Addressing Serverless Computing Vendor Lock-In through Cloud Service Abstraction

Di Mo, Robert Cordingly, Donald Chinn, Wes Lloyd
School of Engineering and Technology
University of Washington
Tacoma, Washington USA
dimo, rcording, dchinn, wlloyd@uw.edu

***Abstract*—Serverless Function-as-a-Service (FaaS) platforms enable easy deployment and hosting of microservices and have gained great traction among software developers. FaaS platforms, however, only host compute-based functionality of applications resulting in vendor lock-in as applications rely on supporting services known as Backend-as-a-Service (BaaS) offered by the cloud provider for key features such as data persistence. Migrating FaaS code to different cloud providers is made more challenging as a result of these dependencies on vendor-specific services. Cloud service abstraction libraries have been developed to alleviate vendor lock-in, but these libraries were largely developed prior to the advent of serverless computing and have not been evaluated in this context. This paper investigates the use of cloud service abstraction libraries to interface with object storage, a key BaaS used in FaaS code. We investigate the utility of these libraries to improve the portability of code to enable easier migration between cloud providers. We investigate performance of seven FaaS functions on AWS and Google Cloud that use object storage using the Apache jclouds abstraction library vs. platform-specific APIs and assess code quality metrics. We then conduct an empirical study leveraging computer science students enrolled in cloud computing courses to assess the impact of cloud abstraction libraries on FaaS function code portability.**

***Index Terms*—Serverless Computing, FaaS, Vendor Lock-In, Multi-Cloud, Cloud Portability, Cloud Interoperability**

I. Introduction

Serverless computing is a cloud computing paradigm that simplifies cloud application hosting by eliminating management and maintenance of servers. Serverless applications designed using Function-as-a-Service (FaaS) functions leverage distinct cloud services to decouple computation from storage and required backend services. This approach provides a pay-as-you-go billing model that can be more cost-effective than other forms of cloud hosting, especially for services requiring high availability with low-to-moderate utilization. Developers are drawn to the ease of provisioning, allowing them to leverage distributed parallelism without the burden of infrastructure management. However, interoperability, a key design goal of distributed systems, is violated resulting in "vendor lock-in" when developers transfer FaaS functions to different cloud platforms [1].

Vendor lock-in occurs when cloud providers offer custom Backend-as-a-Service (BaaS) offerings that require developers to write code specifically for their platform. This creates dependencies on vendor-specific interfaces to BaaS offerings, hindering code migration to other clouds without significant refactoring effort [2]. Developing FaaS applications that are compatible with multiple clouds requires maintaining multiple code versions, resulting in inconsistencies in supported features and increased maintenance burden. FaaS platforms can exacerbate vendor lock-in due to their limited resources and runtime environments, as developers rely on vendor-specific BaaS offerings for core application functionality.

To address challenges imposed by cloud vendor lock-in, prior efforts produced cloud service abstraction libraries [3]–[8]. These libraries provide abstract APIs to access common cloud services enabling developers to create more portable code to ease migration to different clouds. However, their utility in the context of FaaS functions has not been assessed.

This paper investigates implications of adopting cloud abstraction libraries to interface with object storage services in FaaS code. We investigate implications for (1) performance, (2) code quality, and (3) portability. We compare performance of seven FaaS functions using native vs. abstraction libraries to access object storage services. We evaluate code quality using a static analysis tool to assess code quality metrics. To investigate portability, we conducted an empirical study on FaaS code migration leveraging 42 undergraduate senior and graduate computer science students from cloud computing classes as software developers. Students were split into two groups to migrate code from AWS to Google Cloud. One group used an abstraction library to access object storage, while the other used cloud native APIs enabling comparison of developer effort and migration task outcomes.

A. Research Questions

This paper investigates the following research questions:

RQ-1: (Abstraction Overhead) What are the performance implications of using cloud service abstraction libraries to interface with object storage services in FaaS code?

RQ-2: (Code Quality) How does the adoption of cloud service abstraction libraries impact FaaS code quality measured using static code analysis metrics?

RQ-3: (Portability) How does the adoption of cloud service abstraction libraries impact the portability of FaaS code when migrating functions from one cloud provider to another? What factors are related to successful code migration outcomes?

B. Contributions

This paper makes the following research contributions:

1. Object store performance analysis of FaaS functions using native vs. cloud abstraction libraries.
2. Static code analysis of FaaS functions using native vs. cloud abstraction libraries to assess code quality.
3. An empirical study to assess utility of cloud abstraction libraries to support FaaS code migration between cloud providers. Feature analysis is performed to identify important features for predicting migration task outcomes.
4. Creation of tutorials to train developers on the use of Apache jclouds and native cloud libraries to migrate FaaS code using object storage between cloud providers.

II. Background and Related Work

A. Serverless Computing Vendor Lock-in

Vendor lock-in problems arise broadly for cloud computing code migration when developers are required to adapt software to use equivalent supporting services that differ from provider to provider. Migration is a key challenge in serverless computing as the simplicity of FaaS platforms can increase reliance on supporting cloud services whose implementation varies across providers [2], [9]. For example, AWS offers object storage known as the Simple Storage Service (S3), while Microsoft provides Azure Blob Storage [10], [11]. These storage services have distinct APIs for interfacing with FaaS code. To migrate FaaS code dependent on vendor specific cloud services from AWS to Azure requires developers to familiarize themselves with the divergent characteristics of service APIs.

In [12], four use cases (thumbnail generation, serverless create-eead-update-delete API, event processing, and function orchestration) were used to demonstrate difficulties migrating code between FaaS platforms. The authors identified different categories of vendor lock-in: compatibility of an application component, tooling compatibility, and architecture compatibility. In this paper, we focus on the compatibility of application components vendor lock-in problem. Yussupov et al. noted that fulfillment of provider-specific implementation requirements can introduce multiple types of dependencies that may impact an application's portability. Our investigation intersects with the library and service interaction dependency examples described by Yussupov et al.

B. Cloud abstraction libraries

To help mitigate negative effects of cloud vendor lock-in, one solution is to leverage cloud service abstraction libraries when accessing popular services. These libraries typically support interfacing with common services across multiple cloud providers. Existing cloud abstraction libraries with programming interfaces for object storage include: Apache jClouds, Dasein Cloud, Apache Libcloud, and Cloudbridge [3]–[5], [13]. Apache jclouds, Cloudbridge, and Apache libcloud are examples of multi-cloud APIs. Apache jclouds is a Java-based cloud abstraction library that supports widely used commercial clouds that are open source with recent project updates on GitHub. Conversely, DaseIn, another Java-based cloud abstraction library, has not been updated since 2014. For Python, Apache Libcloud and Cloudbridge are cloud abstraction library projects with recent updates on GitHub. Apache Libcloud supports many cloud platforms, whereas Cloudbridge only supports Google Cloud and Microsoft Azure.

Agarwal first evaluated abstraction of IaaS cloud services using Dasein Cloud and jclouds for AWS EC2 and Google Compute Engine [14]. Agarwal highlighted the usability of jclouds, noting jclouds had well-documented webpages and better community support in contrast to Dasein. Several studies compared the performance of abstraction libraries compared to vendor-specific APIs for accessing various cloud services. In [15], performance was evaluated for request latency and throughput. Findings indicated that jclouds had inferior performance compared to platform-specific APIs. In [16], the performance of Apache jclouds and Apache Libcloud was compared with vendor-specific APIs on AWS and Azure. Results revealed that jclouds exhibited subpar performance compared to platform-specific APIs, whereas Libcloud had superior performance in most cases.

III. Methodology

In this research, we investigate the utility of abstraction libraries to remove dependencies to common application building blocks commonly accessed using vendor specific libraries with the goal of creating portable FaaS code. Our research focused on conducting an empirical study with computer science students as developers to investigate the utility of abstraction libraries to support FaaS code migration. To ensure statistical power with a limited pool of volunteer participants who had finite time to dedicate to the study, we restricted the scope of our experimental design in the following ways: (1) only use of object storage services were abstracted (e.g. no other services), (2) use of the Java-based Apache jclouds abstraction library to support code migration because participants were most familiar with Java, and (3) migration from AWS to Google Cloud.

In Fall 2022, students from two cloud computing courses were surveyed about their programming experience. Students self-reported having 2.62 years average experience with Java, but only 1.15 years with Python. **We designed our study using Java because students reported more than double the experience with Java.** To select a cloud abstraction library, we considered actively maintained libraries based on GitHub repository commits that supported Java, AWS, and Google Cloud and selected Apache jclouds [3].

We hope our study catalyzes interest and future research regarding abstraction libraries to help in creating portable FaaS code for other languages, cloud platforms, and FaaS application building blocks. We note that some cloud providers support a subset of the AWS S3 API as a means to encourage migration to their cloud, creating a quasi standard object storage interface. This support, however, only enables migration from AWS S3. We conduct our study on object storage interface abstraction as a proxy to explore potential for abstraction libraries to improve code portability when there is vendor lock-in to any application building block.

We divided our investigation into two experiments. Experiment I investigated refactoring FaaS functions to create platform agnostic FaaS code using Apache jclouds to access object storage on AWS and Google Cloud. Experiment I.a analyzed performance implications **(RQ-1)**, and Experiment I.b assessed code quality metrics **(RQ-2)**. Experiment II investigated the utility of cloud service abstraction libraries to support FaaS code migration between cloud providers **(RQ-3)**.

A. Experiment I.a: Abstraction Library FaaS Performance

For Experiment I.a we refactored seven FaaS functions coded in Java described in Table III to use jclouds to access object storage. We compared performance to the original code that used cloud native APIs. Functions were configured with 2GB of memory and a maximum runtime of 9 minutes on AWS and GCP. 2GB memory across both clouds ensured the execution environments had access to at least one virtual CPU core with 100% timeshare [17]. We evaluated *runtime* in seconds, and *data read throughput* between object storage and the FaaS platform in megabytes per second.

B. Experiment I.b: Abstraction Library FaaS Code Quality

Experiment I.b investigated code quality implications of adopting cloud abstraction libraries. We compared FaaS code versions using Apache jclouds vs. vendor-specific APIs with JArchitect, a static analysis tool, [18]. For the Read_File FaaS function we evaluated the metrics: *JAR file size* in megabytes, the total number of project *source_files*, *third-party elements* the number of non-native Java elements (e.g. class references, method usages, etc.), *LOC* lines of FaaS code, *LOCR* lines of refactored FaaS code including lines added, removed, and modified, and the *average cyclomatic complexity (CC)* computed at the method-level equal to the number of decisions that can be taken in a method.

C. Experiment II: Code Portability Empirical Study

To assess FaaS code portability, we conducted an empirical study involving undergraduate senior and graduate students from cloud computing courses as software developers. Students were recruited from an undergraduate and a graduate cloud computing class in Fall 2022. In the classes, students were introduced to cloud services and APIs commonly used in FaaS applications including object storage, relational databases, logging, workflow orchestration, event processing, and message queues. 42 of 58 students in the classes volunteered to participate in the study. The study was organized as an in-class activity where all participants received full credit regardless of the outcome of the FaaS code migration.

We divided the 42 participants into groups of 20 and 22 students respectively for Group-GCP and Group-Jclouds. In Group-GCP, participants were asked to migrate a FaaS function from AWS (using Amazon S3 APIs) to Google (Google Cloud Storage APIs). For Group-Jclouds, participants migrated a FaaS function from AWS that used the Apache jclouds abstraction library to access Amazon S3 to Google Cloud. To address validation issues, we took steps to minimize threats to validity. In forming the groups, we balanced the number of participants in Group-GCP vs. Group-Jclouds. We also balanced the number of participants on-site versus online to a ratio of ∼1:2. In fall 2022, online participation was popular potentially due to participant interest in isolating due to COVID-19. Finally, we balanced the ratio of undergraduate vs. graduate students between the groups. Table I describes the participant distributions for Group-GCP and Group-Jclouds. For factors such as Java programming experience or participant age that were not balanced, we later analyzed the impact of these factors on task success in the post-experiment analysis.

TABLE I: Participation Overview

Group	Online/Onsite	Section	Number of Participants		
GCP	Online	undergraduate	9		
		graduate	5	14	
	Onsite	undergraduate	2		
		graduate	4	6	20
Jclouds	Online	undergraduate	9		
		graduate	6	15	
	Onsite	undergraduate	3		
		graduate	4	7	22

The empirical study was organized as an extended "mini-Hackathon" class session where participants had up to four hours to complete the code migration training and activity with the investigators present for help. The additional time allowed the training and migration activity to be completed in the same session to maximize retention of concepts. To maximize participation in the study, students were given the option to participate in person or online. One investigator hosted an active Zoom session with video and chat to provide real-time support for remote participants, while the other investigator focused on helping on-site participants.

Before the code migration task, participants completed an interactive tutorial describing how to migrate AWS Lambda functions to Google Cloud with the assigned object storage service interface (i.e. native cloud APIs or Apache jclouds). The tutorial guided participants to migrate a function that reads and writes to object storage providing a code refactoring experience and access to working example source code that could be referenced later for the code migration activity.

For the code migration task, participants migrated a Java-based FaaS function that resized graphical images stored using object storage from AWS Lambda to Google Cloud Functions. Unlike the tutorial, participants were not provided step-by-step instructions. The function read an original image and produced and wrote a resized image back to object storage - an image processing use case. Code migration required the function package and base libraries be adapted for deployment to the destination cloud. In addition, use of Amazon S3 object storage had to be adapted to Google Cloud Storage, the equivalent object storage service on Google Cloud. Migration success was determined by verifying that resized images were successfully written to Google Cloud Storage. To evaluate complexity of the tutorial and migration task, we produced solutions and evaluated the required number of parameter changes, LOC changes, and method changes as shown in Table II. These numbers represent the minimum number of required

TABLE II: Required Source Code Changes for Tasks

	Group-GCP	Group-Jclouds
Upload object task	2 parameters	1 parameters
Read object task	6 LOC, 1 method	7 LOC, 1 method
Code migration task	24 LOC, 2 methods	10 LOC, 1 method

TABLE III: FaaS Function Use Cases

Function	Description
Transform_CSV_File	Reads and transforms CSV file: (formats date columns, converts character field to string, adds processing time and gross margin fields, writes transformed file to obj storage.)
Read_File	Reads any file from object storage.
Read_key-value_pairs	Reads 1k key-value pairs from object storage.
Write_key-value_pairs	Writes 1k key-value pairs to object storage.
Delete_key-value_pairs	Deletes 1k key-value pairs from object storage.
Create_buckets	Creates 10 buckets in object storage.
Delete_buckets	Deletes 10 buckets from object storage.

changes for an expert developer, whereas student solutions may be less efficient resulting in additional changes.

All participants completed surveys (pre-trial and Java assessment) prior to the start of the experiment. After the code migration activity, participants completed a post-trial survey.

Pre-trial survey and Java assessment survey: The pre-trial survey collected the approximate age, years of self-reported Java programming experience, and familiarity with cloud services. The Java assessment survey consisted of 15 multiple-choice questions used to assess participant Java skills. The questions were derived from a Java self-assessment test bank maintained by the computer science department at the University of Washington Tacoma [19]. At the school, the test is optional and self-administered and is used to help students decide which Java programming course to start with upon entering the computer science program.

Post-trial Survey: This survey collected feedback and insights from participants regarding their experiences in performing the code migration task. Participants were asked to rate the perceived difficulty of the training and migration tasks. Participants also provided training and migration task completion times from 15-minute blocks (e.g. 0-15, 15-30, 30-45 minutes, etc.). Participants were asked how intensely they worked on the activity using a 5-point scale and were encouraged to provide additional feedback or comments.

Metrics: used to evaluate developer experiences of code migration: *developer time (minutes)*, *perceived effort* using a 5-point rating scale, *difficulty* using a 5-point rating scale, and *refactoring outcome* e.g. success vs. failure.

D. FaaS Function Use Cases

For experiment I the functions described in table III were refactored to use Apache jclouds replacing Amazon S3 APIs to interface with object storage. Functions were refactored to cache connections to object storage on initialization to avoid reestablishing a TCP connection each time the function was called. We analyzed performance of refactored code with and without connection caching on AWS and Google Cloud.

The Transform_CSV_file and Read_File functions were tested with three different file sizes: 100 rows (13 KB), 10,000 rows (1.2 MB), and 1,000,000 rows (124.8 MB).

E. Platforms and Tools

FaaS functions were tested using the AWS Lambda and Google Cloud Functions FaaS platforms [20] [21]. Each platform provides similar features such as automatic scaling, no server management, and pay-as-you-go payment models.

To investigate cloud service abstraction using Java we used Apache jclouds [3], a multi-cloud toolkit. Jclouds supports cloud providers such as AWS, Google Cloud Platform, Microsoft Azure, OpenStack, and others. It provides abstraction interfaces for compute, blob storage, load balancing, DNS, firewall, object storage, and other cloud services.

We leveraged the Serverless Application Analytics Framework (SAAF) [22] [23] as a FaaS function profiling tool. SAAF supports profiling functions deployed to multiple FaaS platforms including AWS Lambda, Google Cloud Functions, IBM Cloud Functions, Azure Functions, and OpenFaaS. SAAF improves the observability of FaaS functions by enabling server-side profiling to extract insights regarding performance, resource utilization, and infrastructure. For our study, SAAF quantified attributes such as function runtime, memory utilization, and data read/write throughput.

To assess code quality metrics we leveraged JArchitect [18], a static code analysis tool for Java. JArchitect provides extensive code analysis capabilities tailored for Java development. Supported profiling metrics include cyclomatic complexity, class coupling, lines of code, code duplication, and more.

IV. Experimental Results

A. Experiment I.a: Abstraction Library FaaS Performance

To investigate **(RQ-1 Abstraction Overhead)**, we tested FaaS function performance by executing each function 11 times from table III discarding the first run as it was a cold run [24]. We report runtime averages of the subsequent ten warm runs for our functions that used jclouds vs. native libraries.

Connection Caching Performance Implications: Tables IV and Figure 1 describe performance of the Transform_CSV_File function with and without caching connections to the object storage service for different file sizes on AWS and Google. Function runtime using jclouds to access object storage was greater than runtime using native libraries with the largest differences seen when processing small CSV files. When processing larger files, function runtime using jclouds approached that of vendor-specific libraries for AWS (1.06x), and was notably faster on Google (.73x).

Connection caching helped lower function runtime using jclouds on AWS and Google, especially for processing small files. On AWS with jclouds, the runtime was 1.5x of native without connection caching and 1.08x with connection caching. Google was similar with runtime of 1.07x with caching, and 1.26x without. Runtime improvement from caching was less notable when processing large files as the overhead for establishing connections was amortized across longer function runtime. Performance differences for the Transform_CSV_File function tests are shown in Figure 2.

Read_File Function Performance: Table V describes the average runtime and throughput for the Read_File function

TABLE IV: Transform_CSV_File Performance Data

Provider	File Size (rows)	Average Runtime (ms)			
		With Caching		Without Caching	
		Jclouds	Native	Jclouds	Native
AWS	100	623	508	1249	580
	10000	892	912	3711	2870
	1000000	8123	7749	9991	9398
GCP	100	646	606	937	590
	10000	976	687	1075	729
	1000000	11863	16154	12445	17194

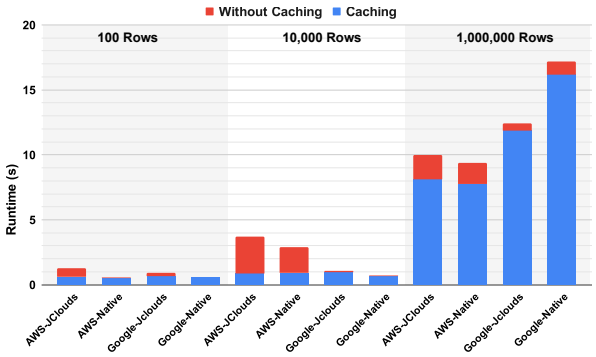

Fig. 1: Transform_CSV_File Function Average Runtime

implementations using Jclouds and native libraries on AWS and Google for reading files of different sizes. On AWS, the jclouds implementation was faster across all file sizes (100, 10,000, and 1,000,000 rows) averaging .95x of the runtime compared to native libraries. On Google, the jclouds implementation was slower than the native for small files: (1.66x 100 rows) and (1.3x 10k rows). For reading large files the jclouds implementation runtime on Google was just (.51x 1m rows) of native. Normalized performance differences for the Read_File function tests are shown in Figure 2. Data read throughput measured in MB/sec with jclouds on AWS provided slightly lower throughput (.87x 100 rows) for small files, but equivalent throughput for larger files (1.02x 10k rows and 1.0x 1m rows). On Google Cloud, jclouds throughput was lower for reading small files (.58x 100 rows) and (.76x 10k rows), but much higher for reading large files (1.8x 1m rows) suggesting that jclouds connection overhead may reduce throughput for small read transactions to object storage.

Other Functions Performance: We compare performance of reading, writing, and deleting batches of 1,000 key-value pairs on object storage using jclouds vs. native libraries as shown in Figure 2. On AWS, jclouds provided equivalent runtime for reading and writing key-value pairs, but deletion runtime using jclouds was slower than native (1.55x). On Google, jclouds runtime was slower for all key-value pair operations: Read (3.65x), Write (1.14x), and Delete (1.22x). Writing 1,000 key-value pairs on Google using jclouds had the highest overhead compared to native libraries of any use case tested.

TABLE V: Read_File Function Performance

Provider	File(rows)	Avg Runtime(ms)		Avg Throughput(MB/s)	
		Native	Jclouds	Native	Jclouds
AWS	100	31	29	0.53	0.46
	10,000	113	104	12.28	12.56
	1,000,000	1358	1351	89.79	89.89
Google	100	54	90	0.26	0.15
	10,000	105	136	12.8	9.7
	1,000,000	5553	2845	22.37	42.18

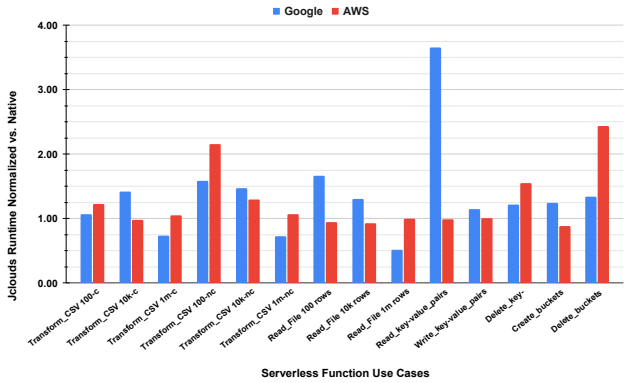

Fig. 2: Normalized Function Runtime Comparison

We also tested performance of creating and deleting ten buckets in succession using jclouds vs. native libraries with the Create_buckets and Delete_buckets functions. Normalized performance differences are shown in Figure 2. On AWS, creating buckets using jclouds was faster than native libraries (.88x). Deleting buckets was considerably slower with runtime of (2.43x) vs. native libraries. On Google, bucket creation and deletion had similar runtime overhead using jclouds (1.25x) and (1.34x) for creation and deletion, respectively.

In summary, we tested object storage operations including reading and writing large files, operations on batches of key pairs, and creation and deletion of buckets with seven FaaS functions and 14 different configurations. **Across all tests, functions with jclouds on AWS had slower average runtime (1.25x) vs. native libraries. On Google, jclouds function runtime was (1.36x) vs. native.** Six functions/configurations on AWS had faster runtime using jclouds, where only three were faster on Google. **Jclouds was faster at reading and writing a large file vs. transactional operations involving multiple key pairs or buckets.**

B. Experiment I.b: Abstraction Library FaaS Code Quality

To investigate **(RQ-2 Code Quality)**, we used the JArchitect static analysis tool to assess code quality metrics. Table VI presents a comparison of metrics for Read_File for our three implementations: AWS-native, Google-native, and jclouds. The AWS-native and Google-native versions only run on a specific cloud, whereas the jclouds version is versatile and can be deployed and run on either cloud platform.

The jclouds version had a larger JAR file size (17.7 MB), in contrast to the AWS and Google versions (∼10 MB). Though each version had seven source files, the jclouds version had more third-party elements (133) compared to AWS (117) and Google (120). The jclouds version had more LOC (308), while the AWS and Google versions had slightly fewer. The AWS version was the initial version and was refactored to produce the Google and jclouds versions. **Producing the jclouds version involved changes to nearly twice as many lines of code (63 LOCR) vs. the native Google version (34 LOCR).** Finally, **the jclouds version exhibited a slightly reduced average method complexity** (cyclomatic complexity of 2.56) in comparison to AWS (2.66) and Google (2.65) suggesting a

possible improvement in code complexity and maintainability.

TABLE VI: Refactored Code - Quality Metrics

	AWS native	Google native	Jclouds
Jar file size (MB)	10	10.1	17.7
Source Files	7	7	7
Third-Party Elements	117	120	133
LOC	283	294	308
LOC refactored	(baseline)	34	63
Average CC	2.66	2.65	2.56

C. Experiment II: Code Portability Empirical Study

To investigate **(RQ-3 Code Portability)**, we conducted an empirical study on FaaS function migration involving two student groups from cloud computing classes to examine the utility of the jclouds cloud abstraction library to support code migration from AWS to Google Cloud. Table II describes the source code modifications required for Group-GCP and Group-Jclouds for the Upload object and Read object function migration training tasks and also the code migration activity. For the code migration activity, participants migrated a function from AWS Lambda to Google Cloud Functions that reads, resizes, and uploads a new image to and from object storage.

For the training tasks, 16 students in Group-GCP successfully migrated the upload object function to Google, while four students encountered difficulties and failed to migrate the function. For Group-Jclouds, 21 students successfully migrated the upload object function, while only one student failed to complete the function migration. We compared the outcomes of the two groups using a two-proportion z-test where the null hypothesis is that the success rate for the two groups is identical. We are unable to reject the null hypothesis (z = -1.5446, p =.12356) and conclude there was no statistical difference in outcome between Group-GCP and Group-Jclouds for the upload training task.

The success rates for Group-GCP and Group-Jclouds for the code migration task are shown in Table VII. Table VIII reports the average code migration task completion times in minutes for successful efforts. Within Group-Jclouds, 11 participants successfully migrated the image resize function to Google (50%), while 11 students encountered difficulties and failed to complete the migration. In Group-GCP, only four participants migrated the image resize function to Google (20%), while many participants (16) failed to complete the function migration. **Using jclouds increased the success of FaaS function migration by 30%.** We compared code migration task outcomes of the two groups using a two-proportion z-test where the null hypothesis is that the success rate for both groups is identical. For the code migration activity, we reject the null hypothesis (z=-2.0265, p=.04236) and conclude that Group-Jclouds had statistically significant better outcomes migrating the image resize function to Google Cloud than Group-GCP. With limited successful Group-GCP migrations, we did not observe a significant difference in code migration times between the groups.

We next treated prediction of code migration task outcomes as a classification problem to facilitate feature analysis and constructed random forest classifiers using scikit-learn [25].

TABLE VII: Code Migration Task Outcomes

Group	Online/Onsite	Section	Number of success		
GCP	Online	undergraduate	2/9	2/14	
		graduate	0/5	(14.3%)	
	Onsite	undergraduate	1/2	2/6	4/20
		graduate	1/4	(33.3%)	(20.0%)
Jclouds	Online	undergraduate	3/9	6/15	
		graduate	3/6	(40.0%)	
	Onsite	undergraduate	2/3	5/7	11/22
		graduate	3/4	(71.4%)	(50.0%)

TABLE VIII: Code Migration Task Completion Times

Group	Online/Onsite	Section	Average time (min)		
GCP	Online	undergraduate	91.5	91.5	
		graduate	DNF		
	Onsite	undergraduate	81.0	95.0	93.3
		graduate	109		
Jclouds	Online	undergraduate	110.7	107	
		graduate	103.3		
	Onsite	undergraduate	106.0	106.4	107.6
		graduate	110.0		

Pearson correlation, a technique for exploring linear relationships between continuous variables, may be less helpful to infer which variables are most influential in differentiating code migration task outcomes, a binary categorical variable. Feature importance in tree-based models such as random forest support exploring relationships between available features and a categorical outcome (success vs. failure). Random forest, an ensemble learning algorithm, combines multiple decision trees to make predictions. It handles complex relationships between variables to effectively handle mixed feature types, demonstrating robust and accurate performance.

We extracted and aggregated features from surveys (pre-trial, java assessment, cloud computing course, and post-trial), experimental observations, code artifact analysis, and course grade data to form a master dataset for classification modeling. This resulted in a feature set with over 100 numerical features.

We performed feature analysis and feature selection by training random forest classifiers and report feature importance for a strong random forest classification model we derived in table IX. Our model had nine features with zero false negatives, one false positive, a precision of 0.9375, recall of

TABLE IX: Random Forest Classifier Feature Analysis

Feature	Description
quiz-1-score	quiz 1 raw score (0-20)
java-quiz-score	# correct answers on java assessment survey
training-time	time spent completing training
completed-course-surveys	# of daily lecture course surveys-completed
years-living-in-WA	self reported years living in WA
course-quiz-score	avg score for quiz 1 and 2 * 20%
course-tutorial-score	avg score on tutorials * 20%
course-surveys-score	avg score on course surveys * 2%
term-paper-score	term paper raw score (0-100)

Feature	Importance	Info Gain	Info Gain Rank	Duplicates info of higher ranked feature
quiz-1-score	0.315	0.182	4	no
java-quiz-score	0.194	0.080	22	no
training-time	0.102	0.136	7	no
completed-course-surveys	0.080	0.116	16	no
years-living-in-WA	0.076	n/a	n/a	no
course-quiz-score	0.076	0.119	14	yes
course-tutorial-score	0.074	0.064	31	no
course-surveys-score	0.043	n/a	n/a	yes
term-paper-score	0.041	0.071	27	no

1, and an F1 score of 0.9677. Importance in the table is the impurity-based feature importance, where higher values indicate more important features. Feature importance is computed as the normalized total reduction of the criterion by the feature, also known as the Gini importance. The table also reports information gain, which is the mutual information represented as a non-negative value that measures the dependency between the feature and code migration outcome. Mutual information equals zero if the two variables are independent, and higher values indicate higher dependency.

The most important feature was quiz-1-score, the raw score of the first of two cloud computing class quizzes, similar to a midterm. The second most important feature was java-quiz-score which captured the number of correct answers on the java assessment quiz. A high java-quiz-score may suggest a student has a strong working knowledge of Java. Training-time, the time spent completing the training task, was the third most important feature. Completed-course-surveys was the fourth most important feature. After lectures in the cloud computing classes, students optionally completed feedback surveys to earn a small amount of extra credit. Dedicated students tend to complete more surveys. The feature likely captured student class engagement, and this was helpful in inferring function migration outcome.

Years-living-in-WA was the fifth most important feature. Undergraduate students reported living in Washington state on average 13.68 years compared to 2.41 years for graduate students. This feature may relate to English language competency. Undergraduate and graduate students had a similar success rate at function migration (34.8% vs. 36.8%) even though undergraduates reported having 3.3 years experience with Java compared to just 1.7 years for graduates. Java familiarity may have been offset by professional experience as graduate students reported an average of 1.43 years experience working in information technology jobs compared to just 0.27 years for undergraduate students. The remaining features duplicated information or were components of the cloud computing course grade. **Six of the nine selected features were course grade components demonstrating a relationship between course grade and successful function migration outcomes.**

To enhance the accuracy of outcome predictions, it is helpful to incorporate a variety of features to differentiate students. Learning new concepts and engaging in the code migration task within a restricted time frame proved challenging for many students. Some students encountered difficulties in setting up the necessary tools, which hindered their progress in completing the migration task. This demonstrated a complex obstacle a developer may face when adopting new cloud services or platforms, highlighting the need for library/tool support to ensure successful code migration outcomes.

V. Conclusions

This research has investigated the utility of cloud abstraction libraries to support FaaS code migration. Ideally, FaaS code would have no vendor specific dependencies, but standard interfaces would exist for all software building blocks to enable creation of portable code. We summarize our findings as follows: **(RQ-1: Abstraction Overhead)** We found use of jclouds on average increased runtime for functions interfacing with object storage by 25% (AWS) and 36% (Google). Performance degradation was highest for transactional functions that interacted with many key-value pairs or buckets. We found that overhead could be reduced via connection caching. **(RQ-2: Code Quality)** Refactoring functions to use jclouds to interface with object storage increased the overall code size by 8% but reduced cyclomatic complexity by 4% suggesting similar maintainability. **(RQ-3: Portability)** In a study involving computer science students, use of jclouds improved serverless FaaS function migration outcomes by 30%(vs. GCP), while Java competency and course grades helped predict success.

References

[1] D. Petcu, "Portability and interoperability between clouds: Challenges and case study," in *Towards a Service-Based Internet*, 2011.

[2] E. Jonas *et al.*, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[3] "Apache jclouds." [Online]. Available: https://jclouds.apache.org/

[4] "Dasein cloud." [Online]. Available: https://github.com/dasein-cloud/dasein-cloud/

[5] "Apache libcloud." [Online]. Available: https://libcloud.apache.org/

[6] "Welcome to mist documentation! — mist 1.8.0 documentation." [Online]. Available: https://mistclient.readthedocs.io/en/latest/

[7] "Public repository of scripts for the cloudcheckr api." [Online]. Available: https://github.com/CloudCheckr/Developer-Community

[8] "Manageiq/manageiq: Manageiq open-source management platform." [Online]. Available: https://github.com/ManageIQ/manageiq

[9] D. Taibi, J. Spillner, and K. Wawruch, "Serverless computing-where are we now, and where are we heading?" *IEEE software*, vol. 38, no. 1, pp. 25–31, 2020.

[10] "Amazon s3." [Online]. Available: https://aws.amazon.com/s3/

[11] "Azure blob storage - microsoft azure." [Online]. Available: https://azure.microsoft.com/en-us/products/storage/blobs

[12] V. Yussupov *et al.*, "Facing the unplanned migration of serverless applications: A study on portability problems, solutions, and dead ends," in *Proc of the 12th IEEE/ACM Int Conf on Utility and Cloud Computing*, 2019, pp. 273–283.

[13] N. Goonasekera *et al.*, "Cloudbridge: A simple cross-cloud python library," in *Proc. of the XSEDE16 Conf on Diversity, Big Data, and Science at Scale*, 2016, pp. 1–8.

[14] U. Agarwal, "Cloud abstraction libraries: Implementation and comparison," 2016.

[15] M. A. D. C. Ismael *et al.*, "An empirical study for evaluating the performance of jclouds," in *2015 IEEE 7th Int Conf on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2015, pp. 115–122.

[16] R. Ré *et al.*, "An empirical study for evaluating the performance of multi-cloud apis," *Future Generation Computer Systems*, vol. 79, pp. 726–738, 2018.

[17] R. Cordingly, S. Xu, and W. Lloyd, "Function memory optimization for heterogeneous serverless platforms with cpu time accounting," in *2022 IEEE Int Conf on Cloud Engineering (IC2E)*. IEEE, 2022, pp. 104–115.

[18] "Jarchitect." [Online]. Available: https://www.jarchitect.com/metrics

[19] "Java programming self-assessment." [Online]. Available: https://www.tacoma.uw.edu/set/programs/undergrad/bscss/java-test

[20] "Serverless computing - aws lambda features - amazon web services." [Online]. Available: https://aws.amazon.com/lambda/features/

[21] "Gcf." [Online]. Available: https://cloud.google.com/functions

[22] R. Cordingly *et al.*, "The serverless application analytics framework: Enabling design trade-off evaluation for serverless software," in *Proc 2020 Sixth Int Workshop on Serverless Computing*, 2020, pp. 67–72.

[23] "Saaf." [Online]. Available: https://github.com/wlloyduw/SAAF

[24] W. Lloyd *et al.*, "Improving application migration to serverless computing platforms: Latency mitigation with keep-alive workloads," in *2018 IEEE/ACM Int Conf on Utility and Cloud Computing (UCC Companion)*. IEEE, 2018, pp. 195–200.

[25] "scikit-learn 1.3.0 docs." [Online]. Available: https://scikitlearn.org/