Evaluating Cloud Platform Architecture with the CARE Framework

Liang Zhao, Anna Liu, Jacky Keung
National ICT Australia Ltd.,
School of Computer Science and Engineering, University of New South Wales, Sydney, Australia
{Liang.Zhao, Anna.Liu, Jacky.Keung}@nicta.com.au

Abstract

There is an emergence of cloud application platforms such as Microsoft's Azure, Google's App Engine and Amazon's EC2/SimpleDB/S3. Startups and Enterprise alike, lured by the promise of 'infinite scalability', 'ease of development', 'low infrastructure setup cost' are increasingly using these cloud service building blocks to develop and deploy their web based applications. However, the precise nature of these cloud platforms and the resultant cloud application runtime behavior is still largely an unknown. Given the black box nature of these platforms, and the novel programming and data models of cloud, there is a dearth of tools and techniques for enabling the rigorously evaluation of cloud platforms at runtime.

This paper introduces the CARE (Cloud Architecture Runtime Evaluation) approach, a framework for evaluating cloud application development and runtime platforms. CARE implements a unified interface with WSDL and REST in order to evaluate different Cloud platforms for Cloud application hosting servers and Cloud databases. With the unified interface, we are able to perform selective high-stress and low-stress evaluations corresponding to desired test scenarios.

Result shows the effectiveness of CARE in the evaluation of cloud variations in terms of scalability, availability and responsiveness, across both compute and storage capabilities. Thus placing CARE as an important tool in the path of cloud computing research.

1. Introduction

Cloud computing [1-3] is the next computing paradigm that leads to a new generation of platforms, by utilizing decades of research and technology innovation, to host business and scientific applications to a completely new horizon. It offers a pool of virtualized computing resources to exploit infrastructures, platforms or software as on-demand services. It also allows a pay-per-use model allowing a more utility-like computing model for IT and business users alike. Cloud vendors such as Amazon, Google and Microsoft also provide various innovative cloud based building block services that IT professionals can reuse and incorporate into their own web applications under development.

The projected benefits offered by Cloud computing are compelling and attractive in many ways. The large pool of shared resources such as compute and storage can be allocated dynamically to the global user base. Coupled with the measured service and or metered usage capability at the cloud server end, the cloud service consumers can simply 'pay for usage'. Further, the 'pay as you go' business model means many companies no longer need to invest capital expenditure on hardware at start up time, making cloud computing an attractive proposition for startup companies. The extreme elasticity in the cloud infrastructure means that resources can be scaled up and back down dynamically, and as long as consumers can give up some degrees of control and simply hand over their services to be hosted in the cloud, they can tap into the business benefits of 'economies of scale' offered by cloud computing.

Amazon, Microsoft and Google are investing billions of dollars in building distributed data centers across different continents around the world providing Cloud computing resources to their users. A typical Cloud platform includes a Cloud application hosting server and a Cloud storage device as database. Many also offer additional services such as customizable load balancing, and relational databases.

This paper focuses on these three Cloud platforms:

1. ***Amazon*** offers a collection of services, called Amazon Web Services (AWS), which includes Amazon Elastic Compute Cloud (EC2) as Cloud hosting server, offering Infrastructure-as-a-Service (IaaS), Amazon SimpleDB and Amazon Simple Storage Service (S3) as Cloud databases.
2. ***Google App Engine*** supports a Platform-as-as-service model, supporting programming languages including Python and Java, and Google App Engine Datastore as a Bigtable-based [4], non-relational and highly shardable Cloud database.
3. ***Microsoft Windows Azure*** is recognized as a combination of IaaS and PaaS. It features Web Role and Worker Role for web hosting tasks and computing tasks, respectively. Also it offers a variety of Azure Storages including table storage (as the non-relational option) and Azure SQL (full relational).

There have been a number of research efforts that specifically evaluate the AWS Cloud platform [5] [6]. However, there has been little in-depth evaluation research conducted on other Cloud platforms, such as Google App Engine and Microsoft Azure. More importantly, these work lack a more generic evaluation method that enables a fair comparison between various cloud platforms.

A novel approach called CARE (Cloud Architecture Runtime Evaluation) has been developed in an attempt to address the following research questions:

1. What are the performance characteristics of different Cloud platforms, including Cloud hosting servers and Cloud databases?
2. What availability and reliability characteristics do cloud platforms typically exhibit? What sort of faults and errors may be encountered when services are running on different Cloud platforms under high request volume or high stress situations?
3. What are some of the reasons behind the faults and errors? What are the architecture internal insights we may deduce from these observations?
4. What are the software engineering challenges that developers and architects could face when using Cloud platforms as their production environment for service delivery?

An empirical experiment has been carried out applying CARE against three different cloud platforms. Result provides an in-depth analysis of the major runtime performance differences under various simulated conditions, providing useful information for decision makers on the adoption of different cloud computing technologies.

The next section summarizes related work and how our approach uniquely makes a contribution. Section 3 presents the CARE evaluation framework. Section 4 discusses the empirical experiment set up and its execution. Section 5 presents the experimental results. Section 6 discusses the application experience of CARE and evaluates the CARE approach. We conclude in Section 7.

2. Related Work

Being able to provide an independent evaluation into the architecture and performance of Cloud platform has been the subject of research in the field of computing. Evangelinos and Hill [5], Hill and Humphrey[6] evaluated Amazon EC2 32-bits and 64-bits instances respectively with MPI and memory bandwidth benchmarks to explore the feasibility of running High Performance Computing (HPC) applications on the Cloud. Similarly, data intensive workload cases are also examined on Amazon Simple Storage Service (S3) by Hoffa et al. [7] and Deelman et al. [8]. Nevertheless, these work focus more on analyzing cases of scientific applications, rather than business applications.

While in the case of business application benchmarks, although those traditional benchmarks, namely TPC-W, RuBiS and PetStore are still being used, the effectiveness of benchmarking Cloud platforms using these applications are arguably weak. In particular, Tickoo et al. [9] recognizes that the single highly parallel performance model does not fit virtualized environments; and that virtualization introduces additional resource contentions and overheads that needs to be represented in new cloud based performance models. Sobel et al. [10] moves away from the traditional TPC-W style of benchmark, and proposes that there needs to be new benchmarking applications that better characterizes future cloud application profiles.

From the view of empirical software engineering, there have been Commercial Off-The-Shelf (COTS) middleware evaluation approaches, such as the i-Mate [11-12] and DeBOT processes. The i-Mate selection and evaluation process is originally used to determine appropriateness of middleware products based on user requirements, what-if scenarios and prototype developments. Some of these ideas have the potential to be extended into Cloud platform evaluation and selection, enabling a fit-for-purpose technology assessment. However, cloud based software engineering presents some new challenges, including new programming models and data distribution and replication models, which calls for the need for revised COTS evaluation approaches tailored for cloud.

The most recent related work in this area is from Kossmann's team [13]. It is the first paper comparing a list of Cloud database services by using the TPC-W benchmark. However, their proposed approach contradicts with their previous work [14], which presented the limitations of using TPC-W for benchmarking cloud platforms. Moreover, in order to make TPC-W workable on NoSQL databases, such as Amazon SimpleDB, Amazon S3 and Google App Engine Datastore, additional handcrafted SQL operations are explicitly coded into the implementation of the test application. Hence introducing an extra layer of performance overhead into the benchmark results, that are heavily dependent on the quality of the handcrafted code.

3. The CARE Framework

The CARE framework is a performance evaluation approach specifically tailored for evaluating across a range of cloud platform technologies. The CARE framework exhibits the following design principles and features:

- Common and consistent test interfaces across all test targets, employing web services and RESTful APIs. This is so that we can keep as much commonality

across the tests against different platforms as possible, hence resulting in a fairer comparison.
- Minimal business logic code in test harness, in order to minimize variations in results caused by business logic code. This is so that performance results can be better attributed to the performance characteristics of the underlying cloud platform (as opposed to the test application itself).
- Use of canonical test operations, such as read, write, update, delete. We can then simulate a wide range of cloud application workloads using composites of these canonical operations. We would also then have a precise way of describing the application profile.
- Configurable client simulation component for producing stepped client request volume simulations for evaluating platform under varying load conditions.
- Reusable test components including test harness, result compilation, error logging.
- Consistent measurement terminology and metric that can be used across all test case scenarios and against all test cloud platforms.

3.1. Measurement Terminology

CARE employs a set of measurement terminology that is to be used across all tests, to ensure consistency in the performance instrumentation, analysis and comparison of results.

It considers major variables of interest in the evaluation of Cloud platforms, including response time timings based on those observed by client side, and from the cloud host server side. Figure 1 illustrates the time measurement terminologies in a typical client request and roundtrip response (See Figure 1).

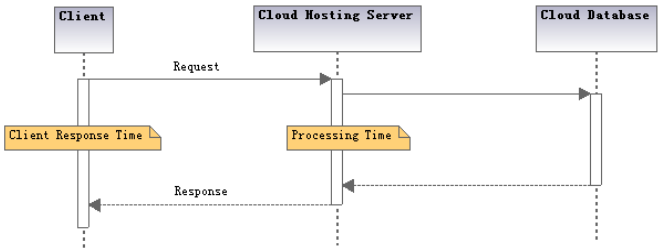


Figure 1. Time Measurement Terminologies

Figure 1 shows a full round-trip transaction initiated by a client. From a user's perspective, a Cloud hosting server and a Cloud database provides three aspects in time-relevant terminologies, they are:

client_response_time is the total (network included) round-trip time as seen by the client, starting from sending the request, through to receiving the corresponding response.

processing_time is the amount of time for processing the request on the server side.

db_processing_time is ideally the amount of time a cloud database takes to process a database request. However, it is practically impossible to measure, due to the absence of a timer process in the Cloud database. The CARE framework thus equates this measurement to time taken to process the database request as seen by the cloud hosting server, measuring *processing_time* of database API as *db_processing_time*, since within the same Cloud platform, the latency between Cloud hosting servers and Cloud databases are negligible.

Further, there are terminologies used to refer to different types of responses based on a request:

incomplete_request is a type of requests when a client fails to send or receive.

completed_request refers to a request that a client sends successfully and receives a confirmation response from the Cloud platform at completion time.

Now, depending on the response, the *completed_request* can be further classified as:

failed_request that contains an error message in the response.

successful_request which completes the transaction without an error.

3.2. Test Scenarios

The CARE framework provides three key test scenarios to differentiate candidate Cloud platforms. While there are other more sophisticated test scenarios possible, the three test scenarios provided by CARE cover most of the usage scenarios of typical cloud applications. Hence, the CARE framework provided test scenarios strikes a good balance between simplicity and coverage.

Client – Cloud Host represents the scenario that a user accesses a web service application hosted on the Cloud platform from an end-user client side application. The *client_response_time* would be the user's primary concern in terms of the Cloud application performance.

Cloud Host – Cloud Database represents the scenario that a user operates on a form or an article hosted in the Cloud database through the Cloud hosting server. We exclude the time taken to send the request from end user client end to cloud host server. The *db_processing_time*s of different data sizes are a main factor taking into consideration. Especially, it would be interesting to observe *db_processing_time* of concurrent requests that are launched by thousands of users simultaneously. The database contention due to concurrent requests will be a key determining factor in the overall scalability of the cloud platform in this type of scenario. Besides identifying different performance characteristics across Cloud databases, a local database (LocalDB) is also provided by the CARE framework in a Cloud hosting server as a reference point for comparison to other cloud databases.

Client – Cloud Database illustrates a large file transfer scenario. We envisage data-intensive computing to be increasingly pervasive in the cloud, and also there is a large variety of new media contents being stored and retrieved from clouds, e.g. video and music, medical images, etc. Understanding the characteristics of cloud (and associated network) behavior in handling 'big data' is an important dimension in the new cloud computing world.

3.3. Load Test Strategies

The CARE framework supports two types of load test strategies: *High Stress Test Strategy* and *Low Stress Test Strategy*. The different load test strategies are ideally applied across the various test scenarios listed in section 3.2, in order to provide a more comprehensive evaluation and comparison.

The *Low Stress Test Strategy* sends multiple requests from the client side in a sequential manner. This is appropriate for simulating systems where there is a small (or a single) number of users. It also provides a reference point for comparison to *High Stress Test Strategy*, and also for obtaining base network latency benchmarks.

The *High Stress Evaluation Strategy* provides simulated concurrent requests to Cloud platforms in order to obtain cloud architecture internal insights, particularly for observing performance behavior under load.

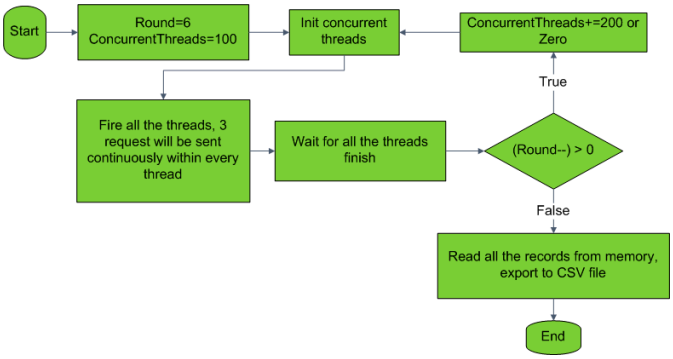


Figure 2. The flow chart of evaluation strategies

Figure 2 demonstrates the workflow of *High Stress Test Strategy*. The configurable parameter called 'repeating rounds' is set to 6 by default, this represents the 'warm up' period, where there is typically large performance variations due to phenomena such as 'cloud connection time'. The performance results arising from the warm up time stage is discarded by the performance results compilation framework, in order to produce more repeatable and stable testing results. Another configurable parameter 'concurrent threads' is set to start at 100 by default. It is then incremented by another configurable parameter 'increment' after every round of test, the CARE framework currently sets the default value to 200 for *High Stress Test Strategy*, and 0 for Low Stress Test Strategy. For example, for High Stress Test Strategy, after the initial 6 rounds, the number of concurrent threads fired by the client would go from 100 to 300, 500, 700, 900 and 1100… in successive rounds.

For the *High Stress Test Strategy*, a number of continuous requests are sent within every thread to maintain its stress on the Cloud platform for a period of time. If there is only a single request sent to the cloud in each thread, our observation is that the expected concurrent stress cannot always be reached, and due to network latency and variability, arrival time and order of packets at the cloud platform can also vary. Hence in the CARE framework, we provide another configurable parameter 'continuous request' (with a default value of 3 – striking a balance between sustaining workload in cloud and enabling testing across concurrent clients) in order to provide a more sustained and even workload to the cloud overtime.

Lastly, as cloud computing is essentially a large scale shared system, where the typical cloud user would be using a publicly shared network in order to access cloud services. We note that there also can be variation in network capacity, bandwidth and latency issues, with fluctuations over time, the CARE framework thus provides a scheduler that support scheduled cron jobs to be activate automatically and repeated taking testing samples across different times over a 24 hour period.

The flow chart of *Low Stress Test Strategy* for requests is essentially a simplified version of Figure 2, with the multi-threaded functions deactivated.

3.4. Building a Test Set with CARE

Now, with the CARE framework, we can combine the various Test Scenarios with the various Load Test Strategies to produce a comprehensive test set.

While the test set can be designed and created using the CARE framework depending on the precise test requirement, the CARE framework also comes with a reusable Test Set that aims to provide test coverage of a large number of commonly found cloud application types. Here we provide an view into this Test Set.

Firstly, there are five contract first Web Service based test methods, namely *High Stress Round-trip*, *low Stress Database Read, low stress Database write, High Stress Database Read and high Stress Database Write.* There are also three RESTful Web Service based methods, *low Stress Large File Read, low Stress Large File Write and low Stress Large File Delete*. The eight key test methods in the test set are listed in Table 1.

Table 1. Building a Test Set

Test Set Methods	Test Scenario	Load Test
High Stress Round-trip	Client – Cloud Host	High Stress
Low Stress Database Read/ Write	Cloud Host – Cloud Database	Low Stress
High Stress Database Read/Write	Cloud Host – Cloud Database	High Stress
Low Stress Large File Read/Write/Delete	Client – Cloud Database	Low Stress

High Stress Round-trip: The clients concurrently send message requests to Cloud hosting servers. For each request received, the servers immediately echo back to the clients with the received messages. The *client_response_time* is recorded in this test. This is the base test that provides a good benchmark for a total round trip cloud application usage experience, response time as experienced by the average user will include and affected by the various variable network conditions. This is a useful test to indicate the likely end user experience, in an end to end system testing scenario.

Low Stress Database Read and Low Stress Database Write uses the *Cloud Host – Cloud Database* scenario. We start with the *Low Stress Test Strategy*, which provides an initial reference result set for subsequent high stress load tests. This test is performed with varying data sizes, representing different cloud application data types. The data types provided by the CARE framework are: a single *character* (1 byte), a *message* (100 bytes), an *article* (1 kilobyte) and a *small file* (1 megabyte). These data types are sent along with the read or write requests, one after another to the Cloud Databases via the Cloud hosting servers. The *db_processing_time* will be recorded and then returned to the client within a response.

In terms of request size, CARE framework follows conventional cloud application design principle, and stores data no larger than 1 kilobyte in structured data oriented storage, namely Amazon SimpleDB and Microsoft Azure Table Storage. While those data larger than 1 kilobyte will be put into binary data oriented databases, including Amazon S3 and Microsoft Azure Blob Storage. In addition, Google App Engine Datastore supports both structure data and binary data in the same Cloud database.

High Stress Database Read/Write is based on the *High Stress Test Strategy*. It simulates multiple read/write actions events concurrently. The number of concurrent requests range is configurable, as described in section 3.3. Due to some common cloud platform quota limitations (for example: Google App Engine by default limits incoming bandwidth at maximum 56 megabytes per minute) this test's default test data size is set to 1 kilobyte. This test data size can be configured to use alternative test data sizes if the target testing cloud platform does not have those quota limitations. Lastly, a cron job is scheduled to perform stress database test repeatedly sampling different time periods across the 24 hour period, as per section 3.3 description.

Low Stress Large File Read/Write/Delete is the test designed to evaluate large data transfer in the *Client – Cloud Database* scenario. The throughput measure is as observed by the client. Once again, this test aims to characterize the total end to end large data handling capability by the cloud platform, taking into consideration the various network variations. The CARE framework provides some default test data: ranging from 1 megabyte, 5 megabytes 10 megabytes and through to 15 megabytes. A RESTful Web Service based client is implemented for a set of target Cloud databases, including Amazon S3 and Microsoft Azure Blob Storage. Note that the CARE framework does not provide a test for Google App Engine, as Datastore does not support an interface for direct external connection for large file access.

4. Application of CARE to Cloud Platform Evaluation

Providing a common reusable test framework across a number of different clouds is a very challenging research problem. This is primarily due to the large variations in architecture, service delivery model, functionality provided across various cloud platforms, including Azure, App Engine and AWS. Firstly, the service models of Cloud hosting servers are different: Amazon EC2 uses the Infrastructure-as-a-Service (IaaS) model; Google App Engine is Platform-as-a-Service (PaaS); while Microsoft Windows Azure combines both the IaaS and PaaS models. Different service models stand for different degrees of system privileges and different system architectures. Moreover, the connections among Cloud hosting servers, Cloud databases and clients are featured with different protocols, frameworks, design patterns and programming languages, these all add to the complexities to the task of providing a common reusable evaluation method and framework.

As shown in Figure 3, for the purpose of keeping as much commonality as possible, Contract-First Web Services and RESTful Web Services are used to ensure a unified evaluation interface. For the Contract-First Web Services: a WSDL file is firstly built; then, the Cloud hosting servers implement to the functions defined in this WSDL file; lastly, a unified client is created from the WSDL file which allows communication via the same protocol, despite of existing variants. While for RESTful Web Services, direct access is made via HTTP. The CARE framework currently provides the reusable common client components, and the cloud server components for Azure, App Engine and EC2.

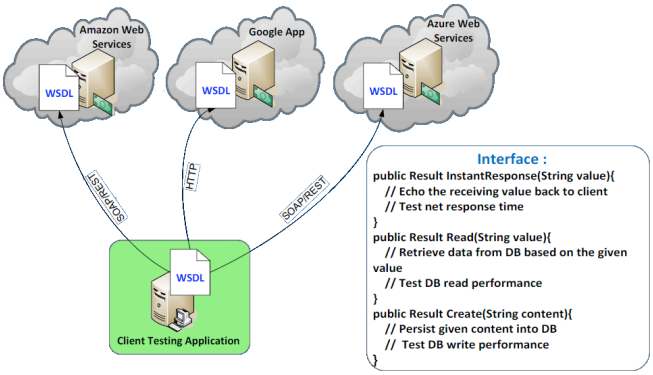


Figure 3. Contract-First Web Service based client

On the Cloud side, in order to provide unified web services, Windows Communication Foundation (WCF) and C# codes are implemented on Microsoft Windows Azure; Python-based ZSI and Zope Interface frameworks are used in Google App Engine; Tomcat 6.0 and Apache CXF run on an Ubuntu-based small instance in Amazon. In addition, a PostgreSQL database, acting as LocalDB, is also installed on the same Amazon instance, accessed via JPA 1.0.

The CARE framework cloud server components follow the design principle of 'always using the native/primary supported language by the cloud' in order to build the most optimal and efficient test components for reuse.

On the client side, a Contract-First Web Service based client is prepared for communicating Cloud hosting servers via WSDL. Whereas a RESTful Web Service based client is used to manipulate Cloud databases directly without passing the Cloud hosting servers.

5. Results

In this section, results of eight evaluation methods is going to be examined, including *Stress Round-trip, Stressfree Database Read/Write*, *Stress Database Read/Write* and *Stressfree Large File Read/Write/Delete*. We also note some environmental information for the tests conducted here:

- the client environment executing the CARE evaluation strategy is a Debian machine with Linux kernel 2.6.21.6-ati, with 3 hardware evaluation platform being a Dell Optiplex GX620, Intel Pentium D CPU 3.00GHz, 2GB memory and 10/100/1000 Base-T Ethernet.

- The sample test results listed here were conducted during the period of April – June 2009.

5.1. High Stress Round-trip

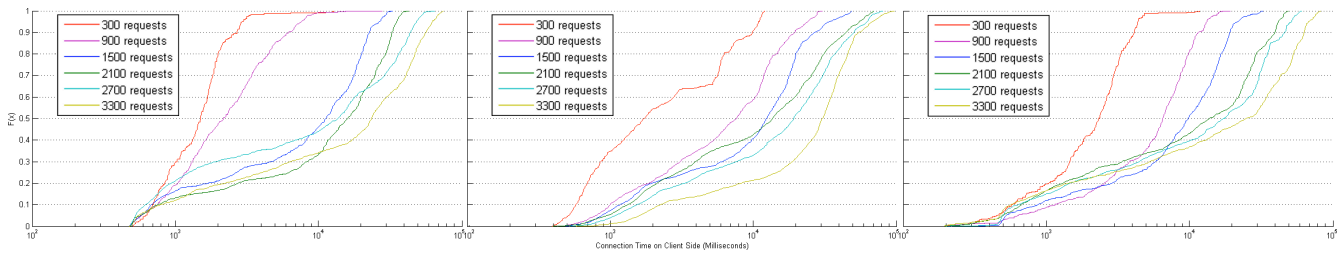
Figure 4 indicates the Cumulative Distribution Function (CDF) of *client_response_time* under varying amount of concurrent stress requests, which range from 300, 900, 1500, 2100, 2700 through to 3300 requests respectively.

The observation of three CDFs confirms that the larger the requests are, the longer the *client_response_time* will be. But the incremental step of *client_response_time* varies from one group of requests to another, depending on the Cloud hosting servers. For 80% of CDFs, the *client_response_time* of Amazon EC2 and Microsoft Windows Azure are dramatically increased at 1500 requests and 900 requests respectively. For Google App Engine, although the *client_response_time* shows an increasing trend, there is no significant leap between neighboring groups of requests.

The reason for these observations could be explained from the scalability aspect. If *client_response_time* increases steadily and linearly under stress in Google App Engine, there is certainly some good scalability capability as its Cloud hosting server is thread based, allowing more threads to be created for additional requests. Nevertheless, the Cloud hosting servers of Amazon EC2 and Microsoft Windows Azure are instance based. The computing resources for one instance are pre-configured. More resources for additional requests cannot be obtained unless extra instances are deployed.

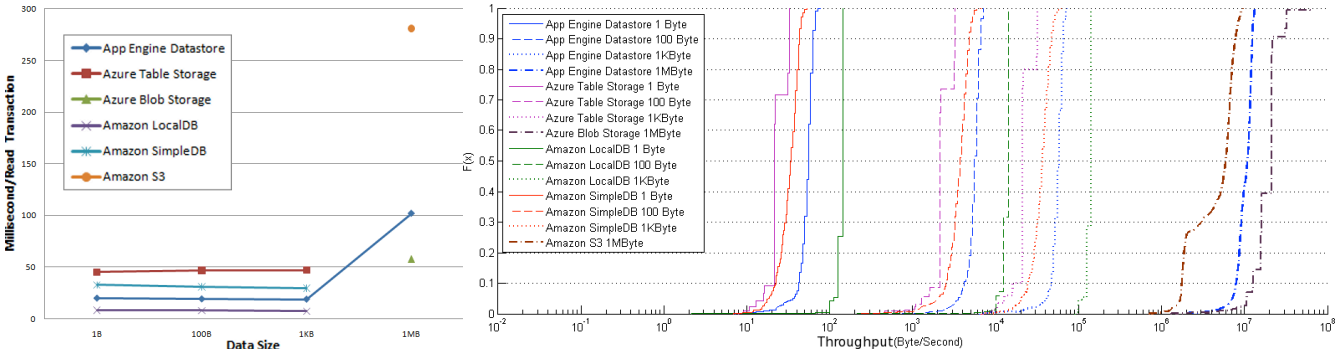
5.2. Low Stress Database Read/Write

In Figure 5(a), the average *db_processing_time* of reading 1byte, 100 bytes and 1 kilobyte are within 50 milliseconds, while the *db_processing_time* of writing small size data in Figure 6(a) varies from 10 milliseconds to 120 milliseconds. From this, we can see that for each Cloud database, the reading performance is deemed faster than the writing performance for the same amount of data. The two figures also state that LocalDB in Amazon EC2



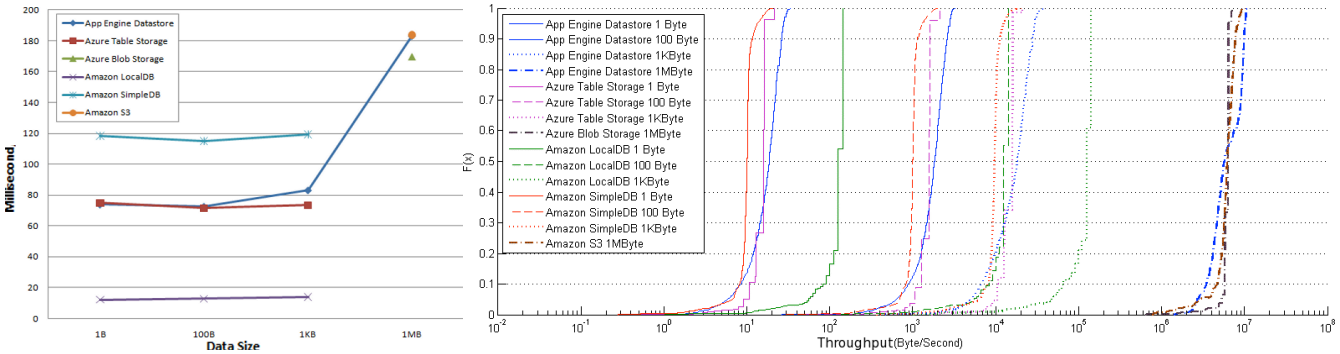
(a) Amazon EC2 (b) Google App Engine (c) Microsoft Windows Azure

Figure 4. CDF of High Stress Round-trip between the clients and the Cloud hosting servers



(a) Average read time in Cloud databases (b) CDF of read throughput in Cloud databases

Figure 5. Low Stress Database Read (1B, 100B, 1KB and 1MB data)



(a) Average read time in Cloud databases (b) CDF of read throughput in Cloud databases

Figure 6. Stressfree Database Read (1B, 100B, 1KB and 1MB data)

instance shows its strength from 1 byte to 1 kilobyte. As the evaluation environment is low stress, and as such, the cloud host is not under load, so it is consistent to see the LocalDB (without any optimizations) can handle requests effectively. The latency from the Cloud hosting server to the LocalDB is also smaller, since they are in the same Amazon EC2 instance.

When the size of request goes to 1 megabyte, Amazon S3 almost has the same write performance as Google App Engine Datastore, but the former is almost three times slower than the latter in reading. Azure Blog Storage takes less time than others in both reading and writing.

The CDFs of read and write throughput in Cloud databases demonstrated similar behavior as in Figure 5(a) and Figure 6(a). Moreover, as for the 1 megabyte database reading and writing, CDFs also show that approximately 80% of requests are processed at 10 megabytes per second.

5.3. High Stress Database Read/Write

In this method, the number of concurrent requests in the evaluation varies from 300 to 3300 with the step being 300. The *db_processing_time*s of each Cloud database under 2100 concurrent requests are collected in Figure 7. From 2100 concurrent requests onwards, Cloud host servers start producing errors, these are listed in detail in Table 5 and Table 6 in Section 5.5.3.

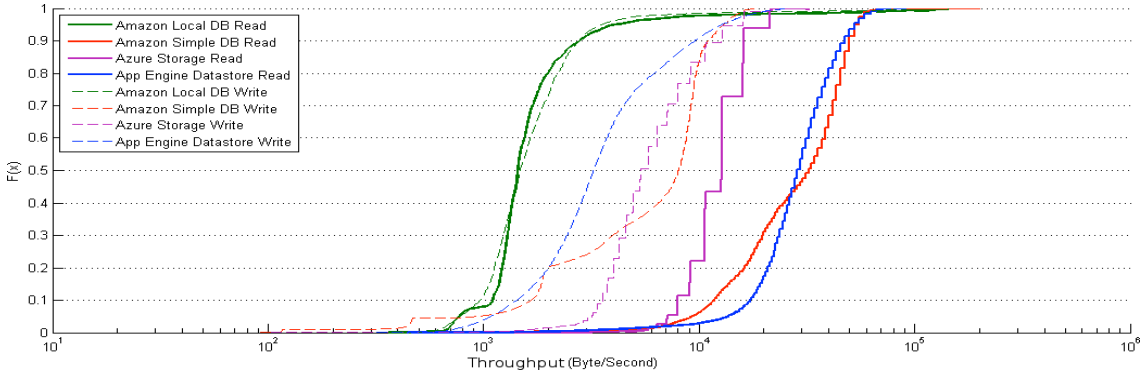
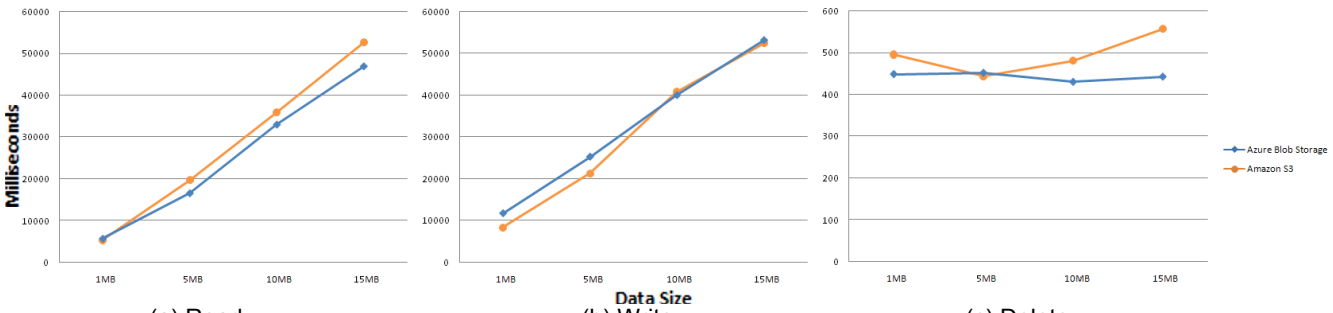


Figure 7. CDF of Stress Database Read/Write throughput on Cloud databases (1KB data)



(a) Read (b) Write (c) Delete

Figure 8. Low Stress Large File Read/Write/Delete (1MB, 5MB, 10MB and 15 MB data)

Instead of being the best performer as in Low Stress *Database Read/Write*, Amazon LocalDB now performs the worst among all platforms. It implies the poor capability of handling concurrent requests within the same instance as the compute capability. Moreover, Google App Engine Datastore, Amazon SimpleDB and Azure Storage all still show faster speed in read operations than write operations.

5.4. Low Stress Large File Read/Write/Delete

Figure 8(a), 8(b) and 8(c) show the average *db_processing_time* of reading, writing and deleting binary files in the Cloud databases directly.

It can be seen that reading is faster in general. Both read and write *db_processing_time*s of Amazon S3 and Azure Blob Storage are fairly close to each other, especially in Figure 8(b) when data size is larger than 5 megabytes. It is likely due to the limitation of the local network environment, the evaluation reaches the threshold of the local network before getting insights of the Cloud databases. This the why the CARE framework provides a range of scenarios (eg. Client – cloud host server – cloud db, as well as cloud host server – cloud db) so that we can evaluate the performance characteristics with and without the network variations and effects in place.

Figure 8(c) indicates the average *db_processing_time* of the delete operation. The interesting observation here is that the delete *db_processing_time* is constant regardless of data sizes. It is confirmed that neither Amazon S3 nor Azure Blob Storage will delete data entries on the fly. Both of them mark the entity and reply with *successful_request* message at the first instant. The actual delete operation will be completed afterwards.

5.5. Exception Analysis and Error Details

5.5.1. Overall Error Details. All error messages and exceptions were logged and captured by the CARE framework. This is a useful feature for carrying out offline analysis. We observe that all errors occurred during the *High Stress Database Read/Write tests*. The CARE framework also logs the errors/exceptions accordingly to various categories:

database_error happens during the period of processing in Cloud databases.

server_error occurs within Cloud hosting servers, for instance, not able to allocate resources.

connection_error is encountered if a request does not reach Cloud hosting servers due to network connection problems, such as package loss, proxy being unavailable.

In general, a response with *connection_error* falls into *incomplete_request*; and a request to *server_error* or *database_error* is recognized as *failed_request*. The error details of each category are listed in Table 4.

5.5.2. Average Errors over Different Time Periods

The CARE framework is also able to produce unavailability information based on error and exceptions

Table 2. Average Error Rates of High Stress Database Read over different time periods

Cloud Database / Category	Google App Engine Datastore		Microsoft Windows Azure Table Storage		Amazon LocalDB		Amazon SimpleDB	
database_error	2.25	(0.007%)	0.00	(0.000%)	0.00	(0.000%)	0.00	(0.000%)
server_error	4.75	(0.015%)	0.00	(0.000%)	16.40	(0.051%)	0.00	(0.000%)
connection_error	5462.75	(16.860%)	11593.80	(35.783%)	6368.40	(19.656%)	41.00	(0.127%)
successful_request	26930.25	(83.118%)	20806.20	(64.217%)	26015.20	(80.294%)	32359.00	(99.873%)

Table 3. Average errors of High Stress Database Write over different time periods

Cloud Database / Category	Google App Engine Datastore		Microsoft Windows Azure Table Storage		Amazon LocalDB		Amazon SimpleDB	
database_error	31.67	(0.098%)	0.00	(0.000%)	0.00	(0.000%)	111.17	(0.343%)
server_error	3037.37	(9.374%)	0.17	(0.001%)	25.20	(0.075%)	9.50	(0.029%)
connection_error	4787.50	(14.776%)	4810.33	(14.847%)	5262.60	(16.243%)	2470.83	(7.626%)
successful_request	24543.66	(75.752%)	27589.50	(85.153%)	27112.20	(83.680%)	29808.50	(92.002%)

Table 4. Total Error Detail Analysis

Category	Error Messages	Reasons	Locations
database_error	datastore_errors: Timeout	Multiple action perform at the same entry, one will be processed others will fail due to contention Request takes too much time to process	Google App Engine Datastore
	datastore_errors: TransactionFailedError	An error occurred for the API request datastore_v3.RunQuery()	Google App Engine Datastore
	apiproxy_errors: Error	Too much contention on datastore entities	Google App Engine Datastore
	Amazon SimpleDB is currently unavailable	Too many concurrent requests	Amazon SimpleDB
server_error	Unable to read data from the transport connection	WCF failed to open connection	Microsoft Windows Azure
	500 Server Error	HTTP 500 ERROR : Internal Error	Google App Engine
	Zero Sized Reply		Amazon EC2
connection_error	Read timed out	HTTP time out	Microsoft Windows Azure Amazon EC2
	Access Denied	HTTP 401 ERROR	Microsoft Windows Azure Google App Engine Amazon EC2
	Unknown Host Exception		Microsoft Windows Azure
	Network Error (tcp_error)	Local proxy connection error	Microsoft Windows Azure Google App Engine

logs over a long period of time. Table 2 and Table 3 show different average error rates of *Stress Database Read/Write* methods over different time periods. As shown in table, both read and write *connection_error* rates of Amazon LocalDB and Google App Engine Database vary in a range from 15% to 20%. We note that this figure is highly variable depending on the 24 hour period, and also is highly subjected to the network conditions, as well as the health status of the cloud server. Amazon SimpleDB takes the minimum rates of reading and writing for less than 10%, especially the reading, which approaches to 0%. On the contrary, Azure Table Storage occupies the largest rate in reading, more than 30%.

In spite of read and write *connection_error* rates, average read *successful_request* rates are high, almost 99.99% of *completed_request*. Although Google App Engine Datastore and Amazon SimpleDB responded write *database_error* for 31.67 and 111.17 times respectively, the write *successful_request* rates are generally high, with the worst one logging at more than 99.67% of *completed_request*.

Among all Cloud hosting servers, Google App Engine exhibits the most number of server errors, containing "500 Server Error" messages. Meanwhile, the largest *server_error* rate happened after May 20 23:30:00 PST 2009. This incident can be tracked in Google App Engine overall system status. There were some significant latency started appearing around one or half an hour earlier than the given time.

5.5.3. Average *connection_error* Rates Under different Loads

In *High Stress Database Read/Write* tests, as expected, we see that the trend of the average *connection_error* rates rise as the number of concurrent requests increases. Google App Engine and Amazon EC2 (SimpleDB) have a smaller percentage trend in reading than writing, while Microsoft Windows Azure and Amazon EC2 (LocalDB) in the contrary, display higher rates in read operatons than write operations.

Amazon EC2 (SimpleDB) maintains the lowest error rates in both reading and writing, almost approaching 0% in read tests. While Amazon EC2 (LocalDB), which shares the same instance with the web application of Amazon EC2 (SimpleDB), started receiving high percentage of *connection_error* from 1500 concurrent requests. The reason of this phenomenon could be explained by that the local database of LocalDB causes additional resource contention by virtually being inside the same instance as the host server instance. Hence leading to a lass scalable architecture (as a tradeoff to smaller latency from host server to cloud db).

For Microsoft Windows Azure, the *connection_error* percentage begins to leap, from less than 1% at 1500 requests, to more than 50% and 30% in reading and writing separately at 3300 concurrent requests. This indicates that we have hit a limit in terms of what this Azure server instance can handle.

For Google App Engine, we observe a large number of connection error under high load. Most *connection_error*s from Google App Engine contain "Access Denied"

Table 5. Average *connection_error* percentage of read requests over different request numbers

Concurrent Requests / Cloud Hosting Server	300	900	1500	2100	2700	3300
Google App Engine	2.11%	11.08%	9.83%	10.74%	23.17%	21.75%
Microsoft Windows Azure	0.00%	0.00%	0.03%	30.24%	48.65%	52.53%
Amazon EC2 (LocalDB)	0.00%	0.08%	0.00%	0.05%	0.32%	0.20%
Amazon EC2 (SimpleDB)	0.00%	0.48%	9.44%	17.09%	23.57%	29.97%

Table 6. Average *connection_error* percentage of write requests over different request numbers

Concurrent Requests / Cloud Hosting Server	300	900	1500	2100	2700	3300
Google App Engine	4.61%	11.83%	23.46%	22.30%	26.67%	28.72%
Microsoft Windows Azure	0.00%	0.00%	0.21%	2.98%	19.62%	30.54%
Amazon EC2 (LocalDB)	1.15%	0.12%	0.97%	6.81%	11.01%	11.13%
Amazon EC2 (SimpleDB)	0.00%	0.53%	6.35%	16.02%	19.88%	23.93%

message, which is a standard HTTP 401 error message. Through cross checking the server side, there is no record of HTTP 401 at all in Google App Engine Dashboard. It means these requests are blocked before getting into the web application. The presumption can be made that the access is restricted due to a firewall in Google App Engine. When thousands of requests go into Google App Engine concurrently from the same IP, the firewall may be triggered. Upon some analysis of how App Engine manages incoming requests (ie. Via HTTP traffic), it is reasonable to conclude that this may be a Security feature around 'Denial of Service Attacks'.

6. Discussion

An empirical experiment was carried out to examine the effectiveness of CARE when applied to testing different Cloud platforms. Results indicate CARE is a feasible approach by directly comparing three major Cloud platforms, including Cloud hosting servers and Cloud databases.

Analysis revealed the importance of acknowledging different service models, and that the scalability of Cloud hosting servers is achieved in different ways. The horizontal scalability is available to some extent in Google App Engine, but is always restricted by the quota limitation. While Amazon EC2 and Microsoft Windows Azure can only scale through manual work on the developer's part, in specifying rules and conditions for when instances should be added. This leads the classic trading off issue of complexity against scalability. Vertical scalability is not possible in Google App Engine since every process has to be finished within 30 seconds, and that we do not have control over the type of machines used for our application in the Google cloud. Where on the other hand, Amazon EC2 and Microsoft Windows Azure allows you to choose and deploy instances with varying sizes of memory and CPUs.

The unpredictable unavailability of Cloud is of a greater issue, particularly for enterprise organization with mission critical application requirements. Whilst in our test, we have noted bursts of unavailability, due to a range of environmental factors, including variable network conditions, we also make the observations that the cloud providers sometimes experience challenges in providing available service. Despite sophisticated replication strategies, there is still a potential risk of datacenter breakdown even in the Cloud, which may in turn affect the performance and availability of hosted applications. We note that at the time of writing, most cloud vendors provide an SLA availability of 99.9%, which is still some way away from the typical enterprise requirement of 99.999%.

The network condition makes a significant impact on the total performance and end user experience for Cloud computing. The performance of the end to end Cloud experience highly relies on the network condition. If a user access cloud services through a poor network environment, it is not possible to take full advantage of the Cloud platforms.

7. Conclusion

This paper introduced a novel architecture runtime evaluation approach for Cloud platforms called CARE. The CARE approach also comes with a framework that include a number of pre-build, pre-configured and reconfigurable components for conducting cloud performance evaluations across a number of example target platforms. The CARE framework is tailored for evaluating various aspects of a Cloud platform at runtime. Given different characteristics of different Cloud platforms, CARE's unified interface allows direct comparison of different Cloud platforms where it was simply not possible.

Empirical result shows CARE is a feasible approach and can be used to identify areas that significantly affect runtime performance and performance bottlenecks. It is considered a significant step forward in Cloud computing research.

We are currently extending the CARE framework to include capabilities of investigating the round-trip time of

the Cloud hosting servers from locations over the world using PlanetLab. The resultant extended framework will then be useful to simulate cloud applications used by a global user base.

We are also extending the CARE framework in the dimension of being able to evaluate data consistency characteristics in Cloud DBs. Early results in this area is promising, and we look forward to present this in a future discussion paper.

The future of cloud is limitless and we hope that this study can serve as a starting point for discussing issues relating to Cloud platform runtime performance evaluation.

8. Acknowledgement

NICTA (National ICT Australia Ltd.) is funded through the Australian Government's Backing Australia's Ability Initiative, in part through the Australian Research Council.

9. References

[1] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, and I. Stoica, "Above the clouds: A berkeley view of cloud computing," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28,* 2009.

[2] R. Buyya, C. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generation Computer Systems,* vol. 25, pp. 599-616, 2009.

[3] L. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds: towards a cloud definition," *ACM SIGCOMM Computer Communication Review,* vol. 39, pp. 50-55, 2008.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *OSDI '06*, 2006, pp. 205-218.

[5] C. Evangelinos and C. Hill, "Cloud Computing for parallel Scientific HPC Applications: Feasibility of running Coupled Atmosphere-Ocean Climate Models on Amazon's EC2," in *Proc. Cloud Computing and Its Applications*, Chicago, IL, 2008.

[6] Z. Hill and M. Humphrey, "A quantitative analysis of high performance computing with Amazon's EC2 infrastructure: The death of the local cluster?," in *Grid Computing, 2009 10th IEEE/ACM International Conference on*, 2009, pp. 26-33.

[7] C. Hoffa, G. Mehta, T. Freeman, E. Deelman, K. Keahey, B. Berriman, and J. Good, "On the Use of Cloud Computing for Scientific Workflows," in *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, 2008, pp. 640-645.

[8] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good, "The cost of doing science on the cloud: The Montage example," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, 2008, pp. 1-12.

[9] O. Tickoo, R. Iyer, R. Illikkal, and D. Newell, "Modeling virtual machine performance: challenges and approaches," *ACM SIGMETRICS Performance Evaluation Review,* vol. 37, pp. 55-60, 2010.

[10] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, S. Patil, A. Fox, and D. Patterson, "Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0," in *Proc. Cloud Computing and Its Applications*, Chicago, IL, 2008.

[11] A. Liu and I. Gorton, "Accelerating COTS middleware acquisition: the i-Mate process," *Software, IEEE,* vol. 20, pp. 72-79, 2003.

[12] I. Gorton, A. Liu, and P. Brebner, "Rigorous evaluation of COTS middleware technology," *Computer,* vol. 36, pp. 50-55, 2003.

[13] D. Kossmann, T. Kraska, and S. Loesing, "An Evaluation of Alternative Architectures for Transaction Processing in the Cloud," in *ACM SIGMOD/PODS* Indianapolis, IN, 2010.

[14] C. Binnig, D. Kossmann, T. Kraska, and S. Loesing, "How is the weather tomorrow?: towards a benchmark for the cloud," in *Proceedings of the Second International Workshop on Testing Database Systems* Providence, Rhode Island: ACM, 2009, pp. 1-6.