Title: Developing Elastic Software for the Cloud

Author Information:
- Author 1*:
  - Name: Shigeru Imai
  - Affiliation: Department of Computer Science, Rensselaer Polytechnic Institute 110 Eighth Street Troy, NY 12180, USA
  - Email: imais@rpi.edu
- Author 2:
  - Name: Pratik Patel
  - Affiliation: Department of Computer Science, Rensselaer Polytechnic Institute 110 Eighth Street Troy, NY 12180, USA
  - Email: patelp8@rpi.edu
- Author 3:
  - Name: Carlos A. Varela
  - Affiliation: Department of Computer Science, Rensselaer Polytechnic Institute 110 Eighth Street Troy, NY 12180, USA
  - Email: cvarela@cs.rpi.edu

Keywords: cloud programming, MapReduce, Hadoop, Google App Engine, actors, SALSA

Summary:
Cloud computing provides great opportunities for application developers to create scalable and elastic applications in a cost-efficient manner. More specifically, well-designed cloud computing services offer not only scalability and elasticity, but also fault tolerance, data locality, load balancing, and pricing models associated with various QoS policies. Since cloud providers take care of these non-functional concerns, the application developers can focus on the functional aspects of their applications and benefit from the power of cloud computing. From the application developer’s perspective, programming models and frameworks to develop cloud applications largely depend on the layered architecture of the cloud computing paradigm, *i.e.*, whether the application is to be deployed on IaaS (Infrastructure as a Service) or PaaS (Platform as a Service) clouds. In this chapter, we describe *Google App Engine* as an example of a PaaS programming framework beside *MapReduce* and *SALSA* (Simple Actor Language System and Architecture) as examples of distributed computing frameworks available on PaaS and IaaS. To give concrete usage
examples of these frameworks, we show a data analytics application using MapReduce, a web application using Google App Engine, and a distributed computing application using SALSA respectively.
1 INTRODUCTION

Developing standalone applications running on a single computer is very different from developing scalable applications running on the cloud, such as data analytics applications that process terabytes of data, web applications that receive thousands of requests per second, or distributed computing applications where components run simultaneously across many computers. Cloud computing service providers help facilitate the development of these complex applications through their cloud programming frameworks. A cloud programming framework is a software platform to develop applications in the cloud that takes care of non-functional concerns, such as scalability, elasticity, fault tolerance, and load balancing. Using cloud programming frameworks, application developers can focus on the functional aspects of their applications and benefit from the power of cloud computing.

In this chapter, we will show how to use some of the existing cloud programming frameworks in three application domains: data analytics, web applications, and Distributed Computing. More specifically, we will explain how to use MapReduce (Dean & Ghemawat, 2008) for data analytics, Google App Engine (Google Inc., 2014) for web applications, and SALSA (Varela & Agha, 2001) for Distributed Computing. The rest of the chapter is structured as follows. In Section 2, we describe non-functional concerns supported at different levels of cloud services and go through existing cloud programming frameworks. In Section 3, we explain MapReduce, Google App Engine, and SALSA (Simple Actor Language System and Architecture). In Section 4, we illustrate how to use these three programming frameworks by showing example applications. Finally, we conclude the chapter in Section 5.

2 PROGRAMMING FOR THE CLOUD

2.1 Non-Functional Concerns

Figure 1 illustrates how cloud programming frameworks hide non-functional concerns from application developers by providing programming languages or Application Programming Interfaces (APIs) to manage the application’s execution on different cloud service models, such as IaaS and PaaS.
Figure 1: Cloud programming frameworks take advantage of layered services by exposing an Application Programming Interface (API) to developers.

Non-functional concerns are not directly related to the main functionality of a system, but guarantee important properties such as security or reliability. In the context of cloud computing services, important non-functional concerns include the following:

- **Scalability**: the ability to scale up and out computing resources to process more workload or to process it faster as demanded by cloud users.
- **Elasticity**: the ability of an application to adapt in order to scale up and down as service demand grows or shrinks.
- **Fault tolerance**: the ability to keep the system working properly even in the event of a computing resource failure.
- **Load balancing**: the ability to balance the workload between heterogeneous networked computing resources.
If not using cloud computing, application developers would have to acquire physical infrastructure (machines, networks, etc.) to support their needs. Using IaaS, developers can create virtual machines (VMs) without up-front cost for hardware; however, they have to install and configure the VM management software, networking, operating system, and any additional libraries that their application needs. Finally, different PaaS providers offer from transparent scalability and elasticity, to transparent fault tolerance and load balancing services. Typically, these non-functional concerns are offered constraining developers to specific programming models and patterns, e.g., MapReduce, web applications, and distributed actor computations.

2.2 Overview of Cloud Programming Frameworks

**Data Analytics:** MapReduce is a popular data processing framework created by Google following a data-parallel programming model based on the *map* and *reduce* higher-order abstractions from functional programming. Hadoop (Apache Software Foundation, 2014) is a popular open-source implementation of MapReduce and has a large user and contributor base. There are a number of projects derived from Hadoop. One such project is Pig (Apache Software Foundation, 2014), which was first developed by Yahoo!. Pig offers a high-level language called Pig Latin to express data analytics programs. The programs authored in Pig Latin are translated into Java-based Hadoop code and thus the Pig users benefit from Hadoop’s scalability and fault-tolerance properties as well as Pig Latin’s simplicity. Another Hadoop related project is Hive (Apache Software Foundation, 2014) initially developed by Facebook. Hive is a data warehouse platform built on top of Hadoop. Mahout (Apache Software Foundation, 2014) is a set of scalable machine learning libraries that also works over Hadoop. Spark (Apache Software Foundation, 2014) is a cluster computing framework that focuses on efficient use of data objects to speed up iterative algorithms such as machine learning or graph computation. Whereas Hadoop reads (writes) data from (to) storage repeatedly, Spark caches created data objects in memory so that it can perform up to 100 times faster than Hadoop for a particular class of applications. Shark (Xin et al., 2013) is a Structured Query Language (SQL) query engine that has compatibility with Hive and runs on Spark. These frameworks do not support elastic behavior of applications, but have good scalability, load balancing, and fault tolerance.
Web Applications: Since web applications have unpredictable and varying demand, they constitute a very good match to cloud computing. There are several PaaS providers that offer a framework for developing and hosting web applications, such as Google App Engine, Microsoft Azure (Microsoft, 2014), and Heroku (Heroku, 2014). Google App Engine supports Python, Java, PHP, and Go. The runtime environment for Google App Engine is restrictive (e.g., no socket use and no file system access), but you get good scalability with a few lines of code. In contrast, Microsoft Azure offers a more flexible runtime environment; however, it requires you to write more code. Microsoft Azure supports .NET, Java, PHP, Node.js, and Python. Heroku originally supported only Ruby, but now supports Java, Node.js, Scala, Clojure, Python, and PHP. In terms of coding flexibility, Heroku is also as restrictive as Google App Engine; however, it has libraries to support data management, mobile users, analytics, and others.

Distributed Computing: Distributed computing systems have typically components which communicate with each other via message passing to cooperatively solve large or complex problems. Erlang (Armstrong et al., 1993) and SALSA are concurrent and distributed programming languages based on the actor model (Agha, 1986), in which each actor runs concurrently and exchanges messages asynchronously while not sharing any state with any other actor. Actor systems can therefore be reconfigured dynamically, while transparently preserving message passing semantics, which is very helpful for scalability, elasticity, and load balancing. Erlang is a functional language which supports fault tolerance and hot swapping. SALSA’s compiler generates Java code allowing programmers to use the entire Java library collection. It also supports transparent actor migration across the Internet based on a universal naming system. Several research efforts have been made to support non-functional concerns for SALSA programs. The Internet Operating System (IOS) (Maghraoui et al., 2006) is a distributed middleware framework that provides support for dynamic reconfiguration of large-scale distributed applications through opportunistic load balancing. In addition to the distributed load balancing capability, the Cloud Operating System (COS) (Imai et al., 2013) further supports elasticity, enabling adaptive virtual machine allocation and de-allocation on hybrid cloud environments, where private clouds connect to public clouds.

Table 1 summarizes the cloud programming frameworks mentioned above.
<table>
<thead>
<tr>
<th>Category</th>
<th>Name</th>
<th>Non-Functional Concerns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Analytics</td>
<td>MapReduce, Hadoop</td>
<td>Scalability, Fault tolerance, Load balancing</td>
<td>MapReduce exposes simple abstractions: map and reduce. Hadoop is an open-source implementation of MapReduce.</td>
</tr>
<tr>
<td>Data Analytics</td>
<td>Pig, Hive</td>
<td>Scalability, Fault tolerance, Load balancing</td>
<td>Pig and Hive are high-level languages for Hadoop.</td>
</tr>
<tr>
<td>Data Analytics</td>
<td>Mahout, Spark, Shark</td>
<td>Scalability, Fault tolerance, Load balancing</td>
<td>Mahout is a machine learning library running on top of Hadoop. Spark is a data analytics framework especially for iterative and graph applications. Shark is a Hive-compatible SQL engine running on top of Spark.</td>
</tr>
<tr>
<td>Web Applications</td>
<td>Google App Engine</td>
<td>Scalability, Elasticity, Fault tolerance, Load balancing</td>
<td>A PaaS from Google. The runtime environment is restrictive, but provides a good scalability. Python, Java, PHP, and Go are supported.</td>
</tr>
<tr>
<td>Web Applications</td>
<td>Microsoft Azure</td>
<td>Scalability, Elasticity, Fault tolerance, Load balancing</td>
<td>A PaaS from Microsoft with automatic scaling, automatic patching, and security services.</td>
</tr>
<tr>
<td>Web Applications</td>
<td>Heroku</td>
<td>Scalability, Elasticity, Fault tolerance, Load balancing</td>
<td>A PaaS from Heroku with libraries for data management, mobile users, analytics and others.</td>
</tr>
<tr>
<td>Distributed Computing</td>
<td>SALSA, Erlang</td>
<td>Scalability, Fault tolerance (for Erlang)</td>
<td>General-purpose programming languages based on the actor model.</td>
</tr>
<tr>
<td>Distributed Computing</td>
<td>IOS, COS</td>
<td>Scalability, Elasticity (COS), Load balancing</td>
<td>Middleware framework for managing distributed SALSA programs.</td>
</tr>
</tbody>
</table>
3 CLOUD PROGRAMMING FRAMEWORKS

In this section, we present existing cloud programming frameworks for data analytics, web applications, and distributed computing, namely, MapReduce, Google App Engine, and the SALSA programming language.

3.1 Data Parallelism with MapReduce

MapReduce created by Google is a data-parallel programming model based on the map and reduce higher-order abstractions from functional programming. Its implementation is also called MapReduce which is designed to automatically schedule parallel processing of large data sets distributed across many computers. MapReduce was designed to have good scalability to achieve high throughput, and fault tolerance to deal with unavoidable hardware failures. These non-functional concerns are transparent to application developers; MapReduce provides a simple application programming interface requiring to define only the map and reduce functions that have the format shown in Listing 1.

\[
\begin{align*}
\text{map} & : (k1, v1) \rightarrow \text{list}(k2, v2) \\
\text{reduce} & : (k2, \text{list}(v2)) \rightarrow \text{list}(v3)
\end{align*}
\]

Listing 1 MapReduce abstract application programming interface

The computation in MapReduce basically consists of two phases: the “map” phase producing intermediate results, followed by the “reduce” phase processing the intermediate results. Both phases take a (key, value) pair as input and output a list. Keys and values can be arbitrary numbers, strings, or user-defined types. First, the map function gets called by MapReduce and transforms a (k1, v1) pair into an intermediate list of (k2, v2). For instance, in text mining applications, v1 may be a partial text of an input file. Next, these intermediate lists are aggregated into pairs where each pair has all v2 values associated with the same k2 key. Finally, the reduce function is invoked with a (k2, list(v2)) pair and it outputs a list of v3 values.

Suppose you have a big input file consisting of DNA sequences, which are combinations of \{C, G, T, A\} characters, one way to write map and reduce functions to count each character occurrence in the input file can be given as shown in Listing 2.
The map function emits count 1 as an occurrence of a DNA character. The reduce function takes a list of counts and emits the sum of these counts for a particular DNA character. By calling these simple two functions repeatedly, MapReduce outputs occurrences of each DNA character. An example execution for the DNA sequence counting example is shown in Figure 2.
Figure 2  Execution sequence of MapReduce

3.2 Service-Oriented Programming with Google App Engine

Google App Engine (App Engine hereafter) is a programming framework to develop scalable web applications running on Google’s infrastructure. Developed web applications are hosted by Google and accessible via the users’ own domain name. Google automatically allocates more computing resources when service demand grows and also balances the workload among the computing resources. App Engine is regarded as a PaaS implementation and charges costs based on the use of storage, CPU, and bandwidth. App Engine currently supports development in Python, Java, PHP, and Go. App Engine’s core
features and its programming framework are described in the following sections. We use only Python for brevity.

### 3.2.1 Core Features
The core features of App Engine (as of July 2014) are as follows.

- **Sandbox:** To prevent harmful operations to the underlying operating system, applications run in an isolated secure environment called sandbox. In the sandbox, an application is allowed to only access other computers on the internet using Uniform Resource Locator (URL) Fetch and Mail services, and also application code is able to only run in a response to a web request. The application is not allowed to write to the file system directly, but instead uses Datastore or Memcache services.

- **Storing Data:** App Engine Datastore is a data storing service based on Google’s BigTable (Chang et al., 2008), which is a distributed storage system that scales up to petabytes of data.

- **Account Management:** Applications can be easily integrated with Google Accounts for user authentication. With Google Accounts, an application can detect if the current user has signed in. If not, it can redirect the user to a sign-in page.

- **App Engine Services:** App Engine provides a variety of services via APIs to applications including fetching web resources, sending E-mails, using cache and memory instead of secondary storage, and manipulating images.

### 3.2.2 Programming Framework for Python
We describe the App Engine’s programming framework for Python as visualized in Figure 3. When the web server receives a Hyper-Text Transfer Protocol (HTTP) request from the browser, the web server passes the request to a framework (a library that helps web application development) via Web Server Gateway Interface (WSGI). The framework then invokes a handler in a user script (in this example, main.py) and the invoked handler processes the request and creates an HTTP response dynamically.
The Python runtime in App Engine uses WSGI as an interface to connect the web server to the web applications. WSGI is simple, but reduces programming effort and enables more efficient application development. Here, we give an example of a request handler using the webapp2 framework that interacts with WSGI as shown in Listing 3.

```python
import webapp2

class MainPage( webapp2.RequestHandler ):
    def get( self ):
        self.response.headers[ 'Content-Type' ] = 'text/plain'
        self.response.write( 'Hello, World!' )

application = webapp2.WSGIApplication( [( '/', MainPage )], debug=True )
```

Listing 3  Application Python script for the Helloworld example

The application object—an instance of WSGIApplication class of webapp2 framework—handles the requests. When creating the application object, a request handler called MainPage is associated with the root URL path (/). The webapp2 framework invokes the get function in the MainPage class when it receives an HTTP GET request to the URL /. In the get function, it creates an HTTP response with a Content-Type header and a body containing a “Hello, World!” message.
3.3 Distributed Actor Systems with SALSA

SALSA is a concurrent programming language based on the actor model (Agha, 1986). Each actor runs concurrently and exchanges messages asynchronously while encapsulating its state. Since an actor does not share any memory with other actors, it can migrate to another computing host easily. As shown in Figure 4, under a hybrid IaaS cloud environment, actors can migrate between the private and public clouds seamlessly with runtime software installed on virtual machines (VMs) on both ends. In this scenario, application developers need to either use COS middleware or support appropriate non-functional concerns by themselves since IaaS clouds only provide scalability by means of VM addition/removal. Nevertheless, hybrid clouds can be attractive for those who can access their private computing resources at no additional cost and who need high computing power only occasionally.

![Figure 4 Workload scalability over a hybrid cloud using actor migration](Imai et al., 2013)

In the following subsections, we introduce the actor-oriented programming model followed by a distributed application written in SALSA.

3.3.1 Actor-Oriented Programming
Actors provide a flexible model of concurrency for open distributed systems. Each actor encapsulates a state and a thread of control that manipulates this state. In response to a message, an actor may perform one of the following actions (see Figure 5).

1. Alter its current state, possibly changing its future behavior.
2. Send messages to known actors asynchronously.
3. Create new actors with a specified behavior.
4. Migrate to another computing host.

Figure 5  Actors programming model
Analogous to a class in Java, SALSA programmers can write a behavior which includes encapsulated state and message handlers for actor instances:

- New actors get created in SALSA by instantiating particular behaviors with the `new` keyword. Creating an actor returns its reference. For instance,
  ```java
  ExampleActor exActor = new ExampleActor();
  ```
- The message sending operation (`<-`) is used to send messages to actors; messages contain a name that refers to the message handler for the message and a possibly empty list of arguments. For instance,
  ```java
  exActor<-m(1, 2);
  ```
- Actors, once created, process incoming messages, one at a time.

### 3.3.2 Programming Distributed Applications

A simple distributed SALSA application that migrates an actor with name as specified by a Universal Actor Name (UAN) from location specified by a Universal Actor Location (UAL) is shown in Listing 4. A UAN is an identifier that represents an actor during its life time in a location dependent manner. An actor’s UAN is mapped by a naming service into a UAL, which provides access to an actor in a specific location.

```java
behavior Migrate {
    void print() {
        standardOutput<-println("Migrate actor is here!");
    }

    void act(String[] args) {
        UAN uan = new UAN("uan://wcl.cs.rpi.edu:3030/myName");
        UAL ual = new UAL("rmsp://host1.cs.rpi.edu:4040/myLocator");
        Migrate migrateActor = new Migrate() at (uan, ual);
        migrateActor<-print()@
        migrateActor<-migrate("rmsp://host2.cs.rpi.edu:4040/myLocator")@
        migrateActor<-print();
    }
}
```

Listing 4  SALSA migration code example
Listing 4 defines a behavior of the Migrate actor. First, the program starts from the act message handler and a Migrate actor named migrateActor is created with a UAN “uan://wcl.cs.rpi.edu:3030/myName” at a UAL “rmsp://host1.cs.rpi.edu:4040/myLocator”. Next, the actor receives a print message and prints out a string “Migrate actor is here!” in the standard output of host1. Right after printing the string, the actor migrates to a new location specified by a UAL “rmsp://host2.cs.rpi.edu:4040/myLocator”. Finally, after the migration, the actor prints out the same string at host2. Note that SALSA’s transparent migration support enables to execute the same print message in two different hosts.

4 SAMPLE APPLICATIONS
In this section, we present sample applications for the three programming frameworks. First, we describe a data analytics application that computes the average temperatures of historical weather data using Hadoop. Next, we show a simple web-based bulletin board application using Google App Engine. Thirdly, we describe a distributed face recognition application using SALSA.

4.1 Data Analytics
The application presented in this subsection processes large amount of data by Hadoop to compute average temperatures for every month using temperature data collected from all over the world.

4.1.1 Global Surface Summary of Day Data (GSOD)
Global surface summary of day data or GSOD (NOAA Satellite & Information Service, 2014) is a collection of daily weather data produced by the National Climatic Data Center. The weather data has been collected since 1929 to the present by stations all over the world. The elements contained in the daily data include: mean temperature, mean sea level pressure, mean visibility, and mean wind speed.

The datasets are provided in ASCII characters. Each row contains weather data for a station on a particular day (See e.g., Listing 5). Fields are explained in Table 2.
Since we compute average temperatures for every month, the fields we are interested in are \texttt{YEAR}, \texttt{MODA} and \texttt{TEMP}.

### 4.1.2 MapReduce Functions for Averaging Numbers

Suppose we are given an input file that consists of \texttt{(tag, number)} pairs, where tag is an arbitrary string of a month, year, station, and so on, we can compute an average number for each tag by the \texttt{map} and \texttt{reduce} functions shown in Listing 7.

```java
map(String key, String value)
// key: tag, value: number
pair = (value, 1);
emit(key, pair);

reduce(String key, String value)
// key: tag, value: a list of (number, count) pairs
sum = 0; count = 0;
foreach pair (n,c) in value do
    sum = sum + n;
    count = count + c;
end
average = sum / count;
emit(average);
```

Listing 7  MapReduce pseudo code for average number calculation
The map function emits an intermediate \((tag, (number, 1))\) pair for every input and then the reduce function computes the average number \( \frac{1}{N} \sum_{i=1}^{N} number_i \) for each tag from an aggregated pair \((tag, [(number_1, 1), ..., (number_N, 1)])\).

4.1.3 GSOD Data Analytics Application

Based on the functions’ pseudo code presented in the previous section, we show a Hadoop application that computes average temperatures for every month using GSOD temperature data. The map function is defined as a GsodMapper class implementation that extends the Mapper class as shown in Listing 8.

```
public class GsodMapper extends Mapper<Object, Text, Text, DoubleIntPair> {
    private Text word = new Text();
    private DoubleIntPair pair = new DoubleIntPair();

    public void map(Object key, Text value, Context context)
            throws IOException, InterruptedException {
        if (value.toString().startsWith("STN")) {
            return;
        }
        String[] data = value.split(" +");
        String yearmon = data[2].substring(0, 6);
        word.set( yearmon );
        pair.set( Double.parseDouble( data[3] ), 1 );
        context.write( word, pair )
    }
}
```

Listing 8  GsodMapper class definition

The Mapper class is a generic class that takes four type variables, i.e., the input key type, input value type, output key type, and output value type. In this example, the input key type is Object, input key value is Text, output key type is Text, and output value type is DoubleIntPair. DoubleIntPair is used to store a pair of double and int values.
The map function repeatedly receives one line of GSOD weather input data in the value variable. If the value starts with an "STN", that means that it is a line for weather field names, therefore we skip it. Otherwise, split the value into a data array in which each element is separated by spaces. Since we want to compute the average temperature for each month, we are interested in the YEARMODA and TEMP fields, which are the third and fourth fields of the weather data. From these two fields, we can extract a year and month by data[2].substring(0,6) in the YYYYMM format and temperature by data[3]. By calling context.write() function, the map function outputs a key value pair in the format (key=YYYYMM, value=(temperature, 1)).

The GsodReducer class that implements the reduce function is defined in Listing 9. The specified input types are Text and DoubleIntPair and output types are Text and DoubleWritable. The reduce function iterates the values which is a list of a temperature and count, and sums up the temperatures and counts to compute the average temperature.

```java
public class GsodReducer extends Reducer<Text, DoubleIntPair, Text, DoubleWritable> {
    private DoubleWritable average = new DoubleWritable();

    public void reduce(Text key,Iterable<DoubleIntPair> values, Context context)
        throws IOException, InterruptedException {
        double sum = 0;
        int count = 0;
        for (DoubleIntPair value: values) {
            sum += value.getDouble();
            count += value.getInt();
        }
        average.set(sum / count);
        context.write(key, average);
    }
}
```
Listing 9  GsodReducer class definition

4.2  Bulletin Board Web Application
In this subsection, we present a simple bulletin board web application developed on the Google App Engine framework (see Figure 6 for user interface). This application lets users post a message with their usernames to a public bulletin board. The posted messages are stored in the server and shared among the users accessing the application.

Figure 6  User interface of the bulletin board application

4.2.1 HTML Template
Unlike the helloworld example, which just returns an HTTP response containing a “Hello, World!” string to the browser, we can create a more elaborate and dynamically generated response using HTML templates such as Jinja2 (Ronacher, 2011). Jinja2 is a template engine for Python supported by Google App Engine. For the bulletin board application, we can use a Jinja2 template as shown in Listing 10.

```html
<html>
<head>
  <link type="text/css" rel="stylesheet" href="/css/main.css" />
</head>
<body>
  <div class="headline">
    <h1>Bulletin Board</h1>
  </div>
  <div class="content">
    <h4>Leave a message!</h4>
    <form action="/postmsg" method="post" class="form-inline">
      <label>user: </label>
      <input type="text" name="user" placeholder="anonymous">
      <label>message: </label>
      <input type="text" name="content">
      <button type="submit">submit</button>
    </form>
    {% for msg in msgs %}
      <b>{{ msg.user }}</b> wrote at <i>{{ msg.date }}</i> : 
      <blockquote>{{ msg.content|escape }}</blockquote>
    {% endfor %}
  </div>
</body>
</html>

Listing 10  Jinja2 HTML template for bulletin board application
In the second div section, this HTML file first creates a form to post a username and a message to the web server. Once these forms are filled by the user and the submit button is pressed, an HTTP POST request is sent to the URL specified by /postmsg. Then, the message showing part is expressed using the Jinja2 template. In the Jinja2 template, “{{ variable }}” refers to the value of a variable. Similarly, “{% control logic %}” refers to control logic such as for, while, or if. In this example, the stored messages are referred to as msgs and the username, creation date, and content of each message are displayed by iterating msgs. Using HTML templates, we can express dynamic and static HTML contents side by side easily.

4.2.2 Application Script
An example script for the bulletin board application is shown in Listing 11.
```python
import os, jinja2, webapp2
from google.appengine.ext import ndb

JINJA_ENVIRONMENT = jinja2.Environment(
    loader=jinja2.FileSystemLoader(os.path.dirname(__file__)),
    extensions=['jinja2.ext.autoescape'])

class Messages(ndb.Model):
    user = ndb.StringProperty()
    content = ndb.StringProperty()
    date = ndb.DateTimeProperty(auto_now_add=True)

class PostMessage(webapp2.RequestHandler):
    def post(self):
        msgs = Messages()
        user = self.request.get('user')
        if user == "":
            msgs.user = "anonymous"
        else:
            msgs.user = user
        msgs.content = self.request.get('content')
        msgs.put()
        self.redirect('/

class MainPage(webapp2.RequestHandler):
    def get(self):
        msgs = ndb.gql("SELECT * FROM Messages
            ORDER BY date DESC LIMIT 10")
        template_values = {'msgs': msgs}
        template = JINJA_ENVIRONMENT.get_template('index.html')
        self.response.write(template.render(template_values))

application = webapp2.WSGIApplication(
    [('/', MainPage), ('/postmsg', PostMessage)], debug=True)
```

Listing 11 Application Python script for bulletin board example
Once the script starts, a global variable `JINJA_ENVIRONMENT` is instantiated and configured to look for Jinja2 HTML templates from the current directory (specified by "__file__"). This program consists of three classes: Messages class for data model, PostMessage class for handling submitted messages, and MainPage class for handling requests for the main page. Meanwhile, an application object is created by WSGIApplication so that HTTP requests for ‘/’ and ‘/postmsg’ are handled by the MainPage and PostMessage class respectively.

The Message class defines a data model that consists of three fields: user for the username, content for the message content, and date for the current time when the model instance is added to the datastore (specified by “auto_now_add=True”). The PostMessage class defines the behavior when the application receives an HTTP POST message. It extracts the user and content fields from the request and put them into a Messages data model instance. The MainPage class defines the behavior when the application receives an HTTP GET message to the main web page. Using a SQL-like query called GQL, it retrieves stored messages from the Messages datastore up to ten in descending order of date (latest date comes first), and then passes the retrieved msgs messages to the template index.html. Finally, it creates a complete HTTP response from the template and returns the response to the browser.

4.3 Distributed Face Recognition

Face recognition is an application that can be vastly improved by leveraging cloud computing resources. Rather than using a single device, in this case a mobile phone, we can offload parts of the image processing to the cloud (Abolfazli et al., 2014). By using cloud computing, we can save battery in the mobile device, and also we can consider larger data sets. Using the SALSA programming language and the FaceRecognizer API from OpenCV (OpenCV, 2014), we can design a mobile phone application to recognize a face in a given image using a database of faces (see Figure 7).
The face recognition application consists of two stages, the training stage, and the prediction stage. The training stage trains a database of faces using the `FaceRecognizer` method defined in OpenCV. The prediction stage predicts a given face using the desired method with a certain confidence. When the database of faces is small, there is no need to offload computation because the phone can process the faces locally just as fast as it would take to offload to the cloud. As the database grows we run into the limitations mentioned above and offloading to the cloud can be beneficial.

The distributed face recognition model consists of a “farmer actor” that creates $N_1$ “worker actors” in the cloud. While the farmer and worker actors reside in the cloud, a client on the mobile phone requests the farmer actor to recognize an unknown face. The farmer actor assigns each worker actor a range of faces ($N_2/N_1$ each) to train from the cloud face database containing $N_2$ faces. The worker actors then predict the closest match to the unknown face based on their assigned database. The farmer actor collects the closest match from each worker actor and calculates the best candidate for the unknown face. Pseudo programs for the client, farmer, and worker SALSA programs are shown in Listings 12, 13, and 14 respectively.
behavior FaceClient {
    void act(String[] args) {
        FaceFarmer farmer = (FaceFarmer) FaceFarmer.getNameByServiceName("uan://nameserver/facefarmer");
        Image unknownImage = new Image(args[0]);
        farmer<-predictAll(unknownImage)@
        displayImage(token);
    }
}

Listing 12  SALSA pseudo code for face recognition client actor

behavior FaceWorker {
    FaceRecognizer faceRecognizer;
    void train(Image[] assignedDatabase) {
        faceRecognizer.train(assignedDatabase);
    }

    ImageMetaData predict(Image testImage) {
        // return the closest match
        return faceRecognizer.predict(testImage);
    }
}

Listing 13  SALSA pseudo code for face recognition worker actor
behavior FaceFarmer implements ActorService {
    //CloudFaceDatabase contains N2 faces.
    void act(String[] args) {
        faceWorker[] workers = new faceWorker[N1];
        for (i = 0; i < N1; i++) {
            workers[i] = new faceWorker(new UAN(…));
            workers[i]<-migrate(new UAL(…));
            workers[i]<-train(
                N2/N1 faces from CloudFaceDatabase);
        }
    }
}

Image predictAll(Image testImage)
    join {
        for (i = 0; i < N1; i++)
            workers[i]<-predict(testImage);
    }@getBestMatch(token)@currentContinuation;
}

Image getBestMatch(ImageMetaData[] closestMatches) {
    // Find best match with the highest confidence.
}
}

Listing 14  SALSA pseudo code for face recognition farmer actor

In Listing 14, note that a join block in the predictAll message handler is used to synchronize all the worker actors executing predict. After all the workers finish processing predict, the join block returns an object array ImageMetaData[], which contains prediction confidence from the workers. Finally, getBestMatch takes the array and finds the best matching image with the highest confidence.

5 CONCLUSIONS
Cloud computing has the potential to bring the benefits of large-scale data analytics and high-performance computing to everyone’s fingertips enabling unprecedented societal applications. We have described three programming frameworks for cloud computing: Hadoop’s MapReduce, Google App Engine, and SALSA. To illustrate the usage of these frameworks, we have shown a weather data analytics application on Hadoop, a simple bulletin board web application on Google App Engine, and a distributed face recognition application on SALSA offloading computation from a mobile device to the cloud. As we have seen, the target applications for these frameworks are very different from each other, and the support level of non-functional concerns is also different between IaaS and PaaS. These frameworks are ideal for the illustrated target application domains, but not necessarily for other application scenarios. Therefore, to get the best from these cloud services, application developers still need to carefully consider the characteristics of their applications and find best matching cloud services in terms of QoS, cost, programmability, vendor lock-in possibility, and others.

ACKNOWLEDGEMENT
This work is partially supported by an Amazon AWS in Education Research Grant and a Google Cloud Credits Award.

REFERENCES


