Final Report - Elastic ML Compute Engine

PS: Our project was showcased in BOOM 2014

Team members

Krishna Sasank Talasila (kt466)
Preetam Mohan Shetty (pms255)
Pritam Bramhadeo Patil (pbp36)
Roshan Shetty Bolar Ramachandra (rb723)
This project is also being taken towards the M.Eng project credit.
Project Advisor: Theodoros Gkountouvas

Introduction

Why?

To enable users to analyze their data using standard Machine Learning algorithms out of the box. So, instead of a user configuring an Amazon EC2 instance with his application code and running his algorithm, we will provide this solution through a simple and configurable dashboard. The user has to simply select the ML algorithm he wants to run through the dashboard, the data on which he wants to run the algorithm and the parameters (if any). The user does not have to write application code from the scratch.

What?

Construct an Elastic Cloud model which will provide "Compute as a Service" (CaaS) as a web- service. For the purpose of this project, we intend to provide some standard machine learning algorithms as the compute services. The user can upload data to our servers or give the location (AWS S3 url). The users will also be able to pipe the output of one algorithm to another. This "cloud model" built will ensure scalability, availability, reliability, fault-tolerance, elasticity and security

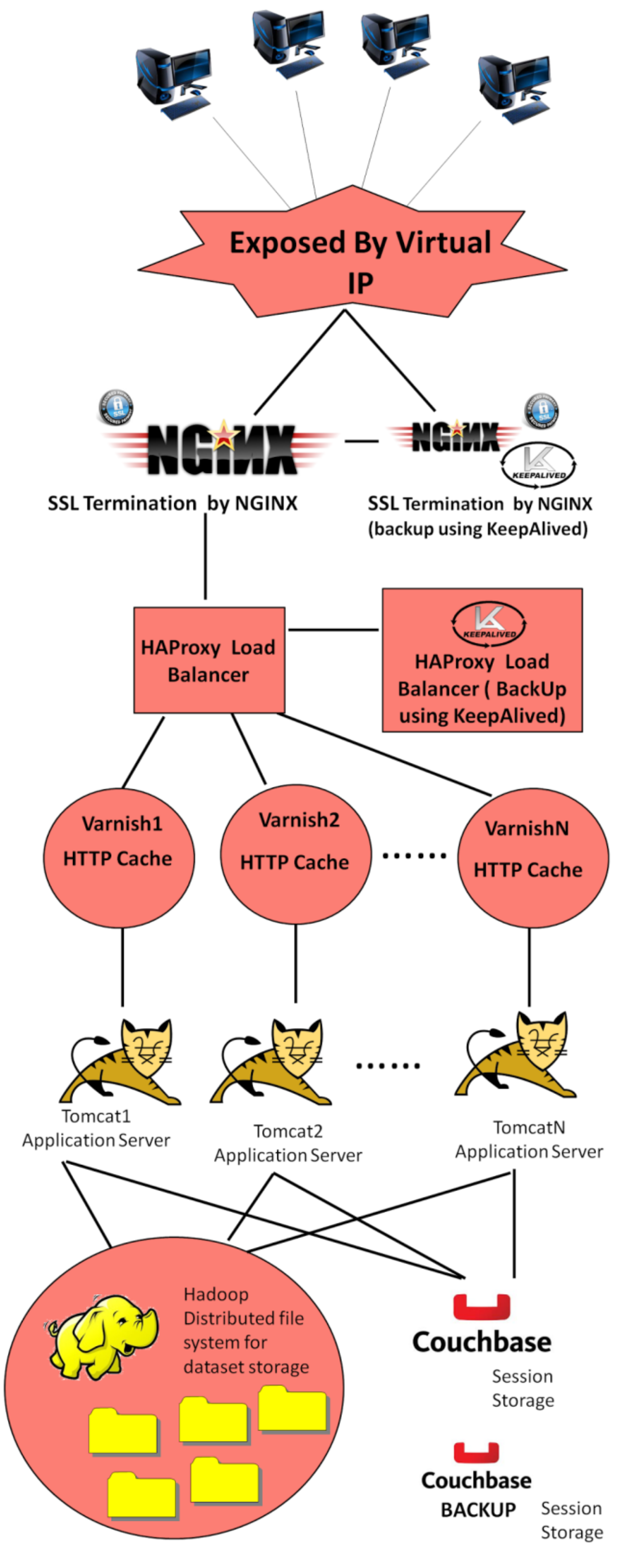
Features

- A variety of machine learning algorithms to choose from via a Dashboard
- Custom datasources like AWS S3 or the user can upload his data to our servers
- User can generate reports and store them. User will be able to export his results as CSV or JSON or even JPEG images of charts
- The user interface delivered both as a web application and as an Android app.
- The system auto scales based on load

System Components

This section broadly describes the high level components present in our system. We have a used a plethora of open-source technologies to complement our system. Details of the tools used along with their customization will be discussed in the next section titled "Implementation". To start off, the below diagram indicates a high-level snapshot of the overall design. The subsequent sub-sections talk about the role of each component sans the actual implementation details.

System Architecture Diagram


Exposed By Virtual IP
SSL Termination by NGINX
SSL Termination by NGINX (backup using KeepAlived)
HAProxy Load Balancer
HAProxy Load Balancer (BackUp using KeepAlived)
Varnish1 HTTP Cache
Varnish2 HTTP Cache
VarnishN HTTP Cache
Tomcat1 Application Server
Tomcat2 Application Server
TomcatN Application Server
Hadoop Distributed file system for dataset storage
Couchbase Session Storage
Couchbase BACKUP Session Storage

Client

The application is known to work with IE 10, Google Chrome 34 and Mozilla Firefox 28 when using the web-based user interface. We use AJAX extensively in our application to enhance user experience. The major components used to design the interface are including but not limited to, JQuery with many plugins, BootStrap , Font Awesome , Flot Charts , HTML5 and CSS. A separate section title "User Interface" talks more about the features.

SSL Terminator

Since we provide our application over HTTPS, we use an SSL termination proxy to handle all the incoming SSL connections, decrypt the SSL and pass on the unencrypted request to the load balancer. We could have also terminated

SSL on the load balancer itself, but we have decided not to do so, to avert the problem of the load balancer becoming a bottleneck. To ensure high-availability we maintain an additional node acting as SSL terminator. Further details in section titled SSL terminator.

Load Balancer

We use a load balancer to optimize resource use, maximize throughput, minimize response time, and avoid overload of any one of the resources. Our load balancer is setup to be non-sticky, which essentially means that any HTTP request can be routed to any of the available application server, regardless of any state. This design decision allowed us to achieve high parallelism. As with SSL terminator, here also we maintain an additional load balancer as backup for high-availability. Further details in section titled Load balancer

Application Server

The application servers run the instance of the application implementing the business logic, which is primarily exposed as a set of RESTful API's. All the application servers run the same copy of the application intercepting client requests routed by the load balancer. To optimize loading of static resources such as CSS, JavaScript...etc. We also install an HTTP accelerator inside each application server. Thus each request first passes through this additional layer before hitting the actual application instance. To achieve fault tolerance, almost little to no state is maintained on the application servers. Each of them also run daemons responsible for achieving fault-tolerance, along with the capability to autoscale based on the load and recover from a crash. Further details in section titled Application Server.

Distributed File System

We use a distributed file system to store large user datasets. This file system is also configured to make it reliable and highly-available.

Session Store

As our load balancer is non-sticky, we need the session state to be "immediately" available to all the application servers, regardless where the session was created. So we have a central replicated and highly-available session store to which all the application servers communicate for session information. This also provides a fault-tolerance mechanism in the form of session failover.

Project Roadmap

Milestone 1 DONE

- Implement the client server model with an app server hosting a basic web service.
- Implementation of few Machine Learning algorithms which would be used for providing the services to the end user. The data-source for computation can live within the server or the user can provide a custom URI.
- Implementation of a basic user interface for register, login, data upload and visualization of results.
- Quality Assurance tests for the web service provided.
  - Correctness of the web services.
  - Verification on Linux(Ubuntu) Platform.
- Basic understanding of the Load Balancers functionality with main focus towards the health check. This would be used for the implementation phase in milestone two.

Milestone 2 DONE

- User profile management
  - User registration and authentication with session management
- Integration of a Distributed file system for persistent storage
  - Integration for Hadoop Distributed File System(HDFS) for persistent storage. Majorly used for storing user datasets and intermediate computation files. We use Hadoop 2.4.0

    - User provisioning: When a new user registers, a new workspace within HDFS is created for him.
    - Investigation of high availability setup for HDFS to circumvent the problem of the NameNode server being the single point of failure.
- ~~Designing load balancer with sticky sessions~~. We are using HAProxy as our load balancer with a non-sticky setup for true parallelization of user tasks/requests.
- Design a SSL terminator to alleviate load from the load balancer. We use Nginx for this.
- Implement a session store to maintain user sessions and state information. This step achieves fault tolerance for our system(further details in a separate section). For this we used Memcached Session Manager in conjunction with CouchBase and Javolution.

MileStone 3 DONE

The deadline for this milestone is April 20th. The tasks under this milestone are:

- Fault-tolerance and achieving parallelism (further details in a separate section)
- Add support for AWS S3 datasets and any custom HTTP url
    - User should be able to upload datasets which are located in either AWS S3 buckets or any other HTTP url.
- User interface enhancements
    - Ability to monitor current tasks. Statuses of the tasks will either be SUCCESS, RUNNING or FAILURE.
    - Notification when the status of a task changes. A small red popup(just like Facebook) will show up with the relevant information.
    - Report management
        - A separate page where the user will be able to visualize all his generated reports based on the algorithms he ran.
        - Ability to export the report as raw text or as an image file.
- Elasticity (further details in a separate section)
    - Autoscale application servers based on load
- Android UI
    - We are building up an Android user interface to complement our web application. More details in section titled Android UI

MileStone 4(For final demo) DONE

The demonstration of our project is scheduled to be on May 6th. We did a feature freeze on May 1st and have been testing since then.

- User Interface testing
- Fault-tolerance testing
- Elasticity testing.

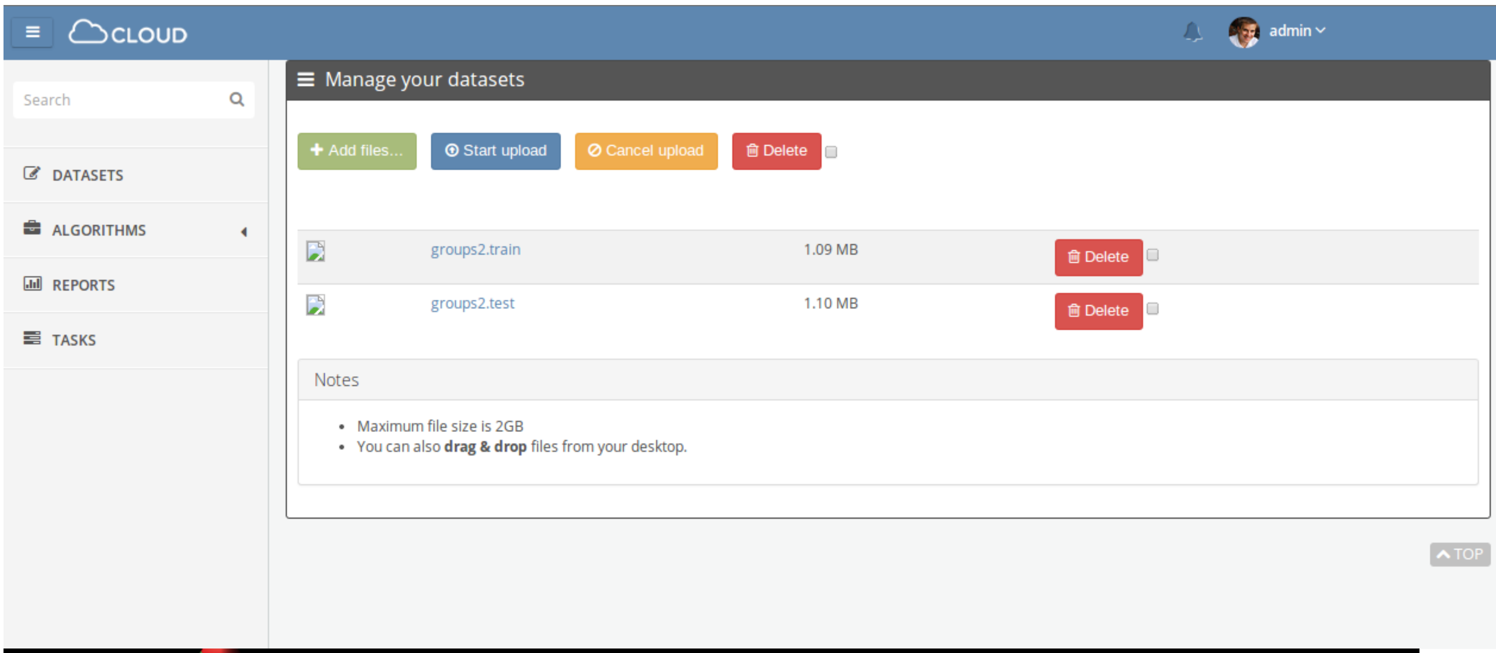
Implementation

User Interface

We have a web-application which the clients shall use to consume our infrastructure. The interface is designed to be responsive, i.e. it works on both desktop and handheld devices. The major features of this interface are:

- User profile creation with login/logout mechanism
- DashBoard showing the status of the tasks being run and the reports generated
- File upload feature with a display of current uploaded files
- Real Time notification of status updates

The below screenshots show the dataset upload page and the dashboard with two tabs, one for tasks and the other for the reports. Also notice the top red icon with a number. This is somewhat a similar notification which we generally see on Facebook. This element has been written using a traditional pull notification service.



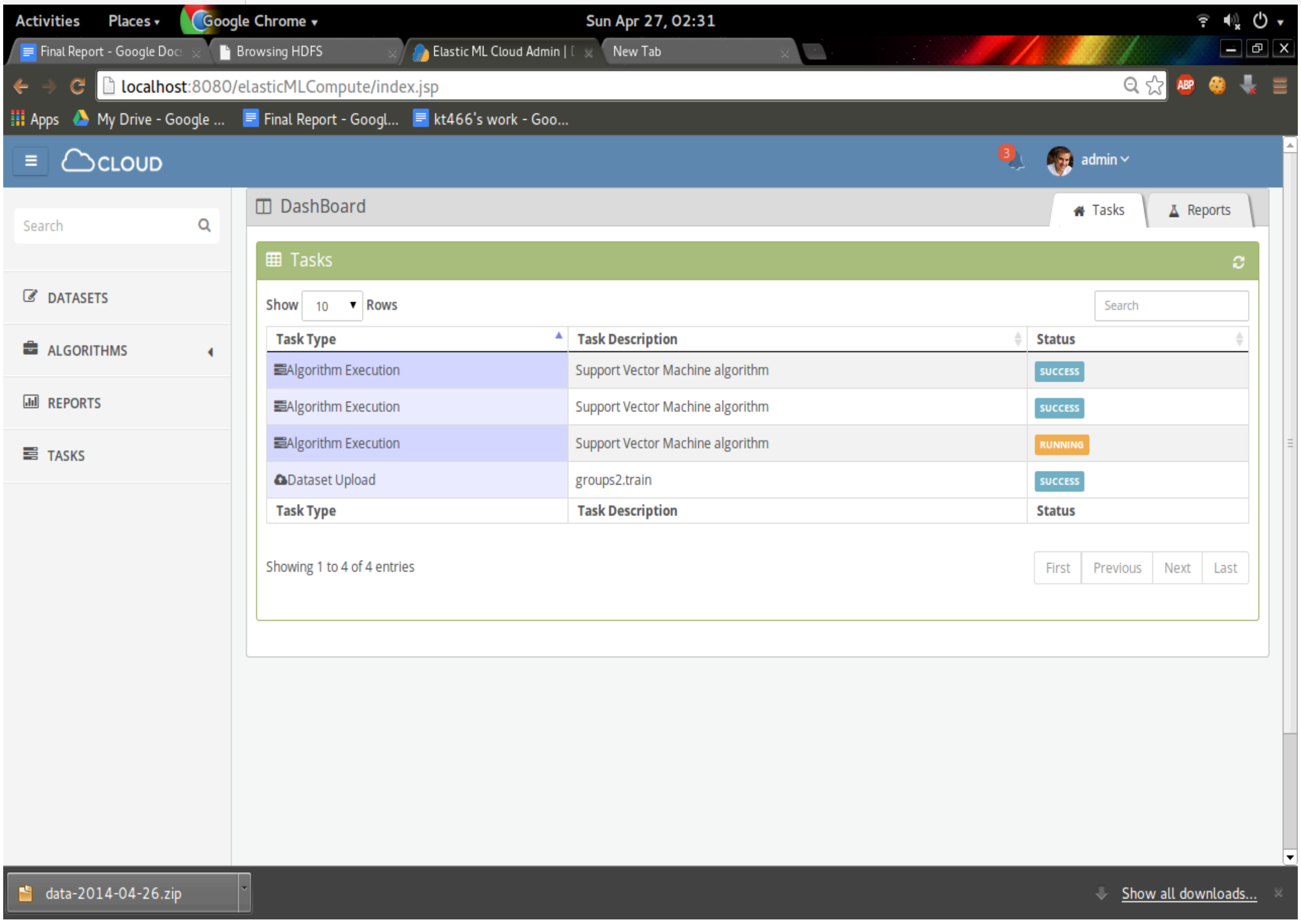
Manage your datasets

Add files... | Start upload | Cancel upload | Delete

	groups2.train	1.09 MB	Delete
	groups2.test	1.10 MB	Delete

Notes

- Maximum file size is 2GB
- You can also **drag & drop** files from your desktop.



DashBoard

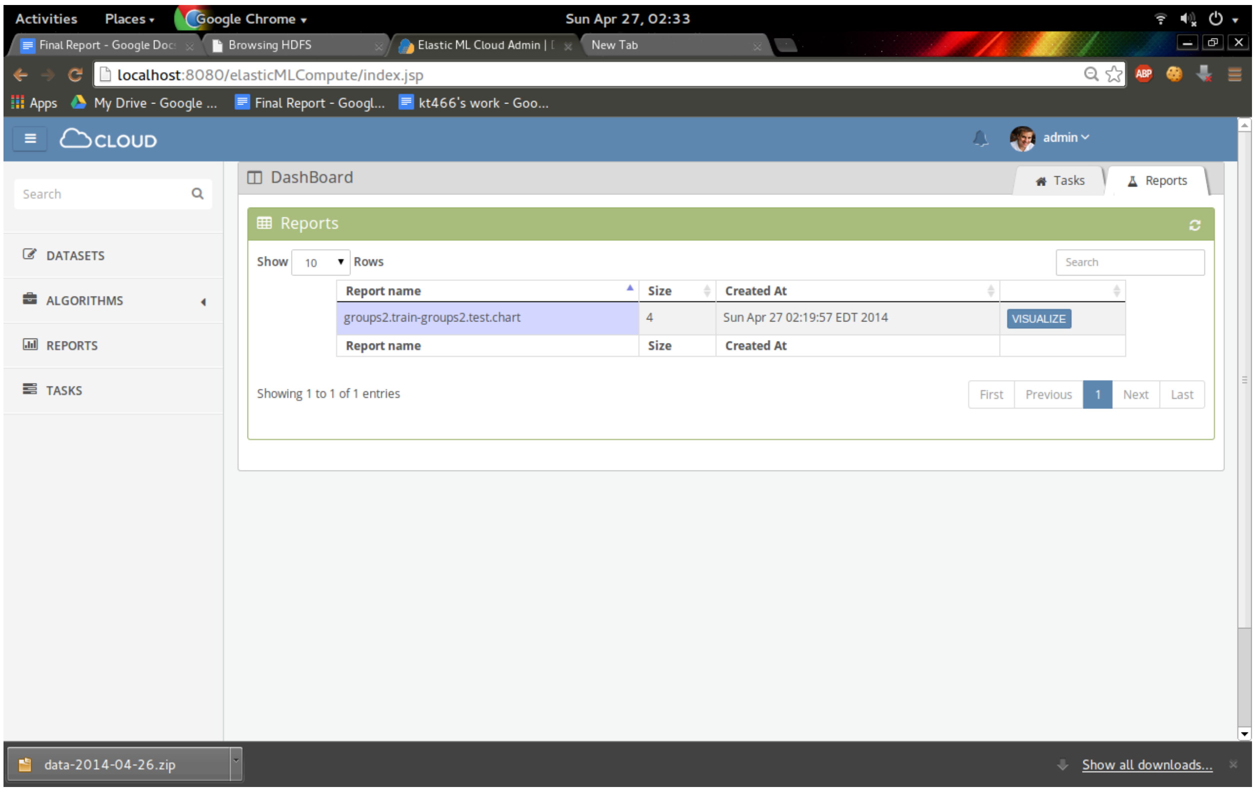
Tasks

Show 10 Rows

Task Type	Task Description	Status
Algorithm Execution	Support Vector Machine algorithm	SUCCESS
Algorithm Execution	Support Vector Machine algorithm	SUCCESS
Algorithm Execution	Support Vector Machine algorithm	RUNNING
Dataset Upload	groups2.train	SUCCESS
Task Type	Task Description	Status

Showing 1 to 4 of 4 entries

First | Previous | Next | Last



Report name	Size	Created At	
groups2.train-groups2.test.chart	4	Sun Apr 27 02:19:57 EDT 2014	VISUALIZE
Report name	Size	Created At	

Showing 1 to 1 of 1 entries

SSL Termination

On the top of our architecture stack, sits the SSL Termination proxy. We have implemented a Nginx server to do this job. This guide as acted the primary reference for this purpose. We generate a self-signed certificate using the OpenSSL library. Since we are using the systems in the M.Eng Lab, we have tweaked its configuration to complement the available hardware. We have 8 core, 16GB RAM machines. The base operating system is Ubuntu 12.04.4 LTS. A backup nginx server is also setup using keepalived for high availability.

Load Balancer

Behind the SSL terminator, sits the load balancer. We have used HAProxy as our load balancer. It's widespread usage and even by Amazon, led us to this choice. The solution itself is highly configurable and operates very efficiently. To complement its capabilities, we accordingly tune the operating system(Ubuntu). We tune the sysctl settings and enable tcp-splicing for performance. In addition we also run an SSH server and install socat to facilitate autoscaling described in this section. The setup is non-sticky and use "Least Connection" strategy while load balancing, i.e the server with the lowest number of connections receives the connection.

Application Server

Our application servers run Ubuntu as the base OS. They receive the HTTP connections routed by the load balancer. It has two components, Apache Tomcat v 7.0.53 as the web server and a HTTP cache, Varnish v3.0.5 which sits on top of it. All the dynamic requests are served by Tomcat and all the static requests like CSS,JS..etc are served by Varnish. This decoupling alleviates some load off the tomcat server. We use Oracle JVM. To increase performance, we so some JVM tuning and install the APR native library

Distributed File System

We implement Hadoop Distributed File System to serve as our highly available persistent storage component. The application servers use this component to store user datasets and intermediate files that may be generated during a computation.

Session Store

The application servers use this component to retrieve user session information. This has been using memcached-session-manager backed by a highly available CouchBase server. We use this guide to modify the tomcat container to complement this. We also use this component to store the task context information to complement fault tolerance described in the next section.

Fault Tolerance And Parallelism

Design goals:

To provide 99.9% fault-tolerance to our system. The major sub-goals are

- ✓ Ensuring every task reaches its completion in the event of multiple server failures
- ✓ Making application servers completely stateless - To simplify server failures
    - ➔ Identifying the failed tasks
    - ➔ Reassigning the task to the live server nodes
    - ➔ Updating the new assignment information in the session store
- ✓ Restarting the failed server node in the event of a crash

Scenarios leading to a task failure

- ➔ Server crashes due to an internal error/exception
- ➔ Task execution is interrupted due some exception, for e.g. “Out of Memory” error

Task Management - Fault tolerance - Background:

A **Task** here is a single unit of execution in our infrastructure. A user request may result in one or more tasks. The task at any point of time can be in one of the following states: RUNNING, FAILURE, INITIALIZED. When an application server receives a HTTP request from a user, based on the nature of the request, it constructs several other HTTP requests. These newly constructed requests are again routed through the load balancer for their appropriate assignment to application servers. The initial app server is the owner of this master task (the first task, which spawned other subtasks). To achieve fault tolerance, we follow a “Log Before Execution (LBE)” workflow for every task that will be executed.

Statelessness of the Application Servers

To simplify server failures, we maintain little to no state on the application servers. All the state information is stored in a highly available and scalable NoSQL store (CouchBase). The below section explains this further.

Log Before Execution (LBE)

When an app server picks up the task, before it executes it, it logs the information about this task to Couchbase. This is done as follows:

- ➢ When a client serves an HTTP request, it hits the primary service called *runDistributedService*. This service runs for every Machine Learning algorithm in every machine. The primary task of this service is to divide the client task into independent subtasks, store it in the couch database and then run the subtasks.

- ➢ We build Task objects for each task. This object contains vital information about the task. Some of which are described below:

  a) UserId: The user who fired the task.
  b) WebServiceURL: The web service url this task is supposed to run.
  c) HostAddress: The host address this task is currently running in.
  d) Status: The current status of the task: *RUNNING*, *FAILURE* or *INITIALIZED*.

e) isParent: Is this task the main task fired by the client or is it a subtask.

The task objects are serialized and placed in the couch database. Each task object has a unique task id. The key in the CouchBase is this task_id and the value is the serialized task object.

- When the initial HTTP request first hits the runDistributedService webservice URL for the corresponding ML algorithm in some app server, we create a Task object for the request and mark its status as *RUNNING*. This is then serialized and placed back in CouchBase.

- Along with the <Task_Id, Serialized_Task_Object> pairs in the CouchBase, we maintain one more entry: <"AllTaskIds", ArrayList<String> taskIds>. AllTaskIds is the constant key. The value is an array list of task ids. This entry is stored in serialized fashion.

- We then identify all the subtasks to be executed from the main task. Each subtask may contain its own subtasks which will be ***parallelized***.

- ***Achieving Parallelism***: Consider the task T broken down as following subtasks:

T1, T2, T3, T4. These subtasks are dependent.
T1 -> T2 -> T3 -> T4. Which means T1 is an independent task. T2 depends on T1 and so on.
Let subtask T2 contain its own subtasks:
T21, T22, T23, T24.................... These subtasks are independent and can be parallelized.
To attain above dependencies, we store one more data structure inside the task object called *parentTaskIds*. This is a list.

a) Task T1 has empty *parentTaskId* list: [ ]
b) Task T21, T22, T23, T24..... each have *parentTaskId* list: [T1]
c) Task T3 has *parentTaskId* list: [T21, T22, T23, T24........]
d) Task T4 has *parentTaskId* list: [T3]

We construct Task objects for each of the above subtasks and store in the couch database.
The status of each of the task is set to *INITIALIZED*.
Once all the subtasks have been recorded in the CouchBase, we set isParent attribute of the parent task T to *True*. This is to signify that this task T is no longer of any use since we have identified and stored the necessary subtasks.
The machine learning web service is written such that we have RESTful API's exposed to run any of the recorded subtasks. From the runDistributedService API, we construct a HTTPUrlConnection for each of the subtasks and send the http request to the load balancer. We make sure that there internal RESTful are not exposed on the public domain. The load balancer submits this request to any of the available application servers which have the RESTful API to execute this request.
Once the RESTful API exposed in some application server receives this task, it sets its status to *RUNNING* and the tasks HostAddress to the current application servers host address.
Whenever a task completes successfully, its status is set to *SUCCESS*.
Note: Since the tasks T21, T22, T23, T24..... are independent, the requests for them are sent simultaneously(asynchronously) to the load balancer using *AsyncClientHttp.* This way we achieve parallelism.

- ***API failure***: If the task execution fails due to Out Of Memory issue or some other exception, the exception is caught and the tasks status is set to *FAILURE*.

- ***Application Server Failure***: Consider a Task with status *RUNNING*. Suppose, before the task completes or fails (due to API failure) and its status is set to *SUCCESS* or *FAILURE,* the application server fails. It is now important to understand that the server has failed but the task status is still *RUNNING*.

To handle this situation, we have a daemon task running on (around 30% of) application servers. This daemon gets the ArrayList of task Ids from the CouchBase using the key: "AllTaskIds". For each of the task ids whose status is either *RUNNING* or *INITIALIZED*, it pings the host address where it was running. If the ping fails, the status of the task is set to *FAILURE*.

Handling immediate failed server restart: Suppose the app server which failed is brought up immediately before the polling happens, then the polling would give wrong information. This is because, the ping would be successful and we would assume that the Task is still running but actually it was a server restart and the Task status should be changed to *FAILURE*. To avoid this problem, we maintain one more attribute: ***hostVersion*** in each task object. When a server starts for the first time, a servlet listener adds an entry in the CouchBase: <Host_IPAddress, hostVersion>. On the first server start, this value would be: <Host_IPAddress, 0>. On further server restarts, the hostVersion is incremented.
Using this we avoid the immediate failed server restart problem as follows:
For each of the task ids whose status is either *RUNNING* or *INITIALIZED*, it pings the host address where it was running (stored in the task object). If the ping fails, the status of the task is set to *FAILURE*. If the ping succeeds, we compare the hostVersion of the task stored in the task object with the host version of the host machine stored in the CouchBase. If the values are different, the status of the task is set to *FAILURE*.
Note: If first ping to the application server fails, we ping it again to avoid false positives.

➢ ***Fault tolerance in action***: We have one more daemon task running on around 80% of the application servers. This daemon task gets the ArrayList of task Ids from the couch database using the key: "AllTaskIds". For each of the task ids whose status is *FAILURE* and the *parentTaskIds* list is empty, it constructs a HTTPUrlConnection and sends this request to load balancer which redirects it to some application server.

Example: Consider the following **failed tasks** stored in the couch database:

a) Task T1 has empty parentTaskIds list: **[ ]**
b) Task T21, T22, T23, T24..... each have parentTaskIds list: [T1]
c) Task T3 has parentTaskIds list: [T21, T22, T23, T24........]
d) Task T4 has parentTaskIds list: [T3]

First polling by the daemon task picks task T1 since it is failed and its parent list is empty which signifies it can be executed independently. Once this task completes successfully, all the tasks which contain T1 as an entry in the parent list is removed. Hence once T1 completes successfully, following are the failed task entries in the couch database:

a) Task T21, T22, T23, T24..... each have parentTaskIds list: **[ ]**
b) Task T3 has parentTaskIds list: [T21, T22, T23, T24........]
c) Task T4 has parentTaskIds list: [T3]

During the next polls, the daemon task can pick any of the tasks T21, T22, T23, T24.... since they have empty parent list and can be executed independently.
This way all the failed independent tasks are picked up in intervals and submitted to application servers for execution until no failed tasks remain.
This infrastructure is hence capable of handling up to 99.9% failure.

Garbage collection of successful task objects

Since the number task objects in the couchbase store will be growing exponentially as the service executes, each application server will also be running a daemon task to clean up those task status objects which were fully completed(this includes all its sub-tasks). This daemon task gets the ArrayList of task Ids from the couch database using the key: "AllTaskIds". For each of the task ids whose status is *SUCCESS* or if the tasks isParent attribute is set to true, it is removed from the list. The updated list is placed back in the couch database.

Supported Machine Learning algorithms

We currently support the below list of algorithms:
➔***Decision Tree:*** Decision tree learning uses a decision tree as a predictive model mapping observations about an item to conclusions about the item's target value. It is one of the predictive modelling approaches used in Machine

learning. In our implementation we use cross validation to find the best optimum height for classification. Early stopping is employed at different heights to find the best height which avoids overfitting.

➔***Support Vector Machine(SVM):*** SVMs are supervised learning models with associated learning algorithms that analyze data and analyze patterns used for classification and regression analysis. The goal in this model is to find the optimal hyperplane which divided the training data with maximum possible geometric margin to perform binary classification. We experiment with different tradeoff parameters (c value) for cross validation to find the best optimum c value which will be used for text classification.

➔ ***Kernels:*** This model is an extension to the support vector machines. SVM's work best when the data is linearly separable. However, the training data available may not always fit this bill. In such scenarios we allow user to experiment with different kernels (polynomial kernels, radial basis functions and sigmoid functions). This technique is used for mapping the data from the kernel space to the user space. Although this approach takes a lot of time (due to the increase in the dimensions), this is a popular approach to get better test accuracies.

➔ ***K-Nearest Neighbor:*** This algorithm is a non-parametric method used for classification and regression. For each test data we find the k nearest neighbours from the training data using certain similarity metrics. In our implementation we have used cosine similarity metric. We experiment with different values of k during the cross validation step to find the best optimum k value which is finally used for classifying the test data.

➔ ***Word Sense Disambiguation:*** Word sense disambiguation is a Natural Language Processing technique to disambiguate the sense in which words are used based on its context. It uses co-occurrence and collocational features to disambiguate the target word. The selection of features influences the effectiveness of the process which in turn depends upon the window sizes for aforementioned features. Therefore we experiment with different values of window sizes for collocational and co-occurrence features in our implementation in order to find respective optimum values which in turn are used for disambiguating word senses of test data.

➔ ***Naive Bayes Classifier:*** This is a generative classification machine learning technique. In this approach, the training involves storing the word counts for different training labels and also the prior probabilities. The prediction step involves calculating the probabilities of the test data belonging to a given label assuming the test data to be a bag of words which are independent. The test data is classified with the label which gives the maximum probability. This technique differs from the above approaches in that we don't initially create a model for classification purpose.

Autoscaling with Crash prevention/recovery

Design goals

To adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible.

How

Autoscaling with HAProxy

HAProxy exposes a unix socket which takes in commands to facilitate dynamic addition/removal of servers from the "active" pool. A daemon task has been written which monitors the current operating load on the application server. Based on the request load, the servers are scaled up or down based on a "scaling trigger". We basically establish an SSH tunnel between the load balancer and application servers to perform the below steps.

The scaling trigger in our case takes into account the current RAM usage. If an application server is operating beyond a certain "threshold" value of these parameters, the server is made to deregister itself from the application server. The daemon task running on this app server sends the below UNIX socket command to the socket on HAProxy load balancer

```
echo "disable server servers/node2" | socat stdio /var/run/haproxy.stat
```

Here, *node2* is the name of the application server and /var/run/haproxy.stat is the unix socket on the load balancer. This is just a sample script where the load balancer and the application server are on the same node. The actual action involves establish an SSH session with the load balancer the sending the socket command over the network.

This command also allows connection draining. This mean it will drain the connections (existing connections to this server will persist, but no *new* connections will be made to it). Thus all existing requests in progress will reach completion

Similarly, once the operating load goes down a certain threshold, we send the below command to re-register itself to the load balancer.

```
echo "enable server servers/node2" | socat stdio /var/run/haproxy.stat
```

Crash Prevention/Repairing

By far we observed "Out of Memory" to be the most common case of an app server crash. Hence, as described above we have a daemon task running on each of application to take care of this. Also, as we mentioned, some of application servers run a poller daemon, which updates the statuses of Task objects pertaining to the crashed servers, in the CouchBase. We just piggyback on this poll request. If we find that server is crashed by its poll response, we try to bring up the server.

Android UI

Implementation of an Android based application which will be developed on Android 4.2 and which will be backward compatible upto versions 2.3. The Android app is going to be a simple UI based application which will be used for connecting to the server. The app is going to have an initial login page which will show his workspace showing all the files he has in the HDFS stored corresponding to him. This will call the web service for authentication.

On login the app is going to give the options for:-

1. Uploading the file.
2. Choosing the S3 file location and uploading it.

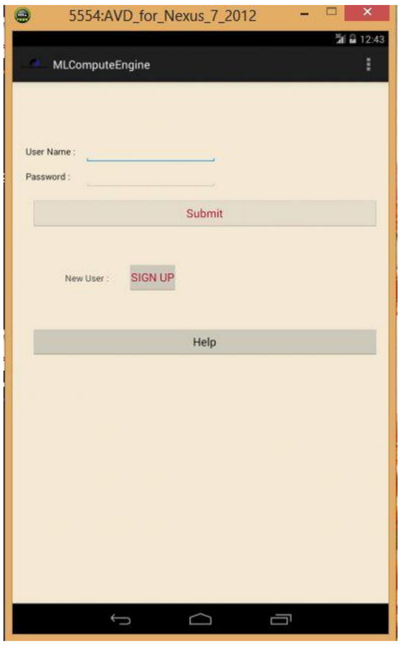
The above options are provided using the responsive UI.

The user can then choose the algorithms from the list mentioning the long list of algorithms.

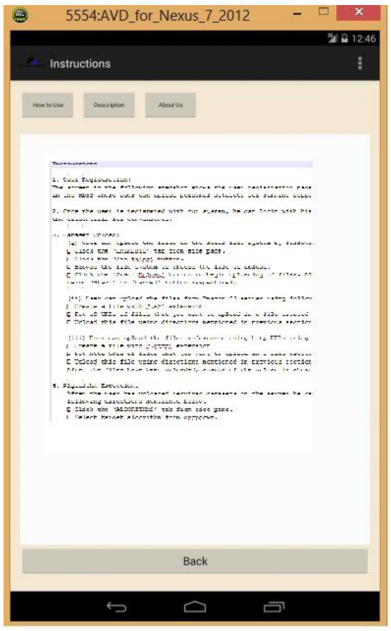
Then the user chooses the files and he can run the algorithms by setting different parameters and then generate the report. The user can share the generated reports as images.

In addition to these pages the Android UI provides an UI page for instructions, usage and the about us page.

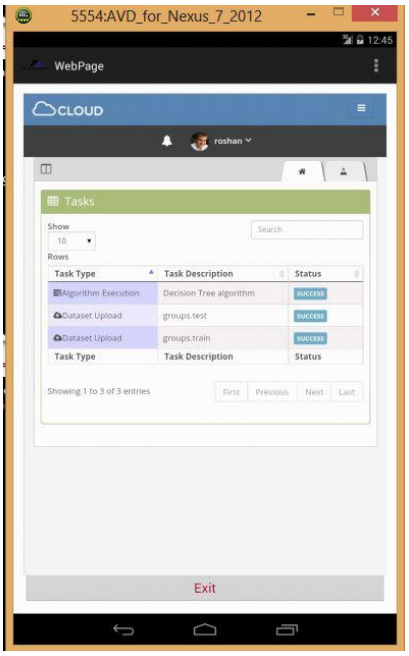
The android UI is highly intuitive with proper instructions with appropriate pop ups.



Login Page



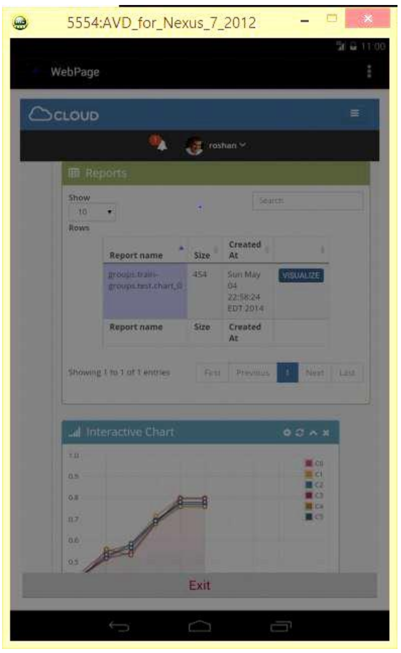
Instruction Page



Task Type	Task Description	Status
Algorithm Execution	Decision Tree algorithm	SUCCESS
Dataset Upload	groups.test	SUCCESS
Dataset Upload	groups.train	SUCCESS
Task Type	Task Description	Status

Showing 1 to 3 of 3 entries

Logged in Page



Report name	Size	Created At	
groups.train-groups.test.chart_0	454	Sun May 04 22:58:24 EDT 2014	VISUALIZE
Report name	Size	Created At	

Showing 1 to 1 of 1 entries

Chart Generation

Performance evaluation

We have used Apache JMeter to create custom test cases for system evaluation and for generating various performance charts. We performed both load and stress testing. As the system is designed to auto-scale, we also do the scalability testing.
Due to lack of space, we have not attached the charts here; we will be showing the charts in the demo.

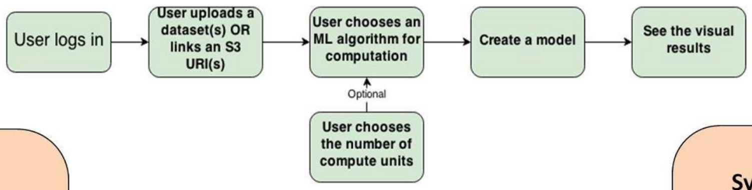
Project Demonstration

- Showing all the actions in the user-interface, both web and Android.
- Demonstrate fault-tolerance by shutting down random servers
- Demonstrate elasticity by autoscaling (up or down) the current pool of servers based on the load.

Poster

ML ComputeEngine

Computation as a Service

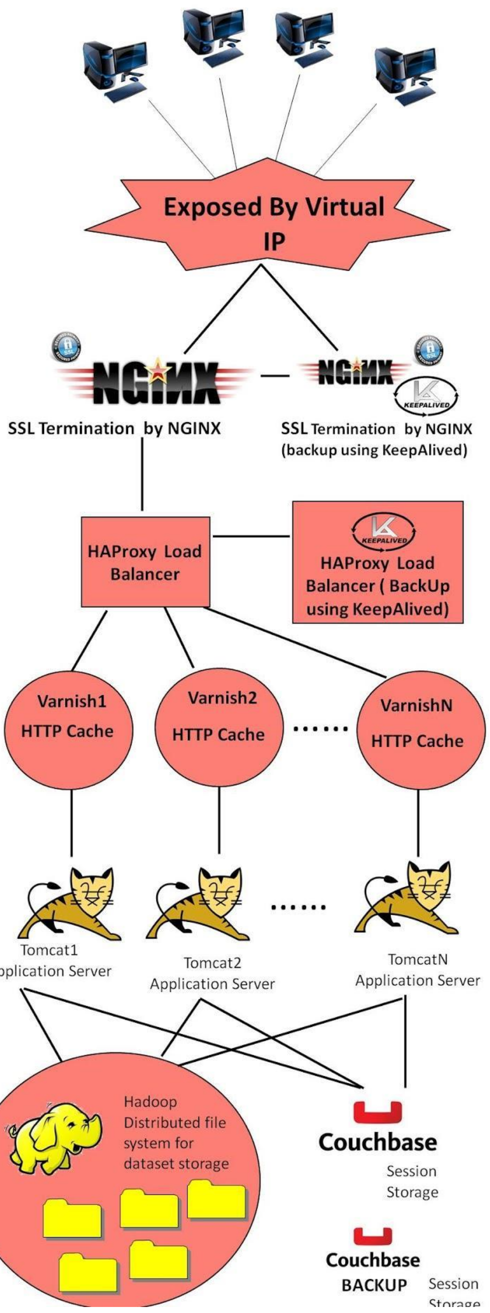

User logs in
User uploads a dataset(s) OR links an S3 URI(s)
User chooses an ML algorithm for computation
Create a model
See the visual results
Optional
User chooses the number of compute units

Why?

•To provide an easy interface to run machine learning algorithms on the cloud.
•User can choose from a wide variety of machine learning algorithms.
•Make Machine Learning usable by all.
•User can run their computations faster by just subscribing to extra compute units- Elasticity!

System Features

• **Elasticity**
User can speed up his computation by subscribing to more compute units.
• **Fault Tolerance**
All user issued tasks will reach completion with 99.9% probability.
WAL maintained in CouchBase which handles failed node computations.
• **High Availability**
99.9% System uptime
• **HTTPS Enabled**
Your data is secure
• **Dataset** can be uploaded or can be a AWS S3 bucket URI


Exposed By Virtual IP
SSL Termination by NGINX
SSL Termination by NGINX (backup using KeepAlived)
KEEPALIVED
HAProxy Load Balancer
HAProxy Load Balancer (BackUp using KeepAlived)
Varnish1 HTTP Cache
Varnish2 HTTP Cache
VarnishN HTTP Cache
Tomcat1 Application Server
Tomcat2 Application Server
TomcatN Application Server
Hadoop Distributed file system for dataset storage
Couchbase
Session Storage
Couchbase BACKUP
Session Storage

System Components

➢ **Nginx**
We use Nginx as an *SSL termination* proxy. This means all the https decryption/ encryption happens here, thus alleviating the load balancer from this task.

➢ **HAProxy**
HAProxy is used as the load balancer. Many famous companies like Quora and Amazon use it.

➢ **Varnish Cache**
Varnish is deployed on top of our Tomcat app servers for caching static HTTP content. This speeds up the performance of the application.

➢ **Tomcat 7**
We use Apache Tomcat 7 as our application server. The web application has been built using the RESTful architecture using JAX-RS spec (Jersey).

➢ **Hadoop Distributed File System (HDFS)**
We use Apache HDFS for large data storage needs. We store user datasets in HDFS. Our HDFS configuration has been made highly-available by having a back-up Namenode server.

➢ **CouchBase**
To ensure fault-tolerance, we make all our Tomcat instances to store/retrieve session information from a couchbase (NoSQL datastore) backed store.

Machine Leaning Algorithms

• **SVM** – Classify test data with high accuracy performing 5 fold cross validation in parallel.
• **KNN** – Classify document data with very good accuracy performing 5 fold cross validation optimizing on 'k'.
• **Random Forest** – Perform 5 fold cross validation optimizing for different heights of the tree and calculate the best accuracy.
• **Naïve Bayes** – Simple but effective algorithm for text classification.
• **Kernel** – Perform polynomial degree classification making use of SVMLite.