Analyzing Performance and Cost Variability in Google Cloud Applications

Ian Gorton, Tianyi Chang, Sunjue Li, Huayuan Wu
Khoury College of Computing and Information Sciences,
Northeastern University, Seattle
Seattle, WA 98122, USA
i.gorton@northeastern.edu

***Abstract*—Commercial cloud platforms are becoming the norm for hosting applications developed by a wide spectrum of organizations, from new startups to Government agencies and multinational businesses. The drivers for cloud deployment revolve around enhanced performance and scalability, access to advanced cloud-based technologies, and reduced costs. To realize these benefits, application architects need to design their systems to exploit the cloud-platform features available for, at minimum, client request processing and data storage. These features need to be exploited and configured appropriately in order to achieve performance, scalability and cost requirements. In this paper, we describe initial work on a project to explore the performance, scalability and cost variations that accrue from different architecture design decisions at the cloud platform level. We perform experiments with the Google Cloud Platform to assess the effects of language selection and performance tuning parameters on throughput, latency, and cost. We show that significant differences occur based on these decisions. We conclude with plans for publishing the data and results in an open repository for use by architects and researchers interested in application performance and analysis at scale.**

Keywords—component, formatting, style, styling, insert (*key words*)

I. Introduction

The landscape of software systems has changed profoundly in the last decade. The emergence of massive scale, global cloud service providers has created an inexorable migration of business applications to the cloud. For example, a 2018 cloud computing study by IDG Communications [1] states:

"*With 73 percent of the 550 surveyed organizations having at least one application, or a portion of their computing infrastructure already in the cloud, it is no longer a question of if organizations will adopt cloud, but how.*"

Using virtualization, applications can often be moved to cloud platforms by migrating their existing deployments to cloud resources. This 'lift and shift' approach is a common strategy to bootstrap application migration, Alternatively, applications can be crafted to be coupled, either loosely or tightly, to a cloud platform infrastructure. These cloud native applications [2], whether based on an open source or cloud-propriety software stack, support a myriad of architectural approaches. Architects are thus faced with a complex, multidimensional design space to navigate in order to deliver applications that meet their operational and business requirements.

Cost is a particularly complex issue in cloud systems. Deploying to a cloud reduces initial infrastructure costs, offsetting these with ongoing charges for computation, communications and data storage [3]. Architects must therefore carefully balance cloud resource usage to provide the required quality of service, while controlling costs.

As systems scale, managing costs becomes critically important. A canonical example is the HipHop compiler from FaceBook.com [5], which reduced the number of servers needed to run Facebook and other web sites by a factor between 4 and 6, drastically reducing operating costs. Even at more mundane scales, a 1% cost reduction can translate to millions of Euros/Dollars of annual savings in absolute terms.

Many studies have explored cost in the context of cloud applications and deployments. For example, [6] analyzes the costs of executing scientific workloads in a cloud. [7] analyzes the total cost of ownership for cloud deployments and finds that particular cost factors are frequently underestimated by practitioners. [8] describes and validates a financial model for adoption and implementation of cloud services for Government agencies. Like [7], it focuses on total ownership costs at the overall platform level and does not delve into specific service and cloud resource costs.

However, to the best of our knowledge, no empirical studies have analyzed the implications, in terms of detailed performance and cost, of architectural design decisions at the cloud services level. The results of such studies could be used to:

- Guide architects when making architectural design decisions and trade-offs that contribute directly to performance, scalability and cost
- Enable researchers to calibrate and assess the validity of various software performance prediction models that have been proposed in the literature

This paper therefore describes our initial method and results to quantify the impact of programming language and scaling parameters on cloud application costs and performance. We first describe some of the major components of contemporary cloud platforms that drive initial architecture decisions. We then explain our experimental approach and analyze the results from multiple experiments. These highlight the significant implications of programming language selection and scaling strategies on application performance and cost.

The contributions of this paper are therefore as follows:

1. An empirical study comparing the price-performance ratios of two different programming languages on cloud servers
2. An empirical study of the delivered price-performance ratios of a cloud server using different scaling parameters

II. ENGINEERING CLOUD SCALE APPLICATIONS

Contemporary commercial cloud platforms are incredibly expansive in their technological offering. Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft's Azure comprise suites of software services that enable organizations to deploy applications on virtualized, globally distributed resources. As an example, in 2017, AWS offered more than 90 services[1]. These include computing, storage, security, networking, analytics and machine learning, workflow services, deployment management, and services for mobile and Internet-of-Things applications. Services are typically exposed to end users through APIs for developers to use in their applications. GCP and Azure and are not dissimilar in their scope and capabilities.

For many applications, the core services required revolve around compute and data resources. Table 1 shows a non-exhaustive classification of these resources on AWS, GCP and Azure.

Compute resources can be classified as Virtual Machine (VM) or Serverless.

VM based resources allow architects to provision specific virtualized CPU/memory computational units. These range from low-end single CPUs with limited memory to, for example in the case of Azure, up to 128 virtual CPUs on a single VM with up to 4TBs of memory. Scaling is achieved by using cloud-specific load balancing services. These start new VMs based on application-specific scaling rules as request load grows, and tear down VMs as load shrinks. Cost is based on the number of hours that a VM is deployed and its specification, with more powerful VMs costing proportionally more per hour.

Serverless based compute resources simplify deploying code on a cloud. Compute resources are dynamically provisioned based on arriving HTTP-based requests. On AWS Lambda Services, for example, the architect specifies the amount of memory a service invocation requires, and Lambda provisions proportional CPU resources. Scaling is achieved transparently to the application code. The serverless deployment infrastructure scales the compute capacity based on the number of simultaneously arriving requests. Service cost is based on the number of service invocations and their associated resource usage. Performance and cost can be tuned through configuring numerous runtime parameters that govern how the application behaves.

[1] https://en.wikipedia.org/wiki/Amazon_Web_Services (retrieved November 2018)

Data resources can be classified as relational (SQL) based databases, or NoSQL databases.

SQL based resources provide a managed relational database instance in the cloud. Database resources can be selectively provisioned to give sufficient compute and memory for the anticipated query load. Scalability – scale up - is achieved by provisioning a database engine with greater capacity as load grows. Costs are based on the provisioned resource capacity and amount of disk storage that the database consumes.

NoSQL database resources support cloud-proprietary data storage. Simplified, typically key-value data models provide a database specific API that services must use. The database scales transparently to the application code and is completely managed by the cloud platform. Costs are based on platform-defined read and write units, providing increased throughput at greater cost.

Service	**AWS**	**GCP**	**Azure**
Compute – Virtual Machine	Amazon Machine Image (AMI)	Google Compute Engine	Azure Virtual Machines
Compute - Serverless	Amazon Lambda Services	Google AppEngine (GAE)	Azure Functions
Database – SQL	Relational Database Service (RDS)	Google Cloud SQL	Azure SQL Database
Database - NoSQL	AWS DynamoDB	Google DataStore	Azure Cosmos DB

Table 1 Comparison of Cloud Compute and Database Services

Hence, designing and deploying an application on a cloud involves decisions on which compute and database combination to select. VM-based compute and SQL database resources often provide a simpler migration path for existing applications. However, serverless compute and NoSQL databases can potentially offer greater performance and scalability, albeit at the cost of engineering the application to utilize these services. Currently however, these decisions have to be made 'on instinct', as architects have no rigorously derived evidence that compares the various compute and database services in controlled experiments. This void of knowledge is what we aim to begin to fill with this work

III. EXPERIMENT DESIGN AND METHOD

A. Aims

The goal of this project is to publish the results of a collection of experiments that compare various cloud-based services in application scenarios. In this paper, we investigate two specific design choices in the context of an application running on GCP. Specifically, we experiment with the serverless Google AppEngine (GAE) compute service, and the Google DataStore NoSQL service. Our experiments aim to answer the following research questions:

- *RQ1: Does programming language selection measurably effect the performance, scalability and cost of a GAE application?*
- *RQ2: Do GAE deployment parameters measurably effect application performance, scalability and cost?*

To address these research questions, we have designed a server application that supports an HTTP-based API and writes and reads data from a database. We have implemented this application in different programming languages to enable a direct comparison of their behavior. We have also tested the application with a variety of configuration settings for GAE parameters that effect performance and scalability. The results from these tests enable us to answer RQ1 and RQ2.

B. Method – Test Application and Scenario

For our initial experiments we have defined an application scenario based on capturing and analyzing data from a wearable device such as a FitBit. Data on a user's step count is continually uploaded to a server that is deployed in a cloud. The application's HTTP interface, known as the *Wearable* interface, is as follows:

POST /userID/day/hour/stepCount

Where:

- ***userID*** ranges between 1 and user population size
- ***day*** ranges between 1 and number of days to upload for a test (default 1)
- ***timeInterval*** between 0 and 23 to represent hours in a day
- ***stepCount*** is an integer between 0 and 5000, randomly generated

GET /current/userID

Where:

- ***userID*** ranges between 1 and user population size
- **Returns *stepCount*** - the cumulative number of steps for the most recent day stored for a user

GET/single/userID/day

Where:

- ***userID*** ranges between 1 and user population size
- ***day*** is a specific day number that is stored for a user
- **Returns *stepCount*** - the cumulative number of steps for the specified day

GET/range/userID/startDay/numDays

Where:

- ***userID*** ranges between 1 and user population size
- ***day*** is a specific day number that is stored for a user
- ***numDays*** is the number of days to returns step count totals for, including start day
- **Returns *stepCounts* []** - the cumulative number of steps for each day specified and the total for all steps, i.e. {17900, 11234, 4900, 34024}

This API allows a variety of specific test scenarios to be devised. In this paper, the test scenario we implemented models a single day of data uploads from a fixed user population, interspersed with random requests to read the current step count from users.

The basic scenario models 24 hours of upload and read requests from a fixed user population, which can be varied to control the test duration. The workload defined by this test scenario is write-heavy. It will stress the write capabilities of the cloud-based database, in this case Google DataStore, that is used to implement the application. This is in fact a common workload in Internet-scale and specifically Internet-of-Things applications. In such applications, millions of sensors continually submit new data, which is stored and read by aggregated periodic queries.

C. Method – Test Scenario Implementation

There are 3 components in the experiment we have designed to explore the research questions. These are described below:

Client: We have designed a multithreaded load testing client framework in Java. The framework is designed to enable different workloads, as defined by a dynamically loaded class, to send concurrent requests to the server. The framework provides mechanisms for performance metrics capture and post-processing to enable analysis of the results. The client test scenario is driven by a properties file that defines the class to be executed for the workload, the number of concurrent client threads and requests to execute, and the location to store the processed results. Requests are sent over HTTP using the *Apache HttpComponents* framework (https://hc.apache.org/). The latency for every request is logged asynchronously using buffered output files to minimize the effect on the client performance of metrics capture. As an example, for the longest test scenarios described in this paper, our client generates over 600MB of raw performance measures for every test.

Server: The server defines HTTP accessible methods to implement the *Wearable* interface. We have implemented the server using the GAE libraries that enable the application to be managed and scaled by GAE. We have implemented three versions of the server, one in Java, one in Go, and one in Python. These servers store and retrieve data from Google DataStore. The basic business logic and request handling is the same across all three implementations.

Database: We have defined two data models that implement the requirements for the *Wearable* test scenario. The first trades-off write performance over read performance, and we denote this as the *WRITE-PREF* data model. Write requests simply add an indexed row to a *StepCount* table. All data therefore exists in the same table, requiring a read request to issue a query to return all entries for the specified day (or days if a range query) and calculate the daily total(s) for a user. The second, known as *READ-PREF*, stores all hourly step counts for a user/day in a single row in the database, along with the daily total. A write therefore requires the whole row for the day to be read, modified, and written back to the database. A read simply requires accessing the pre-calculated daily total(s) from the required day(s).

In our default *WearableWorkload* test class used in this paper, the client executes five phases. Each phase executes a percentage of the maximum number of threads the test is configured to run. These are defined as:

1. WARMUP: 10%
2. GROWTH: 50%
3. PEAK: 100%
4. SHRINK: 33%
5. COOLDOWN: 10%

This periodic load variation is common load testing practice. The WARMUP phase enables the test environment to initialize, for example establishing connections, loading buffer pools and warming up database caches. The GROWTH phase enables us to test how the server environment increases capacity as the request load grows. The PEAK phase tests the capacity of the server to handle a sustained peak load. The SHRINK phase tests the ability of the server to scale down as the request load is reduced. The COOLDOWN phase tests that the server environment has reduced capacity back to the minimum required to cost-effectively serve a low request load.

Our test client therefore executes N (as specified in the test configuration file) threads per phase. Each phase is also configured to generate requests for a fixed number of simulated hours in our 24-hour test scenario. For the five phases above, the number of hours per phase by default are {2, 3, 14, 3, 2} respectively. For each hour, a thread executes a configurable number of POST */userID/day/hour/stepCount* requests – known as *numRequests* (in this paper, we use 100 and 1000 requests in our tests). This means, for example, in the PEAK phase with *numRequests=100*, each thread issues 1400 POST requests. In this way, we can easily control test duration.

Each phase also runs a fixed number (default 5) of threads that issue *GET /current/userID* requests. These test the ability of the application to perform reads while supporting a heavy write load.

In our default *WearableWorkload* test class, every client executes a new request immediately after the results from the previous request are received. This 'non-stop' workload generation strategy exerts a high load on the server, with the PEAK phase acting as a stress test. It also avoids the problem of *coordinated omission* [9] that is typically encountered when testing tools such as JMeter are used for issuing a constant load each time unit (e.g. 100 requests/second).

In this paper, we only test two of the HTTP methods and simulate load from a fixed client population for 24 hours. Larger test scenarios can be easily assembled by increasing the user population, thus increasing the database size, and loading/querying data for multiple days. This will also enable the effect of the range query to be quantified on a large quantity of test data.

D. Experimental Approach

Our test environment comprised:

- **Client**: 8 core (i7 vPro *Gen) Dell Latitude 7490 with 1GB Ram and 0.5TB of SSD local storage
- **Server and Database:** Deployed to closest Google AppEngine datacenter to Seattle. At the time of testing this was in the USA mid-west. All configuration parameters were left at default unless explicitly stated.
- **Network**: CenturyLink 100Mbs accessed by a 5G Wi-Fi router. We compared the latency on this network to the one provided at the Northeastern Seattle Campus and found it to provide consistently better performance

To answer the two research questions, we carried out two distinct sets of experiments.

First, to compare the performance of different server implementations, we ran a series of tests against the Go and the Java servers. Both these implement the WRITE-PREF data model, and hence their performance is directly comparable. We executed a sequence of 3 identical, consecutive tests for each server, with the number of client threads set to {128, 256, 512, 1024} respectively for each test sequence. For each test, *numRequests* was set to 100.

From each set of tests results, we calculated:

1. the mean throughput for the whole test
2. the highest throughput observed for a 1 second interval during the peak phase
3. the wall time for the test

Second, to perform a scalability parameter study, we ran a sequence of three identical tests for six test configurations with different parameter values using the Python server. This server implements the READ-PREF data model.

There are nine runtime *auto-scaling* parameters for a GAE server. We focused on varying three of these, namely:

- **target_cpu_utilization:** This parameter specifies the CPU usage threshold at which new GAE instances will be deployed to handle client requests. The default value is 0.6, meaning that once the CPU for a deployed server instance reaches 60%, GAE will start a new application instance. The GAE documentation states that this parameter *"enables you to balance between performance and cost, with lower values increasing performance and increasing cost, and higher values decreasing performance but also decreasing cost"*.
- **max_pending_latency:** This specifies the maximum amount of time GAE will allow a client request to wait in the pending queue before starting new instances to improve performance. The default value is 30ms, and when this threshold is reached, it is a stimulus for GAE

to scale the application server. The GAE documentation states, *"A low maximum means App Engine will start new instances sooner for pending requests, improving performance but raising running costs."*

- **min_pending_latency:** This value (default 30ms) represents the minimum amount of time GAE will allow a client request to wait in the pending queue before starting a new instance to handle it. GAE documentation states *"when this threshold is reached, it is a signal to scale down, and results in a decrease in the number of instances."*

Based on these 3 parameter settings, we defined 5 configurations in addition to the default. These are defined in Table 2.

	Max CPU	Max Latency	Min Latency
Default	0.6	30	30
Latency tuned	0.6	20	20
CPU relaxed	0.7	30	30
Full Tune	0.7	20	20
Full Tune+	0.8	20	20
CPU SuperRelaxed	0.8	30	30

Table 2 Configurations for Scalability Parameter Testing

For each configuration, we ran 3 tests with 1024 as the maximum threads value, and *numRequests*=1000. For each test we processed the results and produced the following values:

- **throughput -peak:** the maximum throughput observed in a one second interval.
- **throughput – mean:** the mean throughput across all 5 test phases
- **total cost:** the total cost of the test, from the GAE Dashboard
- **maxInstances:** the maximum number of server instances that were created during the test, again taken from the GAE Dashboard as the tests ran.

We also separately processed the results during the peak phase, when the client was sending requests from 1024 simultaneous threads. For the peak phase only, we calculated the following values for each test:

- **throughput – mean:** the mean peak phase throughput
- **throughput – peak:** the maximum interval throughput
- **throughput - p5**: the 5th percentile throughput value. A lower value here indicates slumps in the throughput during the peak phase, and is an indicator that the 99th percentile response time is likely to be higher
- **response time – mean:** The mean latency for all requests in the peak phase
- **phase duration (secs):** the duration of the peak phase

Tests were performed from 8am to 10am (Seattle time) over a series of several days. Before testing each day, we ran a 'calibration test' using a deployed Python server instance. The intent of this calibration test was to attempt to ensure that the shared network and cloud resources were behaving similarly to days when we have previously run tests. If the results of the calibration test were within 5% of the mean of all the previous test days, we continued to run tests. There were in fact two occasions when the calibration results were wildly different, and hence we did not run tests that day.

All client and server code and curated results will be available initially to download from github. As the project progresses, we will build a Wed-based repository to enable results access, analysis and downloads.

IV. Results

A. Language Comparison

To investigate the effect of server language selection, tests were run against the Go and Java servers. Client workloads varied from 128 to 1024 concurrent threads. Each test was run three times. All GAE tuning parameters governing auto-scaling were left at default settings.

In Figure 1, we compare the mean throughput of the Go and Java server for the complete 5-phase test. We see that the Go implementation consistently outperforms the Java code. At best, with 512 clients, the Java code delivers 84% of the throughput of the Go server. At worst, with 256 clients, the Java server lags at 61% of the Go server performance.

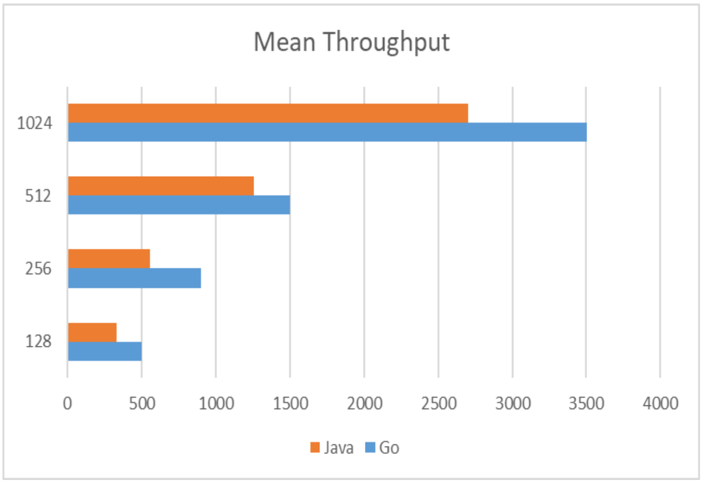

Figure 1 Comparison of Mean Throughput (requests/sec) for 128, 256, 512 and 1024 Clients

Comparing peak interval performance, the difference between the two servers is stark, as shown in Figure 2. At peak, the Go server is able to process 9328 requests/second. In contrast, the Java server delivers between 61% to 77% of the Go code's peak capacity across the tested client loads.

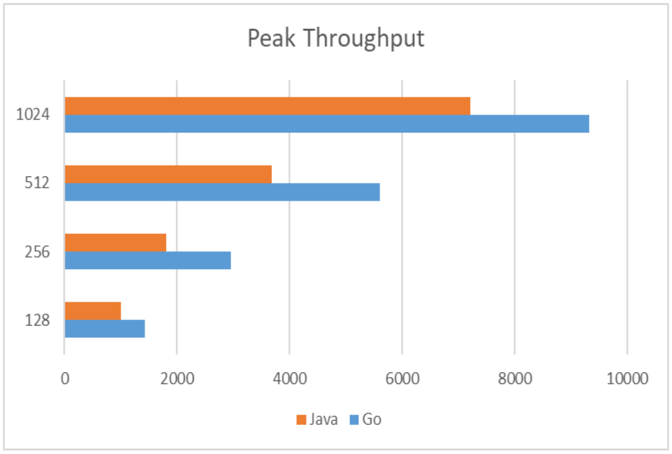

Figure 2 Comparison of Peak Throughput (requests/sec) for 128, 256, 512 and 1024 Clients

Finally we calculated the price-performance ratios for the two servers, as shown in Table 3. While the Go code provides much greater price-performance, it is noticeable that the Java implementation improves as the load grows. We investigated this issue in the GAE Dashboard, and noticed that new Java instances took ~5 seconds on average to load as the server was scaled. Go instances were in the 1-2 second range on average, meaning the server could scale much more quickly. Given that these tests were fairly short (around 3 to 7 minutes in duration), we believe this skews the results in favor of the fast loading Go instances. We can see as the tests grow in duration, the Java codes price-performance ratio improves as the instances are resident for longer and providing greater capacity (i.e. throughput).

maxThreads	Go	Java
128	283	124
256	284	129
512	227	141
1024	228	165

Table 3 Price-Performance Ratio for Go and Java Servers

In summary, for RQ1, we can state that programming language selection is asignificant factor for the performance, scalability and cost of a GAE application.

B. Parameter Study

In order to study the effects of auto-scaling parameters, three tests were run for each of the six different configurations in Table 2. The mean results across all tests were normalized using the Excel STANDARDIZE function to allow a direct comparison across configurations.

First, we compare the results for the full test duration, namely all five phases (typically a test takes around 45 minutes). Figure 3 shows the *FullTune+* configuration provides both the highest peak and mean throughput. In comparison, the other configurations, apart from *CPUSuperRelaxed*, exhibit either above average mean or peak throughput, but not both. *CPUSuperRelaxed* in comparison exhibit considerably lower performance. The *CPUSuperRelaxed* mean and peak throughput are 2237 and 6573 requests/sec. This compares to the means for all configurations of 2478 and 7218 requests/sec, and the highest values 2602 and 7504 requests/sec observed from the *FullTune+* configuration.

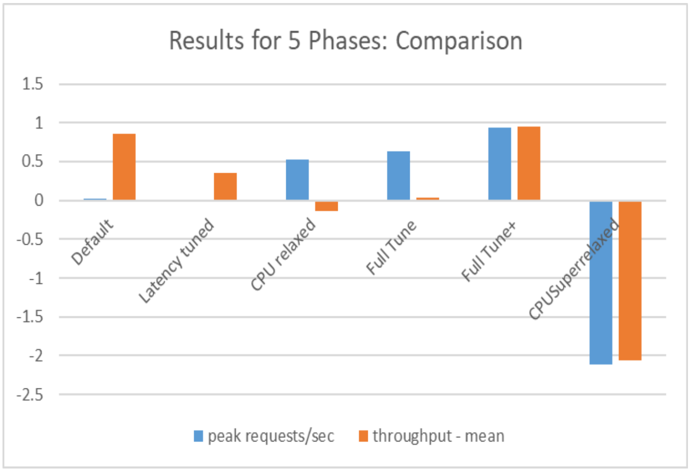

Figure 3 Comparison of Normalized Mean and Peak Throughput Across all Configurations

Figure 4 depicts normalized results for overall test cost and the maximum number of instances spawned by GAE during the peak phase. The *FullTune+* configuration provides the lowest average cost solution. The *CPUSuperRelazed* configuration is cost effective, spawning the lowest number of peak instances. However, this lower cost solution delivers 14% lower overal throughout as compared to the *FullTune+* configuration (see Figure 3). Unlike the *FullTune+* configuration, *CPUSuperRelazed* delivers a cheaper solution but with lower performance.

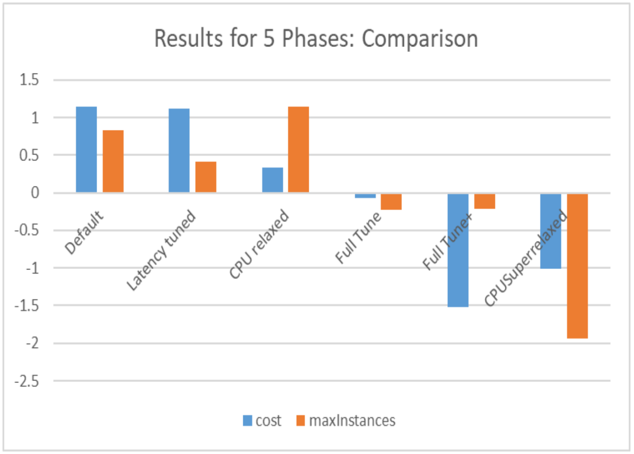

Figure 4 Comparison of Normalized Cost and Maximum Instances and Across all Configurations

Next we analyze the results for the peak phase only. This is when the server platform is under highest sustained load.

Hence we expect to see the effects of the different configurations magnified, making their different behaviors more apparent.

Figure 5 compares the normalized results for the mean and p5 throughput. Again, the *FullTune+* outperforms the other configurations in both measures. The p5 result is especially impressive, with a value of 5887 requests/sec compared to the mean across all configurations of 5343 requests/sec. This shows that the *FullTune+* configuration provides much more stable performance throughout the peak phase. The plot of throughput against time in Figure 6 for one of the *FullTune+* test runs illustrates this behavior. From examining the data, each of the dips in performance during the peak phase are a single second interval. These could be caused by a multitude of both environmental (e.g. network congestion, overload in cloud platform capacity) or application (e.g. database lock) issues. Importantly the server recovers rapidly to its sustained performance level.

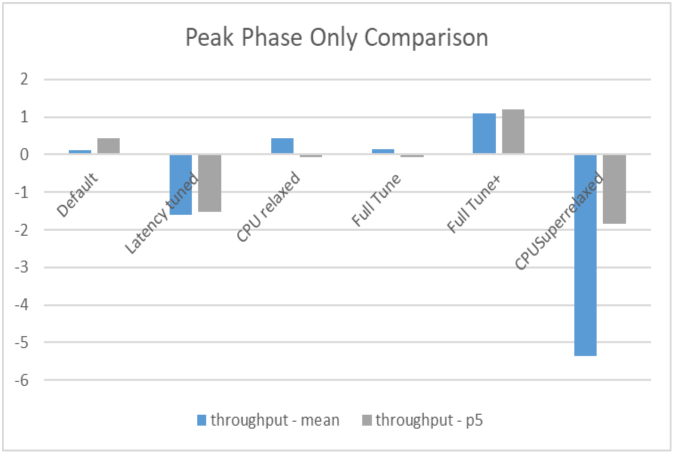

Figure 5 Peak Phase Comparison of Mean and p5 Throughput

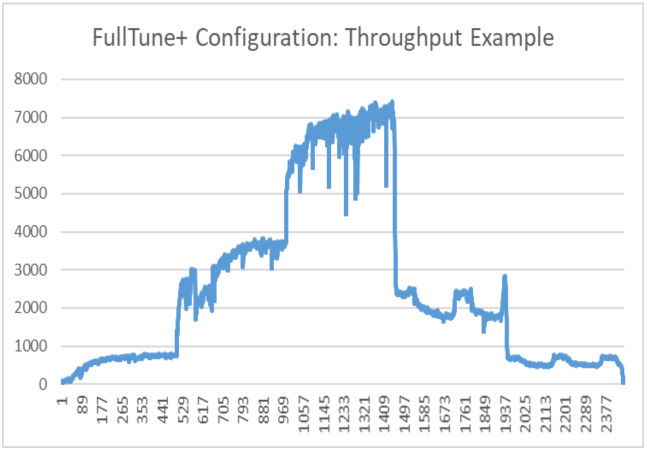

Figure 6 FullTune+ Configuration Throughput Example

Figure 7 compares peak phase mean response time and duration. The lowest values, representing best performance, are again exhibited by the *FullTune+* configuration.

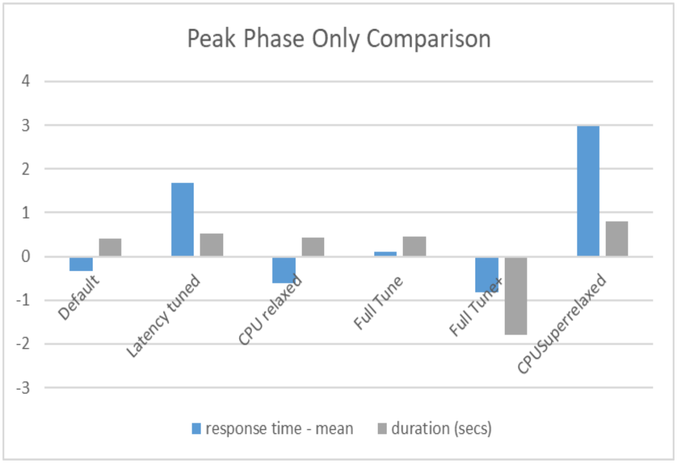

Figure 7 Peak Phase Comparison of Mean Response Time and Phase Duration

These results clearly show the *FullTune+* configuration consistently provides the best solution of the six configurations we tested. 'Best' in this case means both highest performance and lowest cost. The *Default* configuration has similar performance but at an average 11% higher cost. To quantify each solution's combination of performance and cost, we calculate the price/performance ratio as (mean throughput/cost) for the tested workload. This is depicted in Table 4. This clearly shows the advantage of the *FullTune+* configuration.

Configuration	**Price/Performance Ratio**
Default	65.41537
Latency tuned	63.95888
CPU relaxed	64.84997
Full Tune	66.7113
Full Tune+	75.13716
CPUSuperRelaxed	62.8724

Table 4 Price/Performance Ratio for Tested Configurations

Fully interpreting these results requires us to gather more runtime information than we curently have available from our test code metrics and the GCP Dashboard. However, we speculate that the cause is as follows.

As the *target_cpu_utilization* is relaxed from its default 0.6, to 0.7 and 0.8, while other setting are held at default, cost drops along with performance. This makes intuitive sense as the number of processing instances drives cost, and hence the overall server processing capacity is lower.

However, when the *target_cpu_utilization* is relaxed along with the *max_pending_latency* and *min_pending_latency* parameters (20ms as opposed to 30ms default), we do see reduced costs but not reduced performance. The two latency parameters control how quickly the server spawns new instances as load grows, and removes instances as load reduces. With lower values, new instances will be created more quickly, enabling the capacity to increase rapidly until request waiting times stabilize. This should increase throughput. And as load drops off, instances are killed more quickly, reducing cost. With increased a CPU capacity setting at 80%, in conjunction with lower latency parameter values, the *FullTune+* configuration provides the best price/performance ratio of the six configurations we tested.

In summary, these results provide an answer to RQ2, demonstrating that tuning runtime parameters on GAE has a major effect on cost and performance.

V. Threats to Validity

Gathering performance statistics from a cloud environment is a known problem [14], and the greatest threat to validity of the results in this paper. Cloud platforms are a shared resource, and hence executing fully controlled experiments is essentially impossible.

We have attempted to mitigate this problem by running tests during identical periods of the day. We have also run a calibration test before testing periods to check that the cloud and network environment are behaving like previous testing periods. On two days we did experience significance differences during calibration, and hence did not record results that day. However, we have no way of knowing if the cloud platform environment changed during our testing period, which generally lasted between one and two hours. In future work, we will assess the randomized multiple interleaved trials method recommended in [14] to further attempt to detect and minimize environmental changes.

Our budget for testing limited the duration of some of our tests. After an initial experimentation period, we decided to run shorter tests for the language comparison. This was because we saw stark differences between the two codes in shorter tests, and hence longer tests were not necessary to complete the study. We only performed a cursory investigation to see if our conclusions held under higher loads and longer durations. While the initial results were consistent with the shorter tests, we need to more rigorously investigate these configurations to be reduce uncertainty.

Cloud platforms also charge applications based on their network usage. This is typically known as data ingress and egress and is charged on a data transfer size basis. While a simplification, in this work we assume the network costs for an application are independent of the chosen compute and database services. This is certainly true for invoking application services if the clients all see the exact same API. Data transfer costs may vary for an application based on the database they select. Again however, we only compare in this paper applications that utilize the same database resource. Therefore, we can treat data transfer costs as constant and included in the total costs we measure. In future we will attempt to refine costs by breaking them down into different categories to allow more detailed comparisons.

VI. Related Work

Cloud platforms present challenging environments for rigorous performance analysis and prediction. This is due to their shared nature [14] and ever evolving workload patterns [19]. For there reasons, we need new approaches to analyze [10] and model systems and their behavior [13]

There has been previous work on analyzing and modeling a cloud application at the database level. [20] compares NoSQL and NewSQL databases for storing mobile network event data. It find the NewSQL ParStream database is significantly faster for managing this load. [17] performs experiments to compare MongoDB, Cassandra and Riak for a health care application running on AWS. The study find massive throughput variation between the databases, especially for write-heavy loads. [11] builds upon this performance analysis work to create a regression-based prediction model for cloud-hosted NoSQL databases. The model is calibrated based on data from numerous test runs from a multithreaded client accessing the database directly. The model evaluation shows impressive predictive capabilities. However, as there is no cloud-based server tier, this test environment is simplified making the prediction problem less problematic.

[18] and [19] describe the difficulty of cloud-based performance analysis. The former describes monitoring the load of the Gmail service at Google. Two novel approaches, coordinated bursty tracing and vertical context injection are described as solutions at scale for helping characterize cloud-based services. [19] focus on service reliability. It presents a reliability-performance correlation approach based on a Markov model, which considers both VM and physical hardware failure. The model is novel as it can assess reliability in applications that are divided into sub-tasks, such as MapReduce jobs.

VII. Further Work and Conclusions

The technological richness of modern cloud platforms makes them an attractive proposition for an increasing number of businesses. The dark side of this richness is the complexity inherent in service selection, design and configuration. Especially at scale, small optimizations and improvements can deliver huge benefits in terms of service quality and cost reduction.

The results of the study reported in this paper demonstrate some of this inherent complexity. Commercial clouds support multiple programming languages for their serverless platforms, and our initial findings indicate that programming language is an important factor in service performance and cost. More study is needed, and we are already testing a Python version of our server on GCP ,and intend to create Node.js and PHP versions. This will provide more definitive insights into programming language factors.

Equally, we have shown the influence of scaling parameters on cost and performance. By experimenting with different settings for just three parameters, we have shown how our test application can achieve improved performance at lower cost than with default parameter settings. We need to experiment further with other GAE parameter settings to explore how their values effect the cost of scalable service delivery.

Give that these parameter settings represent multiple coupled degrees of freedom, we plan to use an optimized Monte Carlo sampling method to generate candidate configurations for an application. We can then seek out the optimal configuration through running a Monte Carlo style simulation on the application using defined test workloads. The results will enable us to create a response surface that visualizes the interactions of the parameter settings.

We have also commenced similar investigations on the AWS Lambda platform, with Java and Node.js servers running against DynamoDB and MySQL RDS instances. The issues and complexities are similar, but realize themselves practically through different APIs and configuration parameters. We also aim to work with other cloud platforms such as Azure to get a comprehensive perspective of how applications behave across different hosting environments.

Our ultimate aim is twofold. First we want to perform a series of studies across different test applications, cloud platforms and services. The results will give insights to cloud architects on the most influential factors to consider as they build their cloud applications targeted at scale and cost optimization.

Second, we want to provide data from multiple test scenarios for the software performance modeling community. This would be a unique source of data for calibrating and testing performance prediction models for cloud applications. Currently, most projects work with very limited data and attempt to validate their models using cloud simulations. We believe this greatly simplifies the problem, and severely limits the generality and utility of the results.

To serve both aims, we will create a Web=based repository of application code, test data, associated metadata, test analyses and papers. This repository will serve as a valuable source of information for architects, and valuable source of data and code for researchers. Initially we are using github.com for the code and data publication. We have already commenced the design of the repository and will move our initial set of code and results to it in 2019.

Acknowledgment

We would like to acknowledge the GCP Research Credits Team for generously donating credits on the Google Cloud Platform to enable us to carry out this research.

References

[1] https://www.infoworld.com/article/3297397/cloud-computing/cloud-computing-2018-how-enterprise-adoption-is-taking-shape.html, (retrieved November 2018)

[2] https://www.infoworld.com/article/3281046/cloud-computing/what-is-cloud-native-the-modern-way-to-develop-software.html, (retrieved November 2018)

[3] J. U. Daryapurkar and G.B. Karuna, "Cloud Computing: issues and challenges." International Journal on Recent and Innovation Trends in Computing and Communication, Volume: 2 no 4, 770-773.

[4] I. Gorton and J. Klein Distribution, data, deployment: software architecture convergence in big data systems. IEEE Software. 2015 May;32(3):78-85.

[5] Haiping Zhao, Iain Proctor, Minghui Yang, Xin Qi, Mark Williams, Qi Gao, Guilherme Ottoni, Andrew Paroski, Scott MacVicar, Jason Evans, and Stephen Tu. 2012. The HipHop compiler for PHP. In Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA '12). ACM, New York, NY, USA, 575-586.

[6] Iosup A, Ostermann S, Yigitbasi MN, Prodan R, Fahringer T, Epema D. Performance analysis of cloud computing services for many-tasks scientific computing. IEEE Transactions on Parallel and Distributed systems. 2011 Jun;22(6):931-45.

[7] Martens B, Walterbusch M, Teuteberg F. Costing of cloud computing services: A total cost of ownership approach. InSystem Science (HICSS), 2012 45th Hawaii International Conference on 2012 Jan 4 (pp. 1563-1572). IEEE.

[8] Mreea M, Munasinghe KS, Sharma D. Cloud Computing Financial and Cost Analysis: A Case Study of Saudi Government Agencies. InCLOSER 2017 (pp. 431-438).

[9] https://dzone.com/articles/everything-you-know-about-latency-is-wrong-brave-n (retrieved November 2018)

[10] Heinrich, Robert, et al. "Performance engineering for microservices: research challenges and directions." Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion. ACM, 2017.

[11] Farias, V. A., Sousa, F. R., Maia, J. G. R., Gomes, J. P. P., & Machado, J. C. (2018). Regression based performance modeling and provisioning for NoSQL cloud databases. Future Generation Computer Systems, 79, 72-81.

[12] Khanghahi N, Ravanmehr R. Cloud computing performance evaluation: issues and challenges. Comput. 2013;5(1):29-41.

[13] Lladó CM, Sachs K. Theme section on performance modelling and engineering of software and systems. Software & Systems Modeling. 2018 May 1:1-2.

[14] Abedi A, Brecht T. Conducting repeatable experiments in highly variable cloud computing environments. InProceedings of the 8th ACM/SPEC on International Conference on Performance Engineering 2017 Apr 17 (pp. 287-292). ACM.

[15] Al-Faifi, A.M., Song, B., Hassan, M.M., Alamri, A., & Gumaei, A. (2018). Performance prediction model for cloud service selection from smart data. Future Generation Comp. Syst., 85, 97-106.

[16] Farias VA, Sousa FR, Maia JG, Gomes JP, Machado JC. Regression based performance modeling and provisioning for NoSQL cloud databases. Future Generation Computer Systems. 2018 Feb 1;79:72-81.

[17] John Klein, Ian Gorton, Neil Ernst, Patrick Donohoe, Kim Pham, and Chrisjan Matser. 2015. Performance Evaluation of NoSQL Databases: A Case Study. In Proceedings of the 1st Workshop on Performance Analysis of Big Data Systems (PABS '15). ACM, New York, NY, USA, 5-10.

[18] P. Sun, D. Wu, X. Qiu, L. Luo and H. Li, "Performance Analysis of Cloud Service Considering Reliability," 2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), Vienna, 2016, pp. 339-343

[19] Ardelean D, Diwan A, Erdman C. Performance Analysis of Cloud Applications, 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI ’18). April 9–11, 2018 • Renton, WA, USA

[20] Kotiranta, Petri, and Marko Junkkari. "Effectiveness of NoSQL and NewSQL Databases in Mobile Network Event Data: Cassandra and ParStream/Kinetic." Open Journal of Databases (OJDB) Volume 5, Issue 1, 2018