Please provide feedback

Scalability Analysis of Blockchain on a Serverless Cloud

Alex Kaplunovich
Computer Science Department
University of Maryland
Baltimore, USA
akaplun1@umbc.edu

Karuna P. Joshi
Information Systems Department
University of Maryland
Baltimore, USA
karuna.joshi@umbc.edu

Yelena Yesha
Computer Science Department
University of Maryland
Baltimore, USA
yeyesha@umbc.edu

Abstract—**While adopting Blockchain technologies to automate their enterprise functionality, organizations are recognizing the challenges of scalability and manual configuration that the state of art present. Scalability of Hyperledger Fabric is an open challenge recognized by the research community. We have automated many of the configuration steps of installing Hyperledger Fabric Blockchain on AWS infrastructure and have benchmarked the scalability of that system. We have used the UCR (University of California Riverside) Time Series Archive with 128 timeseries datasets containing over 191,177 rows of data totaling 76,453,742 numbers. Using an automated Serverless approach, we have loaded this dataset, by chunks, into different AWS instances, triggering the load by SQS messaging. In this paper, we present the results of this benchmarking study and describe the approach we took to automate the Hyperledger Fabric processes using serverless Lambda functions and SQS triggering. We will also discuss what is needed to make the Blockchain technology more robust and scalable.**

Keywords – blockchain, serverless, AWS Cloud, Lambda, automation, messaging, benchmark

I. INTRODUCTION

Organizations are increasingly looking at Blockchain technologies to automate tracking of their enterprise functions and data. Many applications are also trying to incorporate Blockchain into their functionality. However, scalability remains the biggest challenge of Blockchain. Rapid elasticity, one of the key characteristics of Cloud computing [7], requires instant scalability of the resources and hence there is a potential of Blockchain technologies not performing optimally on a cloud infrastructure.

To address this concern, we are currently quantifying the performance and scalability of Blockchain frameworks on Cloud infrastructure. As a first step, we have run experiments to benchmark the performance of Hyperledger Fabric on the Amazon Web Services (AWS) platform. Hyperledger Fabric, a project hosted by the Linux Foundation, is an open source blockchain framework implementation, hosting smart contracts (chaincodes). The Fabric allows developers to create a network and define schema and security constraints/permissions.

Currently, many steps related to installation, deployment, configuration, maintenance, monitoring and running of Hyperledger Fabric are manual. We have significantly automated these steps by using the latest Cloud techniques like AMI images, instance templates, and Serverless functions triggered by SQS messages to load the data. After installing the Hyperledger Fabric and other necessary components (including Composer and Composer restful server), we created a schema and started uploading data from the UCR Timeseries dataset. All the results were saved into a No-SQL DynamoDB for future analytics. Once we installed and tested our large dataset on one instance, we observed that using our automation methodology, we could easily run our tests on other instances in the Cloud. AWS Cloud provided us the option to only pay for the time we used the server instances. Moreover, we could launch any number of servers on AWS and our work was easily reproducible since we could repeat the same experiments later. To further cut costs, we used Spot instances that provide up to 90 percent cost savings compared to On-Demand instances. One of the potential problems with spot instances is that they can be terminated if the discounted price becomes unavailable. However, as our experiments per server took several hours (not days), we could rely on spot instances.

Our experiments have demonstrated that Hyperledger Fabric blockchain while an excellent technology, requires further automation and scalability functionality for Cloud platforms. In this paper, we initially discuss the background and related work in this area. In section III (A through I), we describe the platform we configured for our tests. In section III J., we describe the automation steps we developed, and section IV describes the results of our experiments. We end with conclusions and future work.

II. RELATED WORK

A. Serverless and FAAS Architecture

Serverless computing is defined in [6] as one where the developer has control over the code they deploy into the Cloud, though that code has to be written in the form of stateless functions. The developer does not worry about the operational aspects of deployment and maintenance of that code and expects it to be fault-tolerant and auto-scaling. In particular, the code may be scaled to zero where no servers are actually running when the user's function code is not used, and there is no cost to the user. This is in contrast to PaaS solutions where the user is often charged even during idle periods.

Serverless architectures refer to application functions that run in ephemeral containers (Function as a Service or "FaaS") [2], [3]. They allow developers to concentrate on their granular code (called function or microservice). Those functions can be deployed and tested independently. The cloud provider will be scaling microservices automatically as needed (up and down). The developers should not worry about servers and their configuration. System owner should not buy or provision any hardware (in house or in the Cloud). Users save money because they are paying only for compute time used. They do not pay for idle server time saving up to ninety percent over a cloud VM.

Serverless is becoming a modern computing standard despite of its young age (Lambda service was introduced by Amazon in 2014). Currently, all the leading Cloud provides offer FaaS.

Tab. I. contains the details of the leading Function as a Service (Faas) Cloud Providers, prices and supported languages. We can see that all the leading providers (Amazon, Microsoft, Google and IBM) support microservices and invest into new features development and popularization of event-based cloud functions. Modern Cloud conferences and Serverless conferences present the best practices and state of the art serverless solutions. More and more software development companies (large and small) are using cloud functions and benefit from them.

TABLE I.   SERVERLESS FUNCTIONS COST AND LANGUAGES

Cloud Provider	Service Name	Languages Supported	Price $
Amazon	Lambda	Java Node.js Python C# Go Ruby	.20 per million calls
Microsoft	Azure Functions	Node.js Python C#	.20 per million calls
Google	Cloud Functions	Node.js Python Go	.40 per million calls
IBM	Cloud Functions	Java Node.js Python	0.000017 per second, per GB

Our blockchain transactions will be invoked from Lambda functions, calling Hyperledger Composer restful service. There is no need to install any additional server for our processing. Moreover, we can limit a number of simultaneously running Lambda functions to control parallelism.

Serverless functions have limited execution time – they can not run longer than fifteen minutes (depending on the Cloud provider). Given such a limitation, we have to code our function to be able to complete within the specified time interval. A Lambda function can queue another event to spawn another function to complete the work.

For our benchmarking, we have deployed two Lambda functions into the cloud – loadTypes and loadSeries. The first one will be used to load dataset types; the second to load time series data associated with datasets. Lambda functions will be triggered by SQS messages (secion III E.). We deploy and test our functions using API Getaway and SAM (Serverless Application Model) [14]. It is important to deploy them together as one serverless application utilizing best practices, CloudFormation and the Cloud ecosystem bundling. We can easily add more functions to the serverless application if needed in the future.

As Peter Sbarski said in [12] "Serverless architectures are the latest advance for developers and organizations to think about, study, and adopt. This exciting new shift in architecture will grow quickly as software developers embrace compute services such as AWS Lambda. And, in many cases, serverless applications will be cheaper to run and faster to implement".

B. Blockchain Software Benchmarking

There were numerous Hyperledger benchmarking studies (see [8], [11], [13] and [15]). Those studies were not using the Cloud ecosystem integration. Our work adds full automation, Cloud ecosystem integration, and serverless architectures methodology.

III.   IMPLEMENTATION

Our approach provides full integration with the Cloud, orchestration and automation. Hyperledger Fabric installation and launch is done automatically as a part of instance AMI and User Data scripts (see below).

Blockchain transactions are invoked by the SQS messages triggering Lambda microservices. There is no need to make any manual steps because the Cloud ecosystem helps us to handle many issues – message retries, Lambda functions and No-SQL database auto scaling, monitoring and logging.

We have been using s time series dataset. If we decide to perform our tests on another data, we will need just to change the S3 bucket name in the launch python script. Our implementation is so flexible, that if we decide to record results to another database table, we just have to change the table name parameter in the script.

Our approach allows us to complete our tests quickly on a multitude of EC2 instances (with different types, number of cores, and memory), save the performance results into the database and analyze them.

A. Blockchain Software Installation

On top of Hyperledger Fabric and docker software, we have been installing Hyperledger Composer [4] and Hyperledger Composer REST Server [5].

Composer simplifies the configuration of your network, provides a pseudo language to create a model, command line interface, and a visual tool to edit our model and configuration parameters. It takes much less time to maintain a network using the Composer tools. Our model (defined in

section III B.) is an example of the pseudo language usage. Otherwise, we would have to edit numerous text configuration files of a plain Hyperldeger Fabric installation. With a Composer web interface, we can view and configure networks from a browser.

```
list=[]
url = 'http://' +ip + ':3000/api/org.blockchain.series.SeriesRow'
data = {
    "$class": "org.blockchain.series.SeriesRow",
    "rowId": type_name_ + sep + type + sep + str(saveSize) +
sep + str(len),
    "label": row[0].values[0].item(),
    "timeSeries": serie,
    "len": len(serie),
    "dataType": type_,
    "typeRef":        "org.blockchain.series.SeriesType#"        +
type_name_,
    "participantId": "lambda"
}
list.append(data)
response = requests.post(url, json=list, timeout=840)
```

Figure 1.   Hyperledger Composer REST Server call example

Hyperledger Composer REST Server provides a set of methods (called over HTTP protocol) that allow to manipulate the network data – query, update, create, etc. Those methods are closely connected to out network and can be called from a client. The REST server communicates with Blockchain for transaction processing. It is important to mention that when a REST server is started it is associated with an identity, and all transactions will be securely signed with that identity.

For example, let us demonstrate the convenience of the REST Server API. Fig. 1 shows how to add a list of time series into our Blockchain using a simple POST HTTP request. Please, note, that timeout is specified in seconds.

B.   Network Model

```
asset SeriesType identified by typeName {
    o String typeName
    o String testFileName
    o String trainFileName
    o String description optional
    o Integer trainSize optional
    o Integer testSize optional
    o Integer numCols optional
}
asset SeriesRow identified by rowId {
    o String rowId
    o String label
    o Double[] timeSeries
    o Integer len
    o DataType dataType
    --> SeriesType typeRef
    o String participantId
}
```

Figure 2.   Hyperledger Fabric Composer Model

Blockchain Network model defines the objects stored in the framework. Hyperledger Fabric allows creating object-oriented model with members containing most of the common data types, enumerations, arrays and references to other objects.

We have defined the following two objects – SeriesType and SeriesRow. As the names suggest, SeriesType represents one of our time series – name, description, file names and the number of rows/columns.

Our second object – SeriesRow will store one row from a dataset file, containing an array of Double numbers – our time series, identification parameters and a reference to the corresponding SeriesType object. Fig. 2 contains the model definition used for our study.

C.   Dataset

We have used the UCR Time Series Classification Archive [1] dataset, which contains 128 different time series. Each of the dataset contains test and train files, with series length ranging from 15 to 2866 numbers per row. Overall, the repository contains over 76,453,742 floating point numbers. As the data is very different, we had to load it in parallel chunks to test the Hyperledger Fabric capacity to load data over a Composer restful web service.

Since the restful service allows to load data row by row as well as loading multiple rows with one call, we were loading data in the following chunks - 1, 2, 5, 10, 20, 50, and 100. In other words, for each dataset (out of 128), we were loading each file (test and train files for each) multiple times in the chunk sizes mentioned above. To accomplish that we have created a network schema handling such a load pattern.

D.   Server Configuration

Server configuration is an integral part of the Blockchain frameworks usage. It should be as simple and flawless as possible. Utilizing the Cloud tools can help us to create a working Fabric instance within a couple of mouse clicks.

Originally, we installed Hyperldeger Fabric and Composer with all the dependencies on a cloud server. Then we created a network with the schema defined above. We used Ubuntu (Unix) servers with 16 cores and 16GB of memory.

After verifying that all the docker containers for the Fabric work, we created an AMI (Amazon Machine Image) from the instance. Image creation takes a couple of clicks, we just have to name it with a human readable name ("FinalBlockchainImage", for example). Once the image is created, we can spawn instances from the image, and the instances will have all the software and network image installed. We will just have to start the Hyperledger Fabric, Composer and Restful service during the instance startup.

The second automation we use is to specify a startup script called User Data. The cloud infrastructure allows to specify that script either through the web interface or over CLI (command line interface). The following script Fig. 3 will start our Fabric so we can use it upon server start.

```
#!/bin/bash
cd /home/ubuntu/fabric-dev-servers
./fabricUtil.sh start
composer-rest-server -c admin@series -n always -w true &
composer-playground &
```

Figure 3.   User Data starting the Hyperledger Fabric

After we configured User Data, we create a Launch Template from AWS web application. It allows us to specify all the parameters for our new instances, such as number of processors, memory, key pair name, AMI and User Data. Once we have created a Launch Template, we can create our instance using just a couple of clicks. And the instance will be running the blockchain (since we create it from an image), it will also have the network and will be ready for us to start saving data into our blockchain. To test it, we can simply go to the web broser and type the server's url on port 808 – we should see the Hyperledger Composer Playground home screen. Fig. 4 demonstrates a web interface for Hyperledger Composer.

Web interface allows to view and modify your existing networks as well as creating the new ones. Although all the operations can be launched using command line interface (CLI), it is much easier and faster to use a web application.
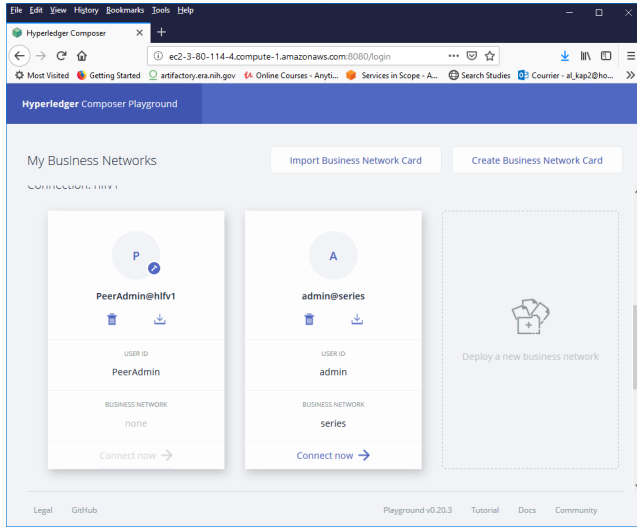

Figure 4. Hyperledger Composer web interface

E. SQS Messaging

SQS is a Cloud messaging service (Simple Queue Service). It is a fully managed service, that is why we do not need to install any additional software, procure any server. We only need to create a messaging queue and use it to send messages. For our benchmarking we have created two queues – load_types and load_series. The names of the queues are self-explanatory. The first one will be handled by the loadTypes Lambda function, the second – by loadSeries.

SQS messages can be a source for Lambda functions. In other words, we can configure the cloud infrastructure to execute a certain code when a message arrives to the queue. Such a configuration provides limitless flexibility, scalability and convenience.

F. Enqueing and Processing

The first thing we did – placed the input files into the Cloud S3 bucket (Simple Storage Service). The whole archive contains 383 objects and has 808.7 Megabytes of data.

G. Launching script (launch.py)

We have developed a little python launch program that will enqueue a loading message into the queue 'load_types'. Our program was made generic, the following parameters are passed to the message

- tableName – the name of the DynamoDB table storing the timing results
- serverType – type of the instance used
- server – IP address of the server running our Hyperledger Fabric
- bucket – the name of S3 bucket storing the data archive
- subfolderName – the name of the subfolder in the bucket above
- queueName – the name of queue that will be loading the series.

Using boto3 library we can easily access most of the Cloud services. Fig. 5 contains most of the code for our launching program.

```
queue =
sqs.get_queue_by_name(QueueName='load_types')
data = {'bucket': bucket, 'subfolderName':
subfolderName, 'queueName': queueName,
'server': server,
        'serverType': serverType,
'tableName':table}
response = queue.send_messages(Entries=[{
        'Id': '1',
        "MessageBody": json.dumps(data)
    }])
print('SQS response',response)
print(response.get('Failed'), data)
```

Figure 5. Python program launching series load into Blockchain

Once our serverless Lambda function loadTypes is configured to be triggered by an SQS message in the queue load_types, we can create the code that will be processing that message.

H. loadTypes Lambda function

The code for the lambda function loadTypes reads the objects in the archive and populates the SeriesType objects in our network model. We are parsing the series README files to extract the number of rows, columns using regular expressions. To decrease the amount of redundant information, we save the name of the test and train files into the same object SeriesType (see Fig. 2 for the schema definition).

To save type objects, we are using composer restful server, installed in previous steps. It allows to query or update our blockchain data using HTTP requests. To save the data we just need to issue a simple POST request and pass the JSON object containing our type(s). We were saving

the types in the chunks of twenty records to accelerate the process of loading the series types.

After loading the types, we enqueue messages into the load_series SQS queue to load the actual time series data. To make the load easier, we enqueue one message per file (one for test file, another one for train file). This way we save a lot of error processing logic if error occurred during a file processing.

I. loadSeries Lambda function

To load a time series file, we enqueue a message containing all the necessary information to load data from S# to the Blockchain. Fig. 6 demonstrates the message enqueueing code.

All the messages will be safely placed into load_series and processed by the corresponding lambda function in parallel (since multiple functions can be spawned). Moreover, we can control the number of loadSeries functions executing at the same time to control the bandwidth of the system. In order to assure that all the types are saved into Blockchain framework, we set the delivery delay to five minutes to assure that the series would not start loading until all the series types are loaded. As you can see in Fig. 7, numerous SQS parameters can be specified, including maximum message size, default visibility timeout and message retention period.

```
response = queue.send_messages(Entries=[{
    'Id': '1',
    "MessageBody": json.dumps({
        'bucket': bucket,
        'subfolderName': subfolderName,
        'file': data['trainFileName'],
        'type': "TRAIN",
        'cols': data['numCols'] if 'numCols'
in data else "Vary",
        'typeName': data['typeName'],
        'server': server,
        'serverType': serverType,
        'saveSize': saveSize,
        'queueName': queueName,
        'tableName': tableName
    })
}
```

Figure 6.   Message enqueuing code to load data from file

As soon as loadType Lamda function starts processing the SQS messages, the archive files will be loaded into the Blockchain one by one. Given we were going to test chunking of the records loaded at once (chunks of size 1, 2, 5, 10, 20, 50, and 100), we enqueue seven messages per file. In other words, we will be loading data for each file seven times, passing chunks of those sizes to the Hyperledger Composer restful server. We are going to compare timing results depending on chunk size.

The loadTypes Lambda function executes the following logic:
1. Obtain a data (CSV) file from S3
2. Read a file by chunks of 10000 lines
3. For each chunk of the time series file

4. If Lambda remaining execution time is less than one minute – schedule a new SQS message with the file name and chunk number parameters to complete the file load in the future
5. For each row in the chunk
6. Populate the array of time series numbers for the row
7. Construct a JSON object *SeriesRow* and append it to the list
8. Save the list to Blockchain if its size is equal to saveSize (a parameter passed to the message, one of (1, 2, 5, 10, 20, 50, or 100)
9. Save all the timing results into the DynameDB table tableName (another parameter passed to the SQS message)
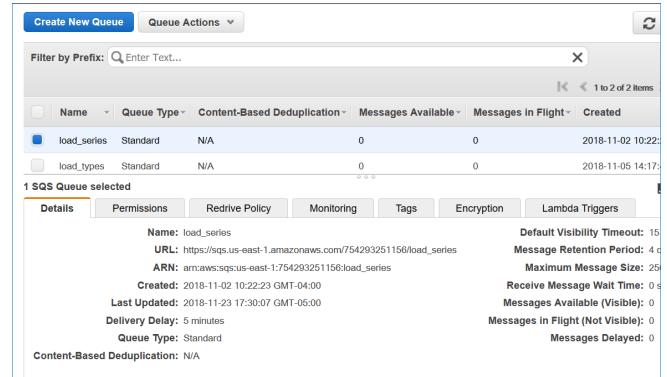

Figure 7.   SQS messaging queue configuration

As you can see, we are handling all the use cases, including the situation when one file data cannot be saved into Blockchain due to latency or file size. In such a case, we are queueing another message with the file chunk number to complete the load with another Lambda serverless function.

J. Automation

Our goal was to automate everything. From the server creation to loading data into the Blockchain and saving timing results. Since the machine AMI image contains all the necessary software installed (including Hyperledger, Composer, Python, boto3 and our launch.py files), we can easily include all the startup commands into our User Data script (see section III D.).

We can disown the running processes from the users to be able to continue execution after user log in. The following addition to the User Data (Fig. 3) will start the loading process (as mentioned in section III L.).

disown %1 %2
python launch.py &

The last statement will start our experiments on the launched server. All the benchmarking data will be saved into DynamoDB database and will be analyzed later.

K. Cost

Let us discuss the prices associated with our Cloud tests. Hyperledger Fabric Blockchain software is free. We will be paying just for the server time, serverless functions execution, SQS messaging and DynamoDB.

Once we are using spot instances, we will be saving up to 90% of the on-demand instance price. Spot instance prices fluctuate over time; however, we can always set the highest price we are willing to pay for the instance. It is possible to make maximum price to be the on-demand price – in such a case your instance will never be terminated, but you might be paying the full price without savings. Fig. 8 Demonstrates the Spot instance dashboard. For example, a spot instance with 16 CPUs and 30GB of memory (c3.4xlarge) will cost 23 cents per hour, while on-demand price for the same instance will be 84 cents per hour.
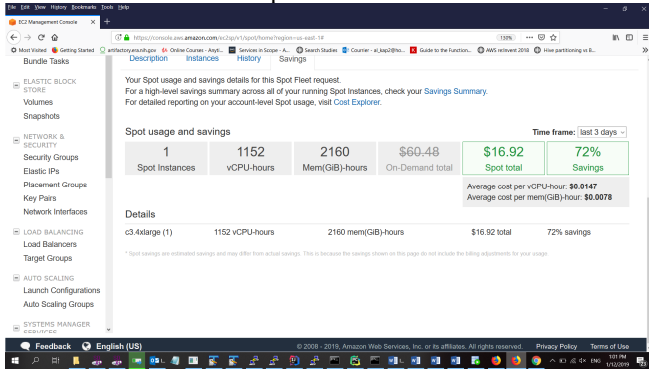

Figure 8.   Spot instance savings dashboard, showing 72% savings

Other costs associated with our experiments are negligible compared to the EC2 pricing.

Lambda microservices cost 20 cents per one million function invocations (after first one million free calls).

SQS will cost us 40 cents per one million messages. Given the nature of our data and messaging structure, we will not exceed one million messages for our tests.

Moreover, DynamoDB pricing is based on WCU (Write Capacity Unit) and RCU (Read Capacity Unit). During our peak loads we set WCU to 200 during our testing, and reset it to 2 afterwards. WCU costs $0.00065 per hour, and RCU costs $0.00013 per hour.

Without the Cloud ecosystem, our system will require manual steps and will cost much more. We are utilizing all the Cloud advantages.

L. Architecture

The architecture diagram for our experiments is below. The launch script enqueues a message into load_types SQS messaging queue. loadTypes Lambda function is automatically triggered as a result of that message. It loads time series types into our Blockchain and adds a bunch of messages into load_series queue (128 datasets * 2 files (train and test) * 7 chunk sizes).

The messages in the load_series queue will automatically trigger loadSeries Lambda functions, each of it will load the corresponding file from the dataset. During the data load executions, all the timing results will be saved in the DynamoDB table specified by the launch.py message parameter.

Our architecture guarantees full automation and efficiency. Fig. 9 shows the architectural diagram of our system.

During our tests we have been using AWS monitoring tools. They allow to set alarms and see the bottlenecks of the system. Snapshot graphs helped us to determine what write capacity we need to set for the DynamoDB, or how many Lambda functions are executing simultaneously.

The beauty of the Cloud infrastructure ecosystem– we can check all the services we use from a single web application. We can check how many SQS messages are waiting in the queue, how much money we save with the EC2 spot instance, or how many Lambda functions had errors during execution. That information was priceless during tuning of our system to find the most appropriate configuration for the best results. All the timing information was saved in the serverless DynamoDB No-SQL database, and will be used to analyze our results in the future sections of the paper.

Hyperledger Fabric is using a CouchDB database to store its transactions. It is a NoSQL document database developed by Apache Software Foundation. Unfortunately, it is not a part of the Cloud ecosystem and its rating is not better than DynamoDB (see [9]). Although CouchDB provides a querying mechanism (see [10]), its performance is not better than DynamoDB.
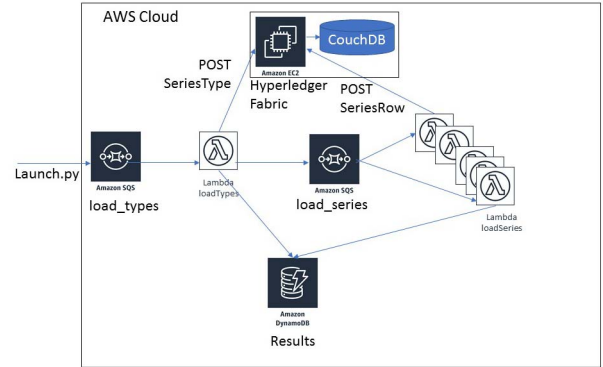

Figure 9.   Our Blockchain benchmarking Architecture

M. Test Scenarios

We have been using several Amazon EC2 instance of different types to compare performance and throughput of our network. Tab. II contains a list of servers we have used.

TABLE II.   AWS INSTANCES USED FOR OUR TESTS

Instance	CPUs	Memory (GB)	Price $ per hour (on-demand)
c3.4xlarge	16	30	0.84
c3.8xlarge	32	60	1.68
c4.4xlarge	16	30	0.796
c4.8xlarge	36	60	1.591

Instance	CPUs	Memory (GB)	Price $ per hour (on-demand)
c5.18xlarge	72	144	3.06
c5.9xlarge	36	72	1.53
g3.16xlarge	64	488	4.56
h1.16xlarge	64	256	3.744
m4.16xlarge	64	256	3.2
m5.4xlarge	16	64	0.768
r3.8xlarge	32	244	2.66
r4.16xlarge	64	488	4.256
r4.8xlarge	32	244	2.128

We have been spawning each of the above instances, using Spot requests (to save over 70% of the price). Our launch template contains the AMI image containing all the necessary software, including Hyperledger Fabric, python and the launch python script to start loading data.
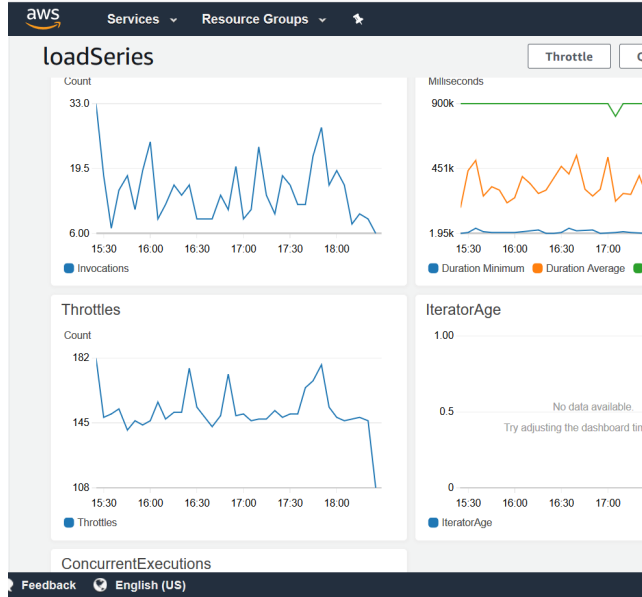

Figure 10. Lambda function monitoring

The data load was taking several hours, and all the timing results were saved into the database. Those results will be analyzed in the next sections of the paper.

We have been monitoring the health of our resources to make sure we do not need to make changes to the DynamoDB WCU, SQS parameters or Lambda concurrency limits. Fig. 10 demonstrates the monitoring of lambda function loadSeries.

IV. RESULTS

After running the tests, it was very easy to use the collected data and query our DynamoDB database to analyze the data. Our experiments have proven that it makes more sense to load data in large chunks. Loading data series row by row heavily depend on the server. Fig. 11 demonstrates

that the average load time is from 1 to 12 seconds depending when we load data record by record. However, for chunk size of one hundred, the instance type does not matter much– the average load time is from 71 to 334 milliseconds.

Blockchain framework and Composer rest server limit the maximum size of the data for the transaction. We were getting the "Request entity too large" error using Hyperledger default parameters.

It turned out that the maximum entity size can be configured in the rest server configuration (the standard file server.js). Although entity size cannot be infinite (we have not found the maximum size in the documentation), the *10000kb* turned out to be sufficient for our datasets. The following code snippet has fixed the problem:

```
app.middleware('parse', bodyParser.json({
  strict: false,
  limit: "10000kb"
}));
```

We cannot have unlimited chunk sizes. However, for our data set, setting the rest server's limit to "10000kb" allowed us to process the data load without size errors.
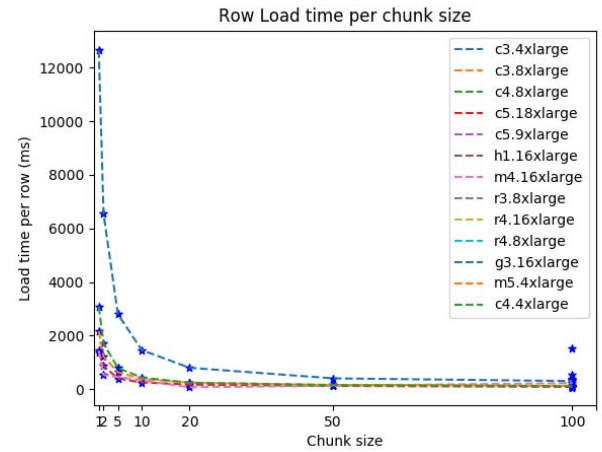

Figure 11. Row load time for different chunk sizes

Average row load time depends on the instance. Fig. 12 demonstrates a different range of average times to load a row into numerous instances. It takes from 54 to 1513 milliseconds to load a row into Hyperledger Fabric in average.

We have discovered that the biggest problem with Blockchain is scalability. During our experiments, up to five hundred and twelve Lambda functions were accessing the Composer rest service to save data into Blockchain (see Fig. 13). We have noticed that the more concurrent Lambda functions are running, the more load errors we are getting. It turned out that no instance can handle that many simultaneous calls to save transactions data. As a result, many HTTP post requests from Fig. 1 were either timing out or failing with the "Failed to connect before the deadline" error.
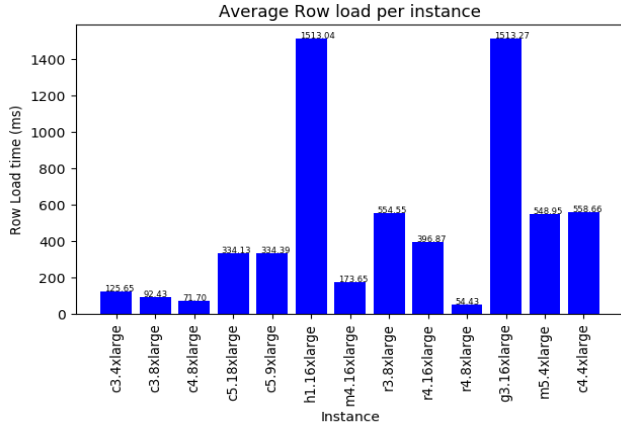
Figure 12. Average row load time per Instance

In order to fix that, we will need to configure additional servers to handle heavy loads. Adding those servers manually is a tedious work. Moreover, servers should be added/removed as a response to a "heavy" or "low" load events or triggers.

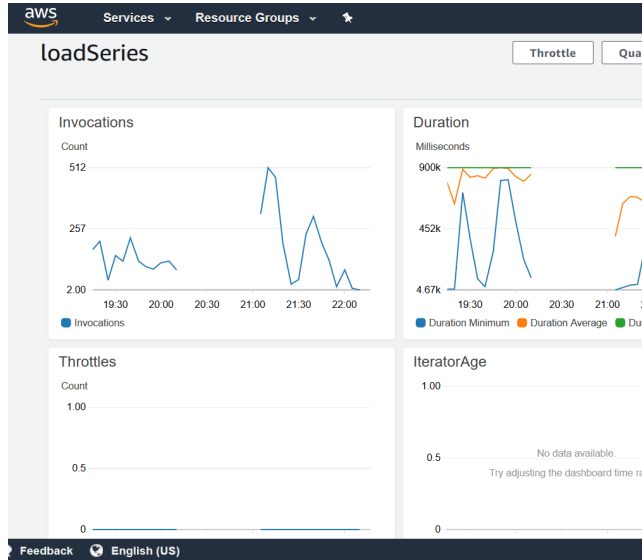As many Cloud services nowadays, Blockchain auto scaling will be a perfect solution.

Figure 13. Lambda function monitoring, over 500 running functions

V. CONCLUSION AND FUTURE WORK

We were able to test the blockchain throughput on numerous servers. It turned out that even the most powerful cloud servers have scalability problem. Our architecture allowed to control the concurrency of the simultaneous requests (using SQS and Lambda synchronization). It turned out that current Hyperledger Fabric Blockchain software cannot handle many simultaneous requests without errors and need auto-scaling to add more servers to process heavy loads.

Blockchain technologies are developing very rapidly. In November 2018, Amazon has announced a Managed Blockchain Service (available for preview only). Such a service can solve installation and scalability problems. We would like to extend our research to that service.

There is a need for auto-scaling Blockchain service that will adapt to the load. Cloud providers already have such services for databases (DynamoDB), map reduce (EMR) and other services. User-friendly auto-scaling implementation for Blockchain will boost the technology usage (as it has boosted EMR in the past).

Another promising area of research – the storage of the Hyperledger data (transactions). Amazon Quantum Ledger Database (QLDB) can standardize and optimize the storage of the Blockchain transactions; improve scalability, reliability and maintenance of the data. It can be used instead of CouchDB as a standard transaction repository.

We believe that blockchain transactions will be used to store and retrieve most of the data in the near future. Especially after scalability problem is solved and blockchain storage is standardized.

REFERENCES

[1] UCR Time Series Classification Archive, Fall 2018, https://www.cs.ucr.edu/~eamonn/time_series_data_2018/

[2] Martin Fowler web site. https://martinfowler.com/articles/serverless.html *(references)*

[3] Maruti Techlabs, What is Serverless Architecture? What are its criticisms and drawbacks?, May 2017,

[4] Hyperledger Composer documentation, https://www.hyperledger.org/projects/composer

[5] Caroline Church "Developing multi-user application using the Hyperledger Composer REST Server", Medium.com, February 2018, https://medium.com//@CazChurchUk/developing-multi-user-application-using-the-hyperledger-composer-rest-server-b3b88e857ccc, last retrieved 1/19/2019

[6] Baldini I. et al. (2017) Serverless Computing: Current Trends and Open Problems. In: Chaudhary S., Somani G., Buyya R. (eds) Research Advances in Cloud Computing. Springer, Singapore

[7] Peter Mell and Timothy Grance, NIST Special Publication 800-145 The NIST Definition of Cloud Computing, 2011

[8] Garrick Hileman, Michel Rauchs, "GLOBAL BLOCKCHAIN BENCHMARKING STUDY", Cambridge Centre for Alternative Finance, 2017

[9] Db-engines.com, "Amazon DynamoDB vs. CouchDB", https://db-engines.com/en/system/Amazon+DynamoDB%3BCouchDB

[10] Jonas Snellinckx, "Hyperledger Fabric & couchdb, fantastic queries and where to find them", September 2017, https://medium.com/wearetheledger/hyperledger-fabric-couchdb-fantastic-queries-and-where-to-find-them-f8a3aecef767

[11] Qassim Nasir et al, "Performance Analysis of Hyperledger Fabric Platforms", Security and Communication Networks, Volume 2018

[12] Peter Sbarski, "Serverless Architectures on AWS", 2017, Manning

[13] Christian Gorenflo,Stephen Lee,Lukasz Golab, S. Keshav, "FastFabric: Scaling Hyperledger Fabric to 20,000 Transactions per Second", 2019, eprint arXiv:1901.00910

[14] "What Is the AWS Serverless Application Model", AWS Documentation, https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/what-is-sam.html

[15] Parth Thakkar,Senthil Nathan,Balaji Viswanathan, "Performance Benchmarking and Optimizing Hyperledger Fabric Blockchain Platform", 2018, eprint arXiv:1805.11390