorithms on throughput, variability (i.e., moving standard deviation), and rate of convergence (or, how quickly each algorithm settles into its best configuration).

Fig. 7.5a illustrates the ATM algorithm’s performance against our selected baselines. This experiment demonstrates that ATM is able to learn a stable system configuration and avoid batch to batch variability compared to the baseline scheduling algorithms while maintaining comparable and even superior performance. This experiment was run 10 times on our desktop testbed and selected batches of 500 random tasks from the SEBS benchmark suite.

Fig.7.5b demonstrates the performance of ATM on the longer running matmul tasks. Given the consistency of execution and compute-bound nature of these workloads, we see ATM achieve comparable task throughput when compared to the other scheduling methods.

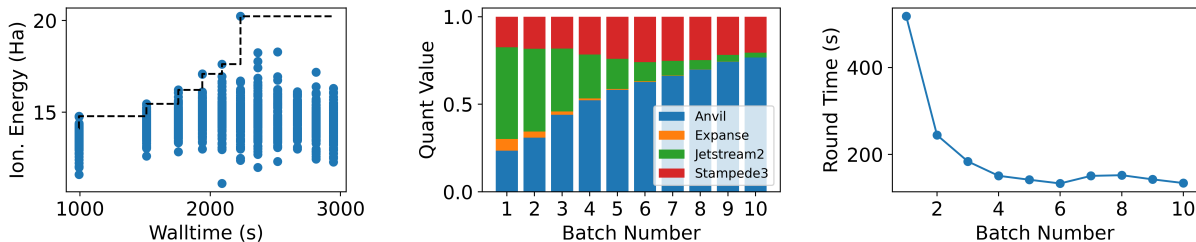


(a) Comparison of scheduling algorithms using Globus Compute for batches of tasks randomly sampled from the SEBS [Copik et al. \[2021\]](#) serverless benchmarking suite.



(b) Comparison of scheduling algorithms on matrix multiply tasks across four cloud instances on Jetstream2.

Figure 7.5: Comparison of scheduling algorithms: (a) SEBS batches using Globus Compute; (b) matrix multiply tasks on Jetstream2.



(a) Discovery results on four HPC systems. (b) Task distribution across the systems. (c) Per-batch timing across systems.

Figure 7.6: Molecular design experiment across four HPC systems, demonstrating the real science results and how learning how to better distribute those tasks over machines continuously accelerates execution.

7.3.5 ATM Scaling

We conducted scaling and overhead tests to ensure ATM did not affect system performance. This study was conducted without task submission as tasks are passed to Globus Compute for execution and thus not under our control. The following results presented represent client-side values as ATM is purely a client-side framework. To test ATM’s overhead at scale, we randomly generate unique endpoint and function IDs, as ATM does not directly need to execute these functions, and pass these as input to ATM. We measure total overheads within the system by running the entire ATM control flow, replacing the submission step with a function that just returns successfully with a random runtime value to fill the place of live system feedback. Again, the value of this feedback has no impact on the scheduling time taken by the system. These experiments were performed with the client running on a 2021 MacBook Pro. We tested scaling of task types and endpoints in a sweep, incremented by powers of 2, from 1 through 16,384 for both values.

At 16,384 unique tasks on 128 endpoints, ATM demonstrated an average scheduling overhead of 0.3282 seconds with a standard deviation of 0.1146 seconds over 10 experimental runs. Additionally, at 16,384 unique task types and 16,384 machines, scheduling took 27.51 seconds on average. This is an increase of approximately $84\times$ scheduling overhead compared to a $128\times$ increase in unique tasks. These results confirm ATM’s overhead on the client is

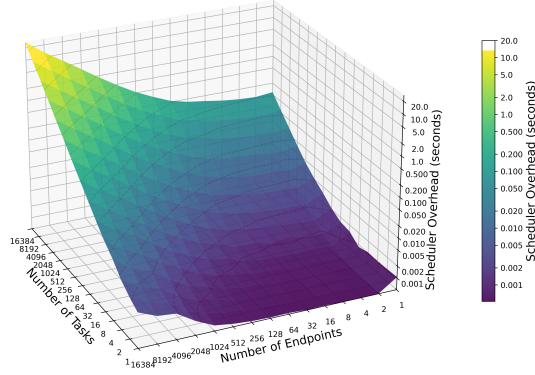


Figure 7.7: Total ATM overhead for different machine and task quantities.

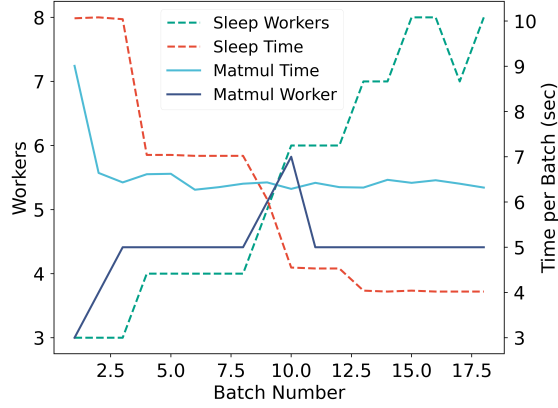


Figure 7.8: Endpoint shaping on a single machine with both sleep and matrix multiplication tasks. Worker configuration adapts based on batch time for the different tasks. Sleep converges to the number of threads on the machine (8) but matmul converges to 5 workers as the `numpy` library uses vectorized operations.

negligible.

7.3.6 Endpoint Shaping

We evaluate the endpoint shaping algorithm independently using 72 synthetic tasks per batch over 20 batches on the desktop testbed (see Fig. 7.8). In these experiments, we witnessed a 20% to 50% time savings over static resource configuration, demonstrating how our endpoint shaping algorithm adapts to the inherently parallel nature of matrix multiplication tasks as well as the single-threaded limits of sleep tasks.

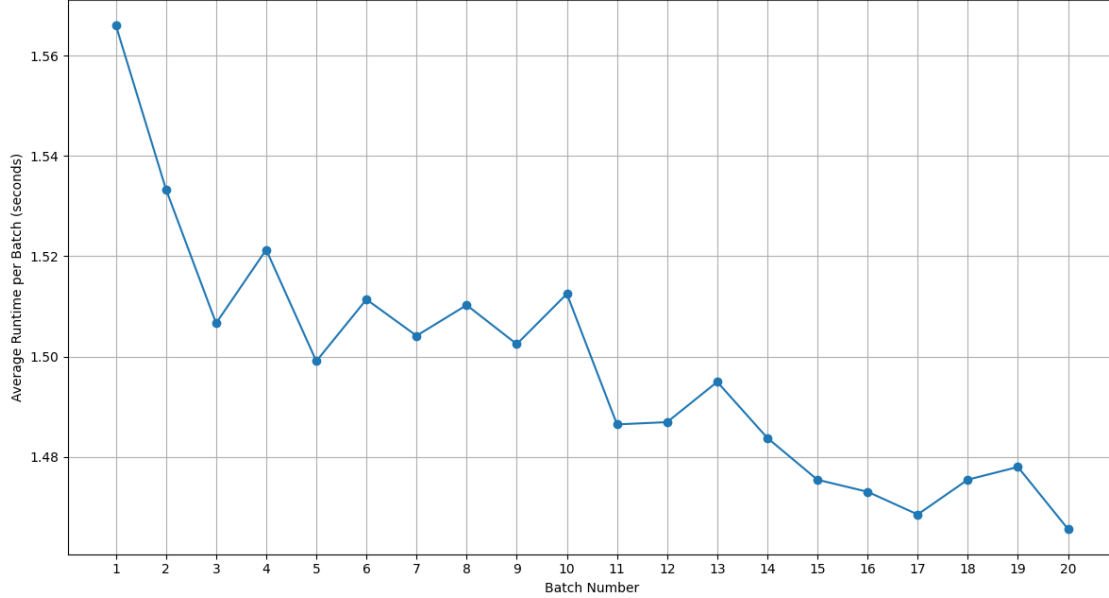


Figure 7.9: Endpoint shaping experiment on the desktop testbed demonstrating the 6.5% reduction in batch time achieved by adding endpoint shaping after task distribution convergence over 20 prior batches.

To evaluate MEP shaping in a real world scenario, we perform an experiment that starts with a converged task distribution (determined through local throughput testing) across our desktop testbed and then turn on endpoint shaping for our $4,096 \times 4,096$ matmul task. Fig. 7.9 demonstrates an 6.5% improvement in makespan, stabilizing after approximately 10 batches of tasks.

7.3.7 Science Applications

To demonstrate the application of ATM to current science workflows, we adapt molecular design and protein docking applications to be evaluated running across multiple computing sites using ATM. Both of these use cases represent a general scatter-gather compute pattern with substantial compute in both the scattered tasks as well as in the aggregation step.

Our evaluation of multi-site execution for these two applications uses NSF ACCESS [Boerner et al. \[2023\]](#) machines. For each science app, we run two experiments—one assumes on-demand queues for the batch scheduled machines and the other runs in real time, waiting

for queues at the beginning and throughout the workflow as necessary. (Note that waiting for queues necessitates results be presented as individual runs and therefore are used for illustrative evidence as the exact queue conditions are not reproducible.)

Specifically, we composed a molecular design experiment of more than 1,200 molecular simulations using ATM on the four real ACCESS systems discussed in Sec. 7.2—Anvil, Stampede3, Jetstream2, and Expanse.

In our experiment using just four ACCESS machines and real-time queues (see Fig. 7.10a), we see that ATM is able to simultaneously launch tasks across all four machines and then iteratively rebalance those allocations based on execution characteristics from the previous batch of tasks (see Fig. 7.6b). Over the course of 10 batches, ATM determines that Anvil and Stampede3 are the most advantageous systems on which to execute the tasks. Additionally, Fig. 7.6c demonstrates that there is a momentary increase in time as Expanse becomes no longer used around batch 7 but that the slowdown is corrected with further rebalancing by batch 10, now using only three of the four machines, saving cost. Overall, this experiment demonstrates the ability to save more than 60% time and more than 70% cost over equally loading all resources. Results of the workflow are shown in Fig. 7.6a, illustrating how the decreasing batch time leads to more rapid results.

7.3.8 *Beyond Walltime*

One important feature of ATM is its ability to incorporate multiple considerations of cost—requiring only a user-defined method to "trade off" between costs. One of the key contributions of this paper is its real-life contributions to accelerating science across multiple HPC systems simultaneously and to do so while accounting for both application performance, execution time, and costs.

ACCESS Simulation

To demonstrate the ability of ATM to manage resources with multidimensional notions of cost at much larger scales, we use our simulator (see Sec. 7.2.2) to analyze the behavior of ATM within an environment that mimics all of the resources present in ACCESS. Specifically, our simulation includes 15 systems from the ACCESS list of resources and imagines a workload comprised of mixed batches of highly parallel and highly sequential tasks. Due to the conversion rate of ACCESS credits to system-specific SUs, we design our experiment to receive feedback in terms of time and ACCESS credits with the optimization factor set as task throughput per credit. The simulation consists of 10 batches of 10,000 tasks each. We assume we can request machine time by full node and that the ACCESS credits we are using have been pre-exchanged. For the workload, we simulate a mix of sequential and multi-threaded tasks that each take approximately 4 to 5 core-minutes due to randomness added to the simulated execution time of each task.

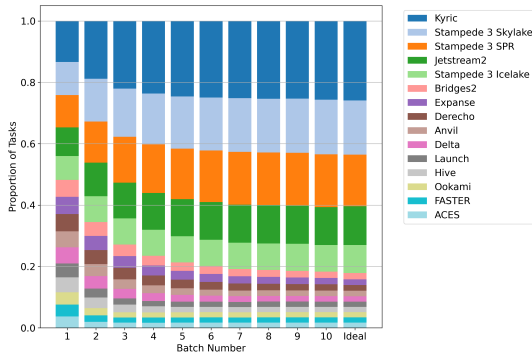


Figure 7.10a: Simulated task distribution across 15 ACCESS resources, demonstrating ATM’s ability to learn near-optimal task distribution over time for multidimensional notions of cost in complex computational environments.

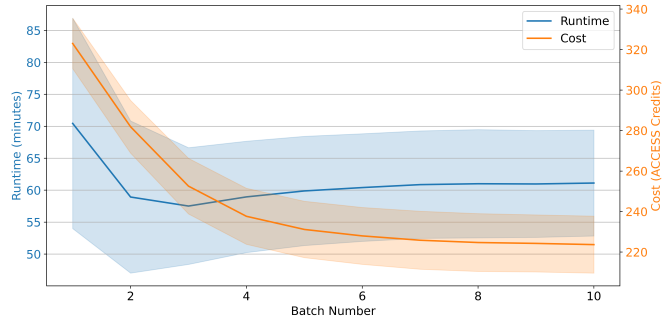


Figure 7.10b: Illustration how runtime and cost are learned by ATM over time in the simulated ACCESS environment. This demonstrates that ATM can significantly reduce cost while still improving execution time in complex computation environments.

In Figure 7.10a, we demonstrate the ability of ATM to quickly and effectively learn the true mapping of performance within the simulated ACCESS ecosystem. The distribution of tasks between the different resources quickly converges to the ideal value of ACCESS credits

per resource performance. By simply trading off between credit cost and time cost, ATM can navigate efficient uses of resource credits while minimizing time. Finally, these results show the ability of ATM to learn these tradeoffs to within 10% of optimal in just 4 batches of tasks and to within 1% in 10 batches.

CHAPTER 8

FUTURE RESEARCH

There are several extensions that follow naturally from the work presented in this thesis. Additionally, there are several that this thesis provides the motivation for that represent more complete research directions. We will take a moment to discuss just a few of these.

8.1 Extensions Addressing Key Limitations

One of the key limitations of this work is the lack of data locality considerations. While we implicitly include data locality, proxied by the round trip time of ATM task timings, we do not explicitly consider data movement. To grossly improve the performance and generalizability of this work, we can extend our methods to serve as a three-level mapping of data to resource to workload rather than just resource to workload. Such consideration will enable us to learn data movement patterns that are more representative of traditional distributed systems workflows. Moreover, data-awareness will enable more accurate resource and workload profiling and characterization as we will be able to decouple the placement strategy from the data movement strategy. Adding an additional level of abstraction will allow us to consider both strategies simultaneously, enabling the three-level graph mapping. The key research challenges here are the creation of a data-locality graph that does not require substantial overheads and the integration of these two mappings into a three-level mapping.

Another extension would be the inclusion of data-awareness with respect to the execution of the workload. Specifically, including the effects specific sizes or types of data have on the execution duration of the workload—as is done in the profiling work and original DELTA framework—could provide substantially more accurate and adaptable cost prediction models, though at the cost of substantial overhead. We anticipate there exists some relationship between data size and contents characterization and data locality characterization and, as such, may allow for reducing these two problems into a single challenge.

8.2 Novel Research Directions

As DELTA exists as a culmination of the individual profiling and cost prediction work, and ATM represents a more unified and generalizable version of DELTA, it is only reasonable to consider what the next step in the evolution of this work is. We believe that the next step likely lies in more robust models of compute, data, and task graphs. The rapid acceleration of development within the AI space has brought about significant advances in graph neural networks. If these models could be leveraged to embed the compute, data, and task graphs into a single abstract space that can be operated on in a unified manner, we could feasibly create a “foundation model” for highly distributed, self-adaptive systems. Additionally, while the allure of self-improving AI systems has captivated the minds of scientists and sci-fi enthusiasts alike for decades, a key capability of creating a foundation model for systems is the ability to use such a system to improve its own underlying model.

Similarly, in this “age of intelligence,” there is a constant push to elicit as much data and information from a system as possible. With the recent advances in AI-based agents, the opportunities for truly distributed intelligence by using the inference capacity of highly disaggregated systems have become more apparent. If you could leverage the system-level task distribution mechanisms of ATM or its eventual successor to distribute agents to available resources based on query or task requirements and the inference capacity of the system, you could create ad hoc systems of agents that sit on the pareto frontier of cost and performance with respect to the resources already available to you.

CHAPTER 9

SUMMARY

In this dissertation, we investigated a wide range approaches to move the field of distributed systems closer to the ideal proposed by the computing continuum. While hardware has largely outpaced software in recent decades, the slowing of Moore’s Law and approach toward physical limits of hardware means we must pursue new methods in order to better use the hardware we have available.

To do this, we proposed novel methods of predicting spot market prices with LSTM neural networks and ARIMA statistical models. Prior to Amazon Web Services’ spot market pricing shift, our LSTM models demonstrated leading performance in predicting spot instance pricing into the future. Following the change to pricing methods, we demonstrated with a study of more than 2 million ARIMA models that most arbitrarily parameterized models could accurately predict the price to within 1%, making price prediction a necessity of the past.

Our modeling capabilities allowed us to predict the price for an arbitrary workload given its runtime on a given resource, but establishing that execution time would require additional work in workload and resource profiling. We developed a novel method of profiling using micro-models that could be independently trained using both historical execution traces and online feedback. Moreover, these models were entirely composable and could be combined with minimal retraining, often requiring only a single additional experiment to reduce prediction error to under 30% across a two-dimensional profiling space (e.g., performance versus data size and CPU cores).

To make this approach generalizable, we extended it with an automated experimental design system. This methodology used a simple simulation to test where in the search space model support would be most increased should an additional observation be included from that point. Using this automated experimental design tool, we showed that we could get within 30% of the ideal error rate with only 30% of the experiments and that 50% of the experiments would yield an error rate within 3% of the ideal.

Using the profiling and prediction methods we developed, we created a framework called DELTA (Distributed Execution of Lambdas with Tradeoff Analysis). This framework acted as a scheduler on top of funcX, a federated function-as-a-service framework. DELTA utilized micropredictors for each machine-task pair and for each stage of task execution. This approach demonstrated significant improvements in task throughput compared to heuristic based scheduling methods. Additionally, this work motivated our hypothesized use case that low-power edge devices can accelerate a multi-device workload even when used in conjunction with much more capable HPC systems via intelligent task placement.

To incorporate dynamically provisioned resources and, accordingly, a wider variety of cost metrics, we built on the original DELTA to create DELTA+. We developed a novel method to curate optimization metrics for DELTA+ by unifying the relation between all relevant cost metrics. Additionally, we introduced a novel approach to automatically generate those optimization metrics via budget setting. Doing so allowed us to integrate dynamic resource provisioning into DELTA+ by simply selecting the resource to be provisioned based on whether it represented a more ideal location on the tradeoff curve created through the budget-based automated metrics.

As DELTA relied on many small models and scheduled tasks individually, scaling was a difficult problem. To this end, we used the lessons we learned from developing DELTA and DELTA+ to create ATM (Adaptive Task Management). ATM’s primary design difference is that it uses historical traces to create a global policy table and makes its scheduling decisions based on all of the tasks and resources at once. This greatly enhances its ability to efficiently distributed large batches of tasks across highly diverse resources. We demonstrated scaling of the client-based framework to more than 16 thousand endpoints simultaneously and ran real science applications across nearly 100 nodes at a time. We were able to demonstrate that ATM’s global scheduling capabilities could accelerate time to science results by increasing overall task throughput and continuously optimizing workload distribution in multi-site

workflows.

Moreover, these capabilities were enhanced by ATM’s ability to incorporate multi-dimensional cost models into its policy generation. We demonstrated this capability by deploying workloads across multiple computing sites using NSF ACCESS credits as an additional cost metric. Our results showed that ATM was able to initially rapidly increase task throughput, then substantially decrease cost over time while largely maintaining throughput. Lastly, we introduced a novel method to dynamically control resource configuration within ATM, allowing the framework to take into account resource contention when deploying potentially conflicting workloads to the same resource. Doing so enabled an additional 6% increase in throughput across our evaluations. In sum, we believe these capabilities make ATM a capable, generalizable tool for coordinating multi-site workflows and motivates substantial future research in this area.

To conclude, the primary contributions of this dissertation are as follows

- An analysis of market dynamics of Amazon Web Services’ spot instances and of how those dynamics change over time;
- Predictive methods to use those markets insights to make better decisions about bidding on terminable cloud instance;
- A profiling approach to incorporate simple models of historical traces and online feedback into more robust composite models, mapping application performance to resource capabilities;
- DELTA and DELTA+—a provisioning framework that using predictive models and tradeoff analysis to inform multi-system scheduling decisions using serverless computing;
- ATM—a novel workflow coordination framework that balances the placement of tasks across all resources using a cost-aware policy table, dynamically updated from online

execution traces.

This work represents substantial progress in the field of distributed systems scheduling, self-adaptive serverless computing, and workflow and resource management. Each of these contributions built iteratively on the previous and each represents a novel innovation, designed to enhance the usability of diverse systems for real science applications. We have deployed these frameworks on resources from single-board computers to TOP500 HPC systems, ensuring the ability to use these tools to accelerate science applications regardless of the resources available to an end user. Moreover, by releasing these frameworks and tools as open source contributions to the community, we intend for them to motivate further innovation in this area and to be built on by all.

Ultimately, this work provides both the conceptual frameworks and the practical tools to move beyond static, siloed computing. By corralling the computing continuum, we transform the compute landscape from one of latent possibility into a tangible and powerful resource, ready to be leveraged for the next generation of scientific discovery.

REFERENCES

- AWS instance types. <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-types.html>. Accessed August 2025.
- Google cloud sizing recommendations. <https://cloud.google.com/compute/docs/instances/apply-sizing-recommendations-for-instances>. Accessed: 2018-08-15.
- Texas advanced computing center. 2023.
- Alberto Abadie, Alexis Diamond, and Jens Hainmueller. Synthetic control methods for comparative case studies: Estimating the effect of california’s tobacco control program. *Journal of the American statistical Association*, 105(490):493–505, 2010.
- Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafir. Deconstructing Amazon EC2 spot instance pricing. *ACM Trans. Econ. Comput.*, 1(3):16:1–16:20, September 2013. ISSN 2167-8375. doi:[10.1145/2509413.2509416](https://doi.org/10.1145/2509413.2509416). URL <http://doi.acm.org/10.1145/2509413.2509416>.
- Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In *USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, 2018.
- Mohammad Imran Ali, Giuseppe Di Modica, Giorgia Rondinini, Andrea Sabbioni, and Antonio Corradi. Function profiling to support smart serverless service deployment in the cloud continuum. In *2024 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6. IEEE, 2024.
- Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. CherryPick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482, Boston, MA, 2017. USENIX Association. ISBN 978-1-931971-37-9. URL <https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/alipourfard>.
- Bryce Allen, John Bresnahan, Lisa Childers, Ian Foster, Gopi Kandaswamy, Raj Kettimuthu, Jack Kordas, Mike Link, Stuart Martin, Karl Pickett, et al. Software as a service for data scientists. *CACM*, 55(2):81–88, 2012.
- Amazon Lambda. Amazon Lambda. aws.amazon.com/lambda/.
- Amazon Web Services. Aws iot greengrass, 2025. URL <https://aws.amazon.com/greengrass/>.
- Amazon Web Services, Inc. Serverless Computing – AWS Lambda. <https://aws.amazon.com/pm/lambda/>, 2025. Accessed: 2025-05-04.

- Rachana Ananthakrishnan, Yadu Babuji, Matt Baughman, Josh Bryan, Kyle Chard, Ryan Chard, Ben Clifford, Ian Foster, Daniel S Katz, Kevin Hunter Kesling, et al. Enabling remote management of faas endpoints with globus compute multi-user endpoints. In *Practice and Experience in Advanced Research Computing 2024: Human Powered Computing*, pages 1–5. 2024.
- David P Anderson. Globally scheduling volunteer computing. *Future Internet*, 13(9):229, 2021.
- Apache OpenWhisk. Apache OpenWhisk. <http://openwhisk.apache.org/>.
- Apache Software Foundation. Apache OpenWhisk: Serverless, open source cloud platform. <https://openwhisk.apache.org/>, 2025. Accessed: 2025-05-04.
- Azure. Azure Functions. <https://azure.microsoft.com/en-us/services/functions/>.
- Y. N. Babuji, K. Chard, A. Gerow, and E. Duede. A secure data enclave and analytics platform for social scientists. In *12th IEEE International Conference on e-Science (e-Science)*, pages 337–342, Oct 2016. doi:[10.1109/eScience.2016.7870917](https://doi.org/10.1109/eScience.2016.7870917).
- Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel Katz, Ben Clifford, Rohan Kumar, Lukasz Lacinski, Ryan Chard, Justin M. Wozniak, Ian Foster, Michael Wilde, and Kyle Chard. Parsl: Pervasive parallel programming in Python. In *28th Intl Symposium on High-Performance Parallel and Distributed Computing*, 2019a. ISBN 9781450366700.
- Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S. Katz, Ben Clifford, Rohan Kumar, Lukasz Lacinski, Ryan Chard, Justin M. Wozniak, Ian Foster, Michael Wilde, and Kyle Chard. Parsl: Pervasive parallel programming in python. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '19, page 25–36, New York, NY, USA, 2019b. Association for Computing Machinery. ISBN 9781450366700. doi:[10.1145/3307681.3325400](https://doi.org/10.1145/3307681.3325400). URL <https://doi.org/10.1145/3307681.3325400>.
- Mohan Baruwat Chhetri, Markus Lumpe, Quoc Bao Vo, and Ryszard Kowalczyk. To bid or not to bid in streamlined EC2 spot markets. In *International Conference on Services Computing*, pages 129–136. IEEE, jul 2018. ISBN 978-1-5386-7250-1.
- André Bauer, Haochen Pan, Ryan Chard, Yadu Babuji, Josh Bryan, Devesh Tiwari, Ian Foster, and Kyle Chard. The globus compute dataset: An open function-as-a-service dataset from the edge to the cloud. *Future Generation Computer Systems*, 153:558–574, 2024.
- Matt Baughman, Ryan Chard, Logan Ward, Jason Pitt, Kyle Chard, and Ian Foster. Profiling and predicting application performance on the cloud. In *11th IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, 2018.

- Matt Baughman, Simon Caton, Christian Haas, Ryan Chard, Rich Wolski, Ian Foster, and Kyle Chard. Deconstructing the 2017 changes to aws spot market pricing. In *Proceedings of the 10th Workshop on Scientific Cloud Computing*, pages 19–26, 2019.
- Christoph Bergmeir, Rob J Hyndman, and Bonsoo Koo. A note on the validity of cross-validation for evaluating autoregressive time series prediction. *Computational Statistics & Data Analysis*, 120:70–83, 2018.
- F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski. The GrADS project: Software support for high-level grid application development. *International Journal of High Performance Computing Applications*, 15(4):327–344, 2001.
- Peter Bodik, Rean Griffith, Charles Sutton, Armando Fox, Michael I Jordan, and David A Patterson. Automatic exploration of datacenter performance regimes. In *1st Workshop on Automated Control for Datacenters and Clouds*, pages 1–6. ACM, 2009.
- Timothy J Boerner, Stephen Deems, Thomas R Furlani, Shelley L Knuth, and John Towns. Access: Advancing innovation: Nsf’s advanced cyberinfrastructure coordination ecosystem: Services & support. In *Practice and Experience in Advanced Research Computing*, pages 173–176. 2023.
- Shawn T Brown, Paola Buitrago, Edward Hanna, Sergiu Sanielevici, Robin Scibek, and Nicholas A Nystrom. Bridges-2: A platform for rapidly-evolving and data intensive research. In *Practice and Experience in Advanced Research Computing 2021: Evolution Across All Dimensions*, pages 1–4. 2021.
- Jun Cai, Wei Liu, Zhongwei Huang, and Fei Richard Yu. Task decomposition and hierarchical scheduling for collaborative cloud-edge-end computing. *IEEE Transactions on Services Computing*, 2024.
- Maria Calzarossa and Giuseppe Serazzi. Workload characterization: A survey. *Proceedings of the IEEE*, 81(8):1136–1150, 1993.
- Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. Wukong: A scalable and locality-enhanced framework for serverless parallel computing. In *11th Symposium on Cloud Computing*, 2020.
- Gabriele Castellano, Stefano Galantino, Fulvio Risso, and Antonio Manzalini. Scheduling multi-component applications across federated edge clusters with phare. *IEEE Open Journal of the Communications Society*, 2024.
- Kyle Chard, Ravi Madduri, Xia Jiang, Farid Dahi, Michael W. Vannier, and Ian Foster. A cloud-based image analysis gateway for traumatic brain injury research. In *9th Gateway Computing Environments Workshop*, pages 13–16. IEEE, 2014a. ISBN 978-1-4799-7030-8. doi:[10.1109/GCE.2014.8](https://doi.org/10.1109/GCE.2014.8).

- Kyle Chard, Steven Tuecke, and Ian Foster. Efficient and secure transfer, synchronization, and sharing of big data. *IEEE Cloud Computing*, 1(3), 2014b.
- R. Chard, K. Chard, K. Bubendorfer, L. Lacinski, R. Madduri, and I. Foster. Cost-aware cloud provisioning. In *11th International Conference on e-Science*, pages 136–144. IEEE, Aug 2015.
- Ryan Chard, Kyle Chard, Kris Bubendorfer, Lukasz Lacinski, Ravi Madduri, and Ian Foster. Cost-aware cloud provisioning. In *11th International Conference on e-Science*, pages 136–144. IEEE, August 2015a. doi:[10.1109/eScience.2015.67](https://doi.org/10.1109/eScience.2015.67).
- Ryan Chard, Kyle Chard, Kris Bubendorfer, Lukasz Lacinski, Ravi Madduri, and Ian Foster. Cost-aware elastic cloud provisioning for scientific workloads. In *8th International Conference on Cloud Computing (CLOUD)*, pages 971–974. IEEE, 2015b.
- Ryan Chard, Kyle Chard, Bryan Ng, Kris Bubendorfer, Alex Rodriguez, Ravi Madduri, and Ian Foster. An automated tool profiling service for the cloud. In *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 223–232. IEEE, 2016a.
- Ryan Chard, Kyle Chard, Bryan Ng, Kris Bubendorfer, Alex Rodriguez, Ravi Madduri, and Ian Foster. An automated tool profiling service for the cloud. In *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 223–232. IEEE, 2016b.
- Ryan Chard, Kyle Chard, Rich Wolski, Ravi Madduri, Bryan Ng, Kris Bubendorfer, and Ian Foster. Cost-aware cloud profiling, prediction, and provisioning as a service. *IEEE Cloud Computing*, 4(4):48–59, 2017.
- Ryan Chard, Ravi Madduri, Nicholas T Karonis, Kyle Chard, Kirk L Duffin, Caesar E Ordoñez, Thomas D Uram, Justin Fleischauer, Ian T Foster, Michael E Papka, et al. Scalable pCT image reconstruction delivered as a cloud service. *IEEE Transactions on Cloud Computing*, 6(1):182–195, 2018.
- Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ian Foster, and Kyle Chard. funcx: A federated function serving fabric for science. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '20, page 65–76, New York, NY, USA, 2020a. Association for Computing Machinery. ISBN 9781450370523. doi:[10.1145/3369583.3392683](https://doi.org/10.1145/3369583.3392683). URL <https://doi.org/10.1145/3369583.3392683>.
- Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ian Foster, and Kyle Chard. funcX: A federated function serving fabric for science. In *High-Performance Parallel and Distributed Computing 2020*, 2020b.
- M. B. Chhetri, M. Lumpe, Q. B. Vo, and R. Kowalczyk. On estimating bids for amazon ec2 spot instances using time series forecasting. In *IEEE International Conference on Services Computing (SCC)*, pages 44–51, June 2017. doi:[10.1109/SCC.2017.14](https://doi.org/10.1109/SCC.2017.14).

- Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. SeBS: A serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference*, pages 64–78, 2021.
- Jerry Danysz, Victor Del Rosal, and Horacio González-Vélez. Aws ec2 spot instances for mission critical services. 2020.
- Christian Davatz, Christian Inzinger, Joel Scheuner, and Philipp Leitner. An approach and case study of cloud instance type selection for multi-tier web applications. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 534–543. IEEE, 2017.
- Christina Delimitrou and Christos Kozyrakis. QoS-aware scheduling in heterogeneous data-centers with paragon. *ACM Transactions on Computer Systems*, 31(4):12, 2013.
- Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and QoS-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.
- Chavit Denninnart, James Gentry, and Mohsen Amini Salehi. Improving robustness of heterogeneous serverless computing systems via probabilistic task pruning. In *Intl Parallel and Distributed Processing Symposium Workshops*, pages 6–15, 2019.
- Anh Vu Do, Junliang Chen, Chen Wang, Young Choon Lee, A.Y. Zomaya, and Bing Bing Zhou. Profiling applications for virtual machine placement in clouds. In *IEEE International Conference on Cloud Computing (CLOUD)*, pages 660–667, July 2011. doi:[10.1109/CLOUD.2011.75](https://doi.org/10.1109/CLOUD.2011.75).
- Timothy Dozat. Incorporating nesterov momentum into adam. 2016.
- Tarek Elgamal, Atul Sandur, Klara Nahrstedt, and Gul Agha. Costless: Optimizing cost of serverless computing through function fusion and placement. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 300–312. IEEE, 2018.
- Firecracker. Firecracker. firecracker-microvm.github.io/. Seen: 10/2020.
- Ian Foster. Coding the continuum. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–1, 2019. doi:[10.1109/IPDPS.2019.00011](https://doi.org/10.1109/IPDPS.2019.00011).
- Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *The International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.
- Enrico Galimberti, Bruno Guindani, Federica Filippini, Hamta Sedghani, Danilo Ardagna, Germán Moltó, and Miguel Caballer. Oscar-p and amllibrary: performance profiling and prediction of computing continua applications. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*, pages 139–146, 2023.

- Wanling Gao, Jianfeng Zhan, Lei Wang, Chunjie Luo, Zhen Jia, Daoyi Zheng, Chen Zheng, Xiwen He, Hainan Ye, Haibin Wang, et al. Data motif-based proxy benchmarks for big data and ai workloads. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 48–58. IEEE, 2018.
- GCP Functions. GCP Functions. <https://cloud.google.com/functions/>.
- Gareth George, Rich Wolski, Chandra Krintz, and John Brevik. Analyzing AWS spot instance pricing. In *35th IEEE International Conference on Data Engineering*, 2019.
- Taiyang Guo, Rami Bahsoon, Tao Chen, Abdessalam Elhabbash, Faiza Samreen, and Yehia Elkhatib. Cloud instance selection using parallel k-means and ahp. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, pages 71–76, 2019.
- Tian Guo, Upendra Sharma, Prashant Shenoy, Timothy Wood, and Sambit Sahu. Cost-aware cloud bursting for enterprise applications. *ACM Transactions on Internet Technology (TOIT)*, 13(3):1–24, 2014.
- Weichao Guo, Kang Chen, Yongwei Wu, and Weimin Zheng. Bidding for highly available services with low price in spot instance market. In *24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 191–202. ACM, 2015.
- David Y Hancock, Jeremy Fischer, John Michael Lowe, Winona Snapp-Childs, Marlon Pierce, Suresh Marru, J Eric Coulter, Matthew Vaughn, Brian Beck, Nirav Merchant, et al. Jetstream2: Accelerating cloud computing via jetstream. In *Practice and Experience in Advanced Research Computing*, pages 1–8. ACM, 2021.
- Ali Jassim Hasan and Mustafa Hammad. Spot hopping: Increasing reliability and reducing cost. *International Journal of Computing and Digital Systems*, 9(6):1237–1250, 2020.
- Mihael Hategan, Justin Wozniak, and Ketan Maheshwari. Coasters: uniform resource provisioning and access for clouds and grids. In *2011 Fourth IEEE International Conference on Utility and Cloud Computing*, pages 114–121. IEEE, 2011.
- Xin He, Prashant Shenoy, Ramesh Sitaraman, and David Irwin. Cutting the cost of hosting online services using cloud spot markets. In *24th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2015.
- Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Serverless computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing*, 2016.
- Samuel Hsia, Udit Gupta, Mark Wilkening, Carole-Jean Wu, Gu-Yeon Wei, and David Brooks. Cross-stack workload characterization of deep recommendation systems. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pages 157–168. IEEE, 2020.

- C. Hsu, V. Nair, V. W. Freeh, and T. Menzies. Arrow: Low-level augmented bayesian optimization for finding the best cloud vm. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, volume 00, pages 660–670, Jul 2018. doi:[10.1109/ICDCS.2018.00070](https://doi.org/10.1109/ICDCS.2018.00070). URL doi.ieeecomputersociety.org/10.1109/ICDCS.2018.00070.
- Nathaniel Hudson, Hana Khamfroush, Matt Baughman, Daniel E Lucani, Kyle Chard, and Ian Foster. Qos-aware edge ai placement and scheduling with multiple implementations in faas-based edge computing. *Future Generation Computer Systems*, 157:250–263, 2024.
- B. Javadi, R. K. Thulasiramy, and R. Buyya. Statistical modeling of spot instance prices in public cloud environments. In *4th IEEE International Conference on Utility and Cloud Computing (UCC)*, pages 219–228, Dec 2011. doi:[10.1109/UCC.2011.37](https://doi.org/10.1109/UCC.2011.37).
- Bahman Javadi, Ruppia K. Thulasiram, and Rajkumar Buyya. Characterizing spot price dynamics in public cloud environments. *Future Gener. Comput. Syst.*, 29(4):988–999, June 2013. ISSN 0167-739X. doi:[10.1016/j.future.2012.06.012](https://doi.org/10.1016/j.future.2012.06.012). URL <http://dx.doi.org/10.1016/j.future.2012.06.012>.
- Fredy Juarez, Jorge Ejarque, and Rosa M Badia. Dynamic energy-aware scheduling for parallel task-based application in cloud computing. *Future Generation Computer Systems*, 78:257–271, 2018.
- Gideon Juve, Ann Chervenak, Ewa Deelman, Shishir Bharathi, Gaurang Mehta, and Karan Vahi. Characterizing and profiling scientific workflows. *Future Generation Computer Systems*, 29(3):682–692, 2013. ISSN 0167-739X. doi:<https://doi.org/10.1016/j.future.2012.08.015>. URL <https://www.sciencedirect.com/science/article/pii/S0167739X12001732>. Special Section: Recent Developments in High Performance Computing and Security.
- Alok Kamatar, Valerie Hayot-Sasson, Yadu Babuji, Andre Bauer, Gourav Rattihalli, Ninad Hogade, Dejan Milojicic, Kyle Chard, and Ian Foster. Greenfaas: Maximizing energy efficiency of hpc workloads with faas. *arXiv preprint arXiv:2406.17710*, 2024.
- Robert Keller, Lukas Häfner, Thomas Sachs, and Gilbert Fridgen. Scheduling flexible demand in cloud computing spot markets. *Business & Information Systems Engineering*, 62(1):25–39, 2020.
- Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- Kubeless. Kubeless. <https://kubeless.io/>.
- Rohan Kumar, Matt Baughman, Ryan Chard, Zhuozhao Li, Yadu Babuji, Ian Foster, and Kyle Chard. Coding the computing continuum: Fluid function execution in heterogeneous computing environments. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 66–75. IEEE, 2021a.

- Rohan Kumar, Matthew Baughman, Ryan Chard, Zhuozhao Li, Yadu Babuji, Ian Foster, and Kyle Chard. Coding the computing continuum: Fluid function execution in heterogeneous computing environments. In *Heterogeneity in Computing Workshop 2021*, 2021b.
- Dac-Nhuong Le, Souvik Pal, and Prasant Kumar Pattnaik. Openfaas. *Cloud computing solutions: architecture, data storage, implementation and security*, pages 287–303, 2022.
- Changyuan Lin and Hamzeh Khazaei. Modeling and optimization of performance and cost of serverless applications. *IEEE Transactions on Parallel and Distributed Systems*, 32(3): 615–632, 2020.
- Wenqiang Liu, Pengwei Wang, Ying Meng, Qin Zhao, Caihui Zhao, and Zhaohui Zhang. A novel model for optimizing selection of cloud instance types. *IEEE Access*, 7:120508–120521, 2019a.
- Wenqiang Liu, Pengwei Wang, Ying Meng, Guobing Zou, and Zhaohui Zhang. A novel algorithm for optimizing selection of cloud instance types in multi-cloud environment. In *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 167–170. IEEE, 2019b.
- Markus Lumpe, Mohan Baruwat Chhetri, Quoc Bao Vo, and Ryszard Kowalczyk. On estimating minimum bids for Amazon EC2 spot instances. In *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 391–400. IEEE, may 2017. ISBN 978-1-5090-6611-7.
- Shu Luo, Linxuan Zhang, and Yushun Fan. Dynamic multi-objective scheduling for flexible job shop by deep reinforcement learning. *Computers & Industrial Engineering*, 159:107489, 2021.
- Ravi Madduri, Kyle Chard, Ryan Chard, Lukasz Lacinski, Alex Rodriguez, Dinanath Sulakhe, David Kelly, Utpal Dave, and Ian Foster. The Globus Galaxies platform: Delivering science gateways as a service. *Concurrency and Computation: Practice and Experience*, 27(16):4344–4360, 2015.
- Ravi K Madduri, Dinanath Sulakhe, Lukasz Lacinski, Bo Liu, Alex Rodriguez, Kyle Chard, Utpal J Dave, and Ian T Foster. Experiences building Globus Genomics: a next-generation sequencing analysis service using Galaxy, Globus, and Amazon Web Services. *Concurrency and Computation: Practice and Experience*, 26(13):2266–2279, 2014.
- Pattie Maes, Maja J. Mataric, Jean-Arcady Meyer, Jordan Pollack, and Stewart W. Wilson. Explore/exploit strategies in autonomy. In *4th Intl Conf. on Simulation of Adaptive Behavior*, volume 4, pages 325–332, 1996.
- Muthucumaran Maheswaran, Shoukat Ali, Howard Jay Siegel, Debra Hensgen, and Richard F Freund. Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. *Journal of parallel and distributed computing*, 59(2):107–131, 1999.

- Alexandre Maros, Fabricio Murai, Ana Paula Couto da Silva, Jussara M Almeida, Marco Lattuada, Eugenio Gianniti, Marjan Hosseini, and Danilo Ardagna. Machine learning for performance prediction of spark cloud applications. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 99–106. IEEE, 2019.
- Andréa Matsunaga and José AB Fortes. On the use of machine learning to predict the time and resources consumed by applications. In *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 495–504. IEEE Computer Society, 2010.
- Gabriele Proietti Mattia and Roberto Beraldi. Online decentralized scheduling in fog computing for smart cities based on reinforcement learning. *IEEE Transactions on Cognitive Communications and Networking*, 2024.
- Narges Mehran, Dragi Kimovski, Hermann Hellwagner, Dumitru Roman, Ahmet Soylu, and Radu Prodan. Scheduling of distributed applications on the computing continuum: A survey. In *Proceedings of the IEEE/ACM 16th International Conference on Utility and Cloud Computing*, pages 1–6, 2023.
- Harshitha Menon, Abhinav Bhatele, and Todd Gamblin. Auto-tuning parameter choices in hpc applications using bayesian optimization. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 831–840. IEEE, 2020.
- Vinicius Meyer, Dionatra F Kirchoff, Matheus L Da Silva, and Cesar AF De Rose. ML-driven classification scheme for dynamic interference-aware resource scheduling in cloud infrastructures. *Journal of Systems Architecture*, 116:102064, 2021.
- Takashi Miyazaki, Issei Sato, and Nobuyuki Shimizu. Bayesian optimization of hpc systems for energy efficiency. In *High Performance Computing: 33rd International Conference, ISC High Performance 2018, Frankfurt, Germany, June 24-28, 2018, Proceedings 33*, pages 44–62. Springer, 2018.
- Raffaele Montella, David Kelly, Wei Xiong, Alison Brizius, Joshua Elliott, Ravi Madduri, Ketan Maheshwari, Cheryl Porter, Peter Vilter, Michael Wilde, Meng Zhang, and Ian Foster. FACE-IT: A science gateway for food security research. *Concurrency and Computation: Practice and Experience*, 27(16):4423–4436, 2015. ISSN 1532-0634.
- Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. {Heterogeneity-Aware} cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498, 2020.
- Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *USENIX Annual Technical Conference*, 2018.
- University of California San Diego. San diego supercomputing center (2025): Expanse. 2025. doi:[10.1145/3437359.3465588](https://doi.org/10.1145/3437359.3465588).

- Panos Patros, Melanie Ooi, Victoria Huang, Michael Mayo, Chris Anderson, Stephen Burroughs, Matt Baughman, Osama Almurshed, Omer Rana, Ryan Chard, et al. Rural AI: Serverless-powered federated learning for remote applications. *IEEE Internet Computing*, 27(2):28–34, 2022.
- J. Gregory Pauloski, Valerie Hayot-Sasson, Logan Ward, Nathaniel Hudson, Charlie Sabino, Matt Baughman, Kyle Chard, and Ian Foster. Accelerating communications in federated applications with transparent object proxies. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701092. doi:[10.1145/3581784.3607047](https://doi.org/10.1145/3581784.3607047). URL <https://doi.org/10.1145/3581784.3607047>.
- J Gregory Pauloski, Valerie Hayot-Sasson, Maxime Gonthier, Nathaniel Hudson, Haochen Pan, Sicheng Zhou, Ian Foster, and Kyle Chard. TaPS: A performance evaluation suite for task-based execution frameworks. In *2024 IEEE 20th International Conference on e-Science (e-Science)*, pages 1–10. IEEE, 2024.
- Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. Low latency geo-distributed data analytics. *ACM SIGCOMM Computer Communication Review*, 45(4):421–434, 2015.
- Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021.
- Raghunathan Ramakrishnan, Pavlo O Dral, Matthias Rupp, and O Anatole von Lilienfeld. Quantum chemistry structures and properties of 134 kilo molecules. *Scientific Data*, 1, 2014.
- Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE micro*, 30(4):65–79, 2010.
- Albert Reuther, Chansup Byun, William Arcand, David Bestor, Bill Bergeron, Matthew Hubbell, Michael Jones, Peter Michaleas, Andrew Prout, Antonio Rosa, et al. Scalable system scheduling for hpc and big data. *Journal of Parallel and Distributed Computing*, 111:76–92, 2018.
- Sophie Robert, Soraya Zertal, and Philippe Couvee. Shaman: A flexible framework for auto-tuning hpc systems. In *Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 147–158. Springer, 2020.
- T. M. Sands, D. Tayal, M. E. Morris, and S. T. Monteiro. Robust stock value prediction using support vector machines with particle swarm optimization. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 3327–3331, May 2015. doi:[10.1109/CEC.2015.7257306](https://doi.org/10.1109/CEC.2015.7257306).

- J. Scheuner and P. Leitner. Estimating cloud application performance based on micro-benchmark profiling. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 90–97, July 2018. doi:[10.1109/CLOUD.2018.00019](https://doi.org/10.1109/CLOUD.2018.00019).
- Burr Settles. Active learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 6(1):1–114, 2012.
- Prateek Sharma, Stephen Lee, Tian Guo, David Irwin, and Prashant Shenoy. Spotcheck: Designing a derivative iaas cloud on the spot market. In *10th European Conference on Computer Systems*, pages 16:1–16:15. ACM, 2015.
- Supreeth Shastri and David Irwin. Hotspot: automated server hopping in cloud spot markets. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 493–505, 2017.
- A. Sheta, S. Ahmed, and H. Faris. A comparison between regression, artificial neural networks and support vector machines for predicting stock market index. *International Journal of Advanced Research in Artificial Intelligence*, 4(7):55–63, 2015.
- Simon Shillaker and Peter Pietzuch. Faasm: lightweight isolation for efficient stateful serverless computing. In *USENIX Annual Technical Conference (USENIX ATC 20)*, 2020.
- Barry Sly-Delgado, Thanh Son Phung, Colin Thomas, David Simonetti, Andrew Hennessee, Ben Tovar, and Douglas Thain. Taskvine: Managing in-cluster storage for high-throughput data intensive workflows. In *Proceedings of the SC’23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, pages 1978–1988, 2023.
- W. Smith, I. Foster, and V. Taylor. Predicting application run times with historical information. *Journal of Parallel and Distributed Computing*, 64(9):1007–1016, 2004a.
- Warren Smith, Ian Foster, and Valerie Taylor. Predicting application run times with historical information. *J. Parallel & Distributed Computing*, 64(9), 2004b.
- X Carol Song, Preston Smith, Rajesh Kalyanam, Xiao Zhu, Eric Adams, Kevin Colby, Patrick Finnegan, Erik Gough, Elizabett Hillery, Rick Irvine, et al. Anvil-system architecture and experiences from deployment and early user operations. In *Practice and experience in advanced research computing*, pages 1–9. ACM, 2022.
- Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *arXiv preprint arXiv:2001.04592*, 2020.
- Supreeth Subramanya, Tian Guo, Prateek Sharma, David Irwin, and Prashant Shenoy. SpotOn: A batch computing service for the spot market. In *6th ACM Symposium on Cloud Computing*, pages 329–341. ACM Press, 2015. ISBN 9781450336512.
- Amoghvarsha Suresh and Anshul Gandhi. FnSched: An efficient scheduler for serverless functions. In *5th Intl Workshop on Serverless Computing*, pages 19–24, 2019.

- S. Tang, J. Yuan, and X. Y. Li. Towards optimal bidding strategy for Amazon EC2 cloud spot instance. In *IEEE 5th International Conference on Cloud Computing (CLOUD)*, pages 91–98, June 2012. doi:[10.1109/CLOUD.2012.134](https://doi.org/10.1109/CLOUD.2012.134).
- Shujiong Tang, Yue Yu, Hui Wang, Guiliang Wang, Wuhui Chen, Zenglin Xu, Song Guo, and Wen Gao. A survey on scheduling techniques in computing and network convergence. *IEEE Communications Surveys & Tutorials*, 26(1):160–195, 2023.
- Wei Tang, Narayan Desai, Daniel Buettner, and Zhiling Lan. Job scheduling with adjusted runtime estimates on production supercomputers. *Journal of Parallel and Distributed Computing*, 73(7):926–938, 2013.
- Jonathan L. Ticknor. A bayesian regularized artificial neural network for stock market forecasting. *Expert Systems with Applications*, 40(14):5501–5506, 2013. ISSN 0957-4174. doi:<https://doi.org/10.1016/j.eswa.2013.04.013>. URL <http://www.sciencedirect.com/science/article/pii/S0957417413002509>.
- Steven Tuecke, Rachana Ananthakrishnan, Kyle Chard, Mattias Lidman, Brendan McColam, Stephen Rosen, and Ian Foster. Globus auth: A research identity and access management platform. In *IEEE 12th International Conference on e-Science (e-Science)*, pages 203–212, 2016. doi:[10.1109/eScience.2016.7870901](https://doi.org/10.1109/eScience.2016.7870901).
- Volodymyr Turchenko, Vladyslav Shults, Iryna Turchenko, Richard Wallace, Mehdi Sheikhalishahi, Jose Vazquez-Poletti, and Lucio Grandinetti. Spot price prediction for cloud computing using neural networks. *International Journal of Computing*, 12(4):348–359, 2014.
- Vidushi Vashishth, Anshuman Chhabra, and Apoorvi Sood. A predictive approach to task scheduling for big data in cloud environments using classification algorithms. In *2017 7th International Conference on Cloud Computing, Data Science & Engineering-Confluence*, pages 188–192. IEEE, 2017.
- Shivaram Venkataraman, Zongheng Yang, Michael J Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *NSDI*, pages 363–378, 2016.
- Vincent A Voelz, Vijay S Pande, and Gregory R Bowman. Folding@ home: Achievements from over 20 years of citizen science herald the exascale era. *Biophysical journal*, 122(14):2852–2863, 2023.
- William Voorsluys and Rajkumar Buyya. Reliable provisioning of spot instances for compute-intensive applications. In *26th International Conference on Advanced Information Networking and Applications*, pages 542–549. IEEE, mar 2012. ISBN 978-1-4673-0714-7.
- R. M. Wallace, V. Turchenko, M. Sheikhalishahi, I. Turchenko, V. Shults, J. L. Vazquez-Poletti, and L. Grandinetti. Applications of neural-based spot market prediction for

- cloud computing. In *7th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS)*, volume 02, pages 710–716, Sept 2013. doi:[10.1109/IDAACS.2013.6663017](https://doi.org/10.1109/IDAACS.2013.6663017).
- Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *USENIX Annual Technical Conference*, pages 133–146, 2018.
- Michael Wilde, Mihael Hategan, Justin M Wozniak, Ben Clifford, Daniel S Katz, and Ian Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9): 633–652, 2011.
- Rich Wolski, John Brevik, Ryan Chard, and Kyle Chard. Probabilistic guarantees of execution duration for amazon spot instances. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '17*, pages 18:1–18:11, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5114-0. doi:[10.1145/3126908.3126953](https://doi.org/10.1145/3126908.3126953). URL <http://doi.acm.org/10.1145/3126908.3126953>.
- Walter Wong, Aleksandr Zavodovski, Lorenzo Corneo, Nitinder Mohan, and Jussi Kangasharju. Spa: Harnessing availability in the aws spot market. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 1–6. IEEE, 2021.
- Timothy Wood, Ludmila Cherkasova, Kivanc Ozonat, and Prashant Shenoy. Profiling and modeling resource usage of virtualized applications. In *9th ACM/IFIP/USENIX International Conference on Middleware*, pages 366–387, 2008.
- Chaofeng Wu, Ted Summer, Zhuozhao Li, Anna Woodard, Ryan Chard, Matt Baughman, Yadu Babuji, Kyle Chard, Jason Pitt, and Ian Foster. Paraopt: Automated application parameterization and optimization for the cloud. In *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 255–262. IEEE, 2019.
- Yujian Wu, Shanjiang Tang, Ce Yu, Bin Yang, Chao Sun, Jian Xiao, and Hutong Wu. Task scheduling in geo-distributed computing: A survey. *arXiv preprint arXiv:2501.15504*, 2025.
- Fatos Xhafa and Ajith Abraham. Computational models and heuristic methods for grid scheduling problems. *Future generation computer systems*, 26(4):608–621, 2010.
- Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, 2018.
- Hong Xu and Baochun Li. Dynamic cloud pricing for revenue maximization. *IEEE Transactions on Cloud Computing*, 1(2):158–171, 2013. ISSN 2168-7161.

- Neeraja J Yadwadkar, Bharath Hariharan, Joseph E Gonzalez, Burton Smith, and Randy H Katz. Selecting the best VM across multiple public clouds: A data-driven performance modeling approach. In *Symposium on Cloud Computing*, pages 452–465. ACM, 2017.
- Jiadong Yang, Hua Xu, Li Pan, Peifa Jia, Fei Long, and Ming Jie. Task scheduling using bayesian optimization algorithm for heterogeneous computing environments. *Applied Soft Computing*, 11(4):3297–3310, 2011.
- Fan Zhang, Junwei Cao, Keqin Li, Samee U Khan, and Kai Hwang. Multi-objective scheduling of many tasks in cloud platforms. *Future generation computer systems*, 37:309–320, 2014.
- Liang Zheng, Carlee Joe-Wong, Chee Wei Tan, Mung Chiang, Xinyu Wang, Liang Zheng, Carlee Joe-Wong, Chee Wei Tan, Mung Chiang, and Xinyu Wang. How to bid the cloud. In *SIGCOMM*, volume 45, pages 71–84. ACM Press, 2015. ISBN 9781450335423.
- Xinran Zhu, Yang Liu, Pieter Ghysels, David Bindel, and Xiaoye S Li. Gptuneband: Multi-task and multi-fidelity autotuning for large-scale high performance computing applications. In *Proceedings of the 2022 SIAM Conference on Parallel Processing for Scientific Computing*, pages 1–13. SIAM, 2022.