Font Detective: Identify Fonts in the Cloud

A report on the design and implementation of a cloud-hosted web application

Luke Mitchell
University of Bristol
Merchant Venturers Building
Bristol, UK
lm0466@my.bristol.ac.uk

***Abstract*—This paper presents Font Detective, a cloud-hosted application that performs computer vision analysis on user-supplied images with the aim of identifying a font present. The app utilises Amazon's Web Services platform to deploy a fully-scalable, load-balanced and resilient product. The application is available online at *nugenthill.com*.**

Keywords—cloud; font; computer vision; OpenCV; Amazon; AW; Haar-like; cascade

I. Introduction

The inspiration behind Font Detective came gradually, through multiple meeting with designers, in a variety of scenarios. It seemed to me that design is often about gaining inspiration in the work of another, questioning "what colour is used here?" and working with a similar palette, asking "what if I use a similar image, but in this context?". Until now, however, identification of a font has been almost impossible, down to pure experience. I have attempted to create a tool to allow designers, along with the wider population, to easily identify a font used in an image, and to fuel their creativity.

Font Detective requires users to upload sample images to the website, consuming a large amount of space. Font Detective also requires image classification using computer vision techniques, a computationally-intensive task, consuming a lot of resource. Due to these requirements, the task lends itself to the cloud, an environment where available resources can easily be scaled to meet demand.

The application I have created allows a user to upload an image and have it classified, in the cloud, in real time. Unfortunately, due to time constraints, I have not been able to progress this to the classification of fonts; I have, however, put in place a framework that allows the easy replacement of the current classifiers (bananas and faces) with those for individual fonts. In Fig. 1 you can see an example of a completed job, showing a classified face.

II. Design

I compiled several key goals when designing Font Detective (the application), wishing to ensure consistently high quality of service with a minimum of human intervention. These goals were that the application scales automatically in response to periods of high or low utilisation; that the application is capable of self-diagnosing any issues or errors experienced, and that

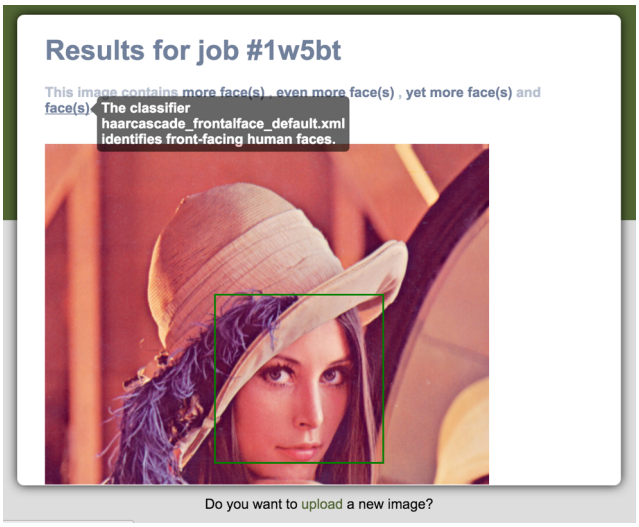

Fig. 1. Interface showing results of classification

appropriate action would be taken; that the application is secure, with isolated components where possible; that the application minimises downtime, even during upgrades and, finally, that it can be deployed quickly and easily, in an automated fashion where possible. The motivation for these goals is to reduce dependence upon human activity, which is intrinsically error-prone, and to shift responsibility to automated, testable services, scripts and health checks.

To achieve these goals, I considered the main infrastructure-as-a-service (IaaS) providers: *Amazon Web Services* (AWS), *Microsoft Azure* and *Digital Ocean*. I also considered platform-as-a-service (PaaS) providers such as *Google Cloud* and *Heroku*, however I quickly deemed that these would not be suitable due to the diverse requirements of the application. PaaS provides a quick, easy-to-deploy environment for applications utilising certain technologies, such as Ruby, Node.js and Go, it doesn't however, have the flexibility of IaaS platforms, particularly with the deployment of a scalable, multi-element application.

I also opted to ignore *Rackspace*, *Linode* and the multitude of similar offerings as, although their hosting was cloud-based, they lacked the breadth of tools of the larger providers. *Amazon* and *Microsoft* both offered a wealth of tools for network

monitoring, routing, instance-scaling and storage, whereas the smaller platforms often lacked one or more of these elements.

Another disadvantage was the lack of documentation and community support, comparatively with AWS, *Azure* and *Digital Ocean*; I feared this would slow development in a project with a relatively tight time budget.

Comparing AWS, *Azure* and *Digital Ocean*, I found that the former two have significantly more functionality. *Digital Ocean* provides high-quality cloud hosting, with the ability to auto-scale and dynamically route traffic, but lacked platform features such as databases, queues or health checks. All of these features were possible but required configuration by hand. *Amazon* and *Microsoft* were much more similar in their features; as such my decision to use AWS was ultimately decided by familiarity with the service.

To ensure easy scalability for the application, in keeping with the specified design criteria, I decided that the architecture should feature homogenous front-end nodes, serving the website content and communicating with clients, and homogenous back-end nodes, processing the requests as they are issued. The advantage here is that software only has to be written once and that network load can easily be balanced across all the servers: there is no individual point of failure. The application features a front-end node with a web server, a processing node capable of performing classification and, optionally, a training server capable of creating new classifiers.

To maximize security, processing nodes are not accessible via the Internet; they need only to communicate with the front-end nodes and the database. Front-end nodes require Internet access but should only have read access to database resources, as these will only be used to display results, not written to directly. The network is divided into subnets accordingly, isolating the processing nodes and allowing access permissions to easily be set on that database and other infrastructure.

III. Implementation

The application involved two main development areas: the front-end website, with the software responsible for serving it, and the back-end processing service, performing classification. These areas could be worked on independently once an interface had been specified, and so I developed them serially, focusing on the front-end first.

A. Front-end

The front-end is required to serve static content, HTML, JavaScript, CSS and images, and to handle user-uploaded sample images. Uploaded images were stored in the cloud and a processing request added to a queue, making it available to the processing nodes. Additionally, some asynchronous communication between the front-end and the client's browser is performed, leveraging WebSockets and a REST API to achieve this. Due to the breadth of the requirements, I opted to implement a custom web server for the task; simple file-serving could easily be achieved using *Apache* or *Nginx*, but adding the additional functionality would require separate applications to be written anyway.

The web server is implemented using Node.js (Node), a lightweight, asynchronous server-side JavaScript engine with a wealth of plugins available. Node applications are single-threaded by default, consuming very little in the way of resources, lending them to the lower-tier servers available on AWS. Another advantage of using Node is that multiple instances of the web server may be run in parallel when required, for example when running on a larger, multi-core server. This capability further lends it to the intended scalability of the application.

The software is built using a library called *Express.js*, a framework for quickly creating web applications in Node. The main website is static HTML, JavaScript and CSS, served from a directory on the server. The software also exposes API endpoints to which the client can make REST requests, using AJAX; these endpoints allow the client to retrieve the status of an individual job, lists of completed jobs and information about the classifiers. Additionally, the client initiates a WebSocket connection each time a new page is loaded; this connection, identified using a unique string stored in the user's session, allows information about the job to be passed quickly and asynchronously between the client and the server.

When the user connects to the website they are greeted with an upload page. This allows them to select a sample image, using a 'drag and drop' interface or by clicking the form. The uploaded file is stored in the cloud, in a dedicated S3 bucket, and the URL passed back to the client, ensuring that the image can be served regardless of which front-end node they connect to. Local copies are then deleted. The user is then presented with the option to select a region of the image containing the font to identify; this is achieved using an intuitive 'click and drag' mechanism. Once the region, known as the bounding-box, has been selected, the client sends this via WebSockets to the server, which then adds a processing request to an SQS queue. Whilst the request is being processed, the user is presented with a 'spinner'; during this time the client is polling a REST endpoint for the completion status. Once the results are ready, the user is redirected to a results page, showing the classification status, and allowing them to see detected objects by hovering over a displayed link. The website design can be seen in Fig. 2.

The website is designed to be lightweight, easy to use and informative. There is a requirement for JavaScript, due to the asynchronous nature of the communication, however there are relatively few[1] Internet users without JavaScript, so this was

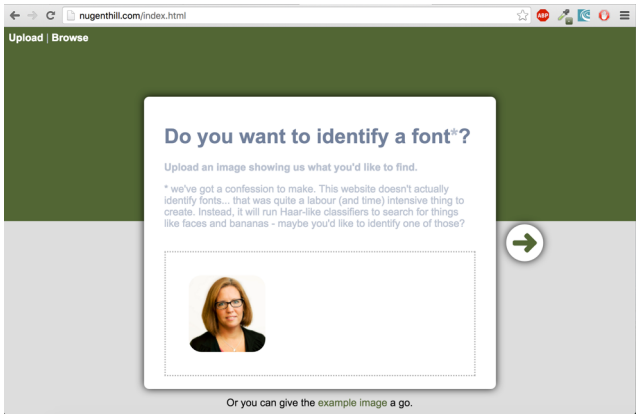

Fig. 2. The upload page of the website, as displayed by *Google Chrome*.

deemed to be acceptable. The website also features some minimal animation, fading between pages, to provide interest but not distract the user. There are customised error pages, including 400 errors, 500 errors and a "job not found" page. Finally, no page exceeds 200 KB[1], excluding sample images, keeping loading times low.

B. Back-end

The back-end, processing nodes performs classification on the sample images uploaded by users, using computer vision to do so. The standard approach for performing image classification is to use the *Viola-Jones* algorithm[2]: this uses a framework of prominent features, present in the object being searched for, to determine whether the object is present in an image or not. The feature frameworks are commonly known as *Haar Cascade* classifiers and are created by performing a lengthy training step, in which a large number of images containing the target object are analysed.

To implement the classification, I used the open-source library *OpenCV*. The library is a fast, full-featured computer vision package, used extensively in academia and industry[3]. *OpenCV* is written in C++ but has bindings available in many languages, including Node and Python. I opted to use Node for my back-end, for some of the same reasons stated earlier in this section, as well as its ease of integration with the AWS API, and consistency across the application.

The processing node is fairly simple in function: it periodically polls the job queue, checking for processing requests created by the front-end. Once a job has been retrieved, the software retrieves the sample image from cloud storage, crops it to the required size, as specified by the user-specified bounding-box, and then runs all the available classifiers on the image by calling the relevant *OpenCV* functions. The software then determines the results of the classification and stores the result in a database, making it available to the front-end. All downloaded sample images are removed after use, ensuring the server does not run out of storage, and completed jobs are removed from the queue.

As mentioned in the Introduction, the processing nodes don't actually identify fonts. This is due to the enormous amount of time, and resource, required to train and test classifiers for fonts. The processing nodes work in the exact way that they would for font classification, however, performing the same algorithm on the same classifier type. The software currently runs several face classifiers, provided with the *OpenCV* library, and a somewhat-temperamental banana classifier[4].

C. Instances

For both the front- and back-end nodes, I set up instance images (AMIs) containing the requisite software. These images were configured to start the software at boot, allowing them to be quickly and easily deployed when required. The AMIs were installed with *Git* and Node.js, also *nginx* for the front-end, *imagemagick* and *OpenCV* for the back-end. Each image was also installed with a tagged version of the *GitHub* repository for the relevant software. The images used a 64-bit *Ubuntu* Linux distribution as the operating system, allowing for a range of servers to be provisioned – with greater than 4 GB of memory, if required. The *upstart* service, along with a tool called *forever*, was used to start the Node applications at boot.

A third AMI was also created for the purpose of training new *Haar Cascade* classifiers. This image contained *OpenCV* and a selection of tools that I wrote for the purpose[2]. Training a classifier involves using these tools manually, specifying the desired font to classify, running a tool to create positive samples and then starting the training program[3]. It would be possible, with more time, to create a web interface to facilitate and automate this however.

The former two AMIs are extremely lightweight and will run on the free-tier instance of Amazon's cloud hosting, *Elastic Cloud Compute* (EC2). Running on the free-tier, *t2.micro*, the CPU utilisation is 0.1% and 20% at idle, for the front- and back-end nodes, rising to 40% each with 10 concurrent connections, all attempting to classify images. An instance of this size is extremely cheap, costing under $10/month[4], and the application can cope with light load with just two of these provisioned. You can see the *CloudWatch* graphs showing the CPU utilisation in Fig. 3.

The training AMI is significantly more resource-hungry. Training a classifier is a process that takes a few hours to several days depending upon factors such as the size of the sample images, the number of sample images and the number of iterations – more of which all increases accuracy, as well as training time. The training process is also fairly memory-intensive, and will run out of memory fairly quickly on the

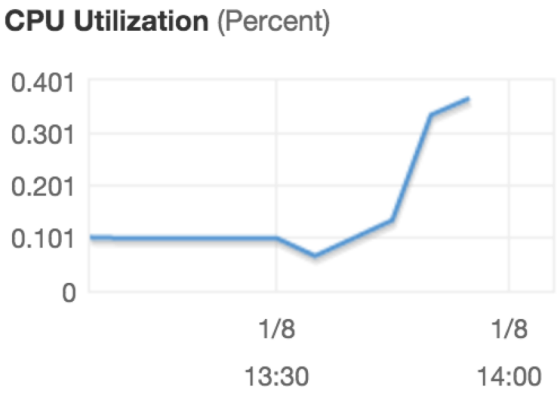

Fig. 3. Graphs indicating the processing load on a front-end node with 10 concurrent connections

1. This could be reduced further by *minifying* the pages; this was not performed in order to keep the code easily readable.

2. These tools are 'literary-rain', available at https://github.com/font-detective/literary-rain, and 'font-detective-classifier-training', available at https://github.com/font-detective/font-detective-classifier-training.

3. There is a video demonstration of the tool available at https://vimeo.com/150895682.

4. As calculated by 0.013 * (24 * 30) = 9.36

t2.micro, resorting to swap storage and further slowing the process. In my experiments, I found that the *m3.large* instance type performed fairly well. Keeping a training instance online is much costlier than running the application, costing around $90/month[5]; this is $6 for each classifier[6].

D. Scaling

In order to fulfil the automatic scalability requirement, specified in the Design section, I created *AutoScaling Groups* (ASGs) for both front-end and processing nodes. These are an infrastructure component within AWS that allows for the definition of rules for scaling the number of EC2 instances up or down. Both ASGs scale the number of instances up by one once the average CPU utilisation reaches 80%, and down by one when it is beneath 40%, within a range of one to five[7]. This scaling occurs without any human action and, providing the activity spike does not exceed the health check interval (see Network sub-section, later in this section), there should be no downtime as a result. In the event that a sharp activity spike does occur, the application will recover itself, launching an instance at a time until the load falls beneath the threshold.

An ASG works in tandem with a *LaunchConfiguration*, a component for specifying an instance type and an AMI to launch. Both front-end and processing nodes launch their respective AMIs onto *t2.micro* instances during scaling, as specified in the Instances sub-section. The configuration also specifies which subnet and security group to place the instance in, ensuring all security settings propagate correctly.

E. Message Queue

A queue is implemented using *Amazon*'s *Simple Queue Service* (SQS): this scales under heavy load and provides features such as atomic consumption of messages, ensuring only one receiving instance receives each message for a specified timeout. SQS is used for the job queue, containing processing requests created by the front-end nodes. When a processing node consumes a message it is hidden from other nodes for 60 seconds, giving the software time to complete the job and upload the results to the database; in practise, a job is usually completed in two to five seconds, leaving a large margin for error. If a job has not been removed from the queue in this time, it becomes visible to the other processing nodes; this behaviour is desirable in the event of a processing node failing during a job.

F. Cloud Storage

The application stores user-uploaded images using *Amazon*'s *Simple Scalable Storage* (S3) component. S3 scales automatically under heavy load and provides a useful, central data store for content common to all front-end nodes. A second S3 'bucket' is also used as a failover website, displaying a static error page in the event of simultaneous failure of all the front-end nodes. This utilises a feature of S3 for hosting a static website, where the exposed endpoint can be aliased in the hosting record, allowing it to serve contain at a domain name. Hosting the main website using S3 does not make sense, however, due to the dynamic nature of some of the pages.

G. Database

Database storage is performed using the *DynamoDB* NoSQL, another offering by *Amazon*. This is utilised for quickly and easily storing result information from the processing nodes, and to store longer-term data about the classifiers used, such as descriptions and names[8]. The database contains two tables, one for the job results and one for the classifier data. As *DynamoDB* is NoSQL, data stored within it is only eventually consistent, and may not be available to all nodes at the same time. However, as data is only written to the database once, and only read thereafter, this is of no concern.

H. Network

A network architecture was designed using the tools available on the AWS console. As mentioned in the previous section, I wanted to ensure that individual components were properly isolated from one another, residing in dedicated subnets, on identifiable IP ranges. To this effect, I created a subnet for the front-end nodes, specifying that it automatically assign a public IP address to each instance. I also create a subnet for the processing nodes; these were firewalled from the Internet and could only be accessed via SSH from a specified IP range, tunnelling through a gateway node.

To further segregate the nodes, I created *Security Groups*, collections of firewall rules, for the front-end and processing nodes. The front-end is accessible from the Internet, via a direct client connection, and via the ELB, on a different port. The processing nodes are not open to the Internet and are only accessible, via SSH, from a gateway node, as described in the Design section. The ELB resides in its own subnet; connections from this subnet are permitted to the front-end using the PROXY protocol, as described in later in this section.

Other network resources, namely the *DynamoDB* tables, the S3 buckets and the SQS message queue, are also controlled using *Security Groups*. The database tables are only readable by instances in the front-end subnet, and only writeable by those in the processing node subnet. The S3 image bucket is readable by the public, as this is used as a distributed store for uploaded images (known as a content distributed network, or CDN), and its endpoints are embedded into the website directly. The SQS queue is writable by both front-end and processing nodes, but only readable by the latter, as jobs are not processed by the front end. This configuration minimises the potential for harm in the event of an application error, or malicious action if a server is compromised, as each node is sandboxed to the greatest extent possible.

5. As calculated by 0.12 * (24 * 30) = 86.4

6. Assuming it takes two days to train the average classifier.

7. Obviously in a real-world scenario, the maximum value would be set to much greater than this. This was kept conservatively low to protect my bank balance.

8. This information is displayed in a tool-tip on the results page.

External routing to the application uses the domain name *nugenthill.com*, a domain I purchased for another project but never used. The routing service available within AWS is called *Route53*, which I utilise to route to a load-balancer and to a failover site. The ELB routes the request to the front-end node with the least CPU utilisation, ensuring maximum performance across the network. *Route53* can also be used to route according to other policies, namely geolocation and latency; the former would be used to direct requests to the nearest geographical server in a multi-zone application. The ELB is monitored using a *HealthCheck*, another tool available via AWS. The *HealthCheck* performs a HTTP GET request to load-balancer every 10 seconds, ensuring that it is still online. After three successive failures, the ELB is considered offline and the route is redirected to the failover site, a static error message hosted on an S3 bucket. A diagram of the network can be seen in Fig. 4.

The application was designed to minimise centralisation, allowing geographically disparate instances to be used, ensuring low-latency response times. The decentralised approach also has the benefit of allowing easy, automatic scaling, in response to periods of high or low load. The current configuration only utilised a single datacentre, or zone, however by duplicating the infrastructure in another, or several more datacentres, and by routing client requests to the nearest using a geolocation policy, this can be easily extended.

One caveat I experience was when routing WebSocket connection through the: the load-balancer did not provide the client's IP address for TCP connections, which are partially used by the sockets, as such, the sender's location could not be determined. To resolve this, I had to enable the PROXY protocol[5] as a policy within the ELB, routing the requests to a specific port on the front-end nodes. The policy utilises the X-Forwarded-For header to store the client's address; this is then stripped and the packet re-routed to the front-end software, bound to a different port, by a local *Nginx* instance, run on each front-end node. [6]. This behaviour can be seen in Fig. 5.

I. Automating Deployment

Once the network infrastructure had been completed, I automated the set-up process using *Amazon*'s *CloudFormation* (CF). This tool allows AWS infrastructure to be specified in

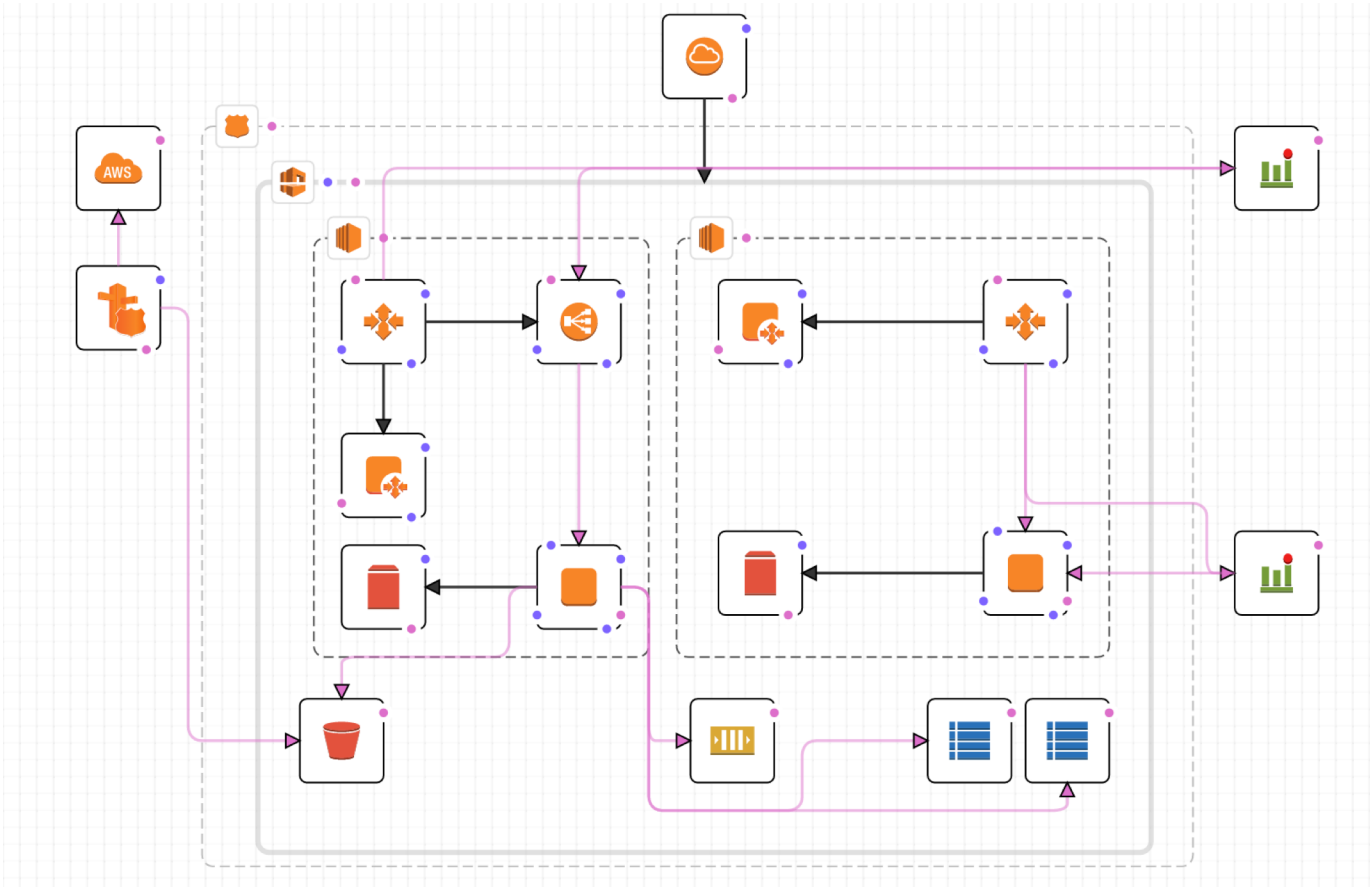

Fig. 4. Set-up diagram, as displayed by the *Amazon Web Services CloudFormation* tool. The diagram shows two subnets, residing within a *Virtual Private Cloud* (*VPC*). The left-hand subnet contains a front-end instance (bottom-right), with attached storage volume (bottom-left); it also contains a *LaunchConfiguration* (middle-left), an *AutoScaling Group* (top-left) and a load-balancer (top-right). The right-hand subnet contains a process node instance with an attached volume, a *LaunchConfiguration* and an *AutoScaling Group*. The *VPC* is attached to an Internet gateway (top-middle) and contained within a *Route53 Hosted Zone*. Also within the *VPC* is an S3 bucket (bottom-left), an SQS queue (bottom-middle) and two *DynamoDB* tables (bottom-right). The icons to the right of the *VPC* are *HealthChecks* for the front-end and processing nodes. The icons to the left of the *VPC* are the routing table entries for the domain.

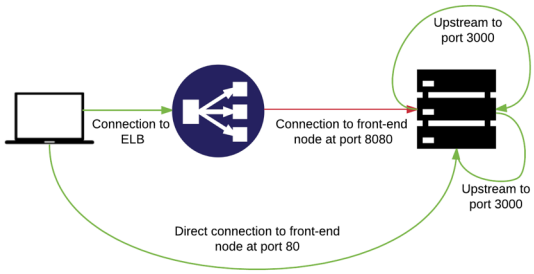

Fig. 5. Illustration of the PROXY routing protocol in use with the front-end server. Here the client connects to the load-balancer which forwards the request using the PROXY protocol, adding the X-Forwarded-For header with the client's IP address. The front-end node runs an instance of *Nginx* that strips this header and forwards the request to the web server software, replacing the origin address with the client's IP. Nginx listens at port 80 for direct connections, and 8080 for load-balancer connections. The webserver software listens at port 3000.

JSON and provisioned automatically via an online-interface. Utilising CF for this dramatically increases the speed at which the application can be set-up, not only streamlining the process but shielding the user from the hazardous task of using the AWS console.

IV. Considerations

All AMIs are configured to run on start up, leading to quick spin-up times during scaling or deployment. A possible improvement, however, would be to configure the instances to automatically update. The only way to do this currently would be to bring down the application and re-deploy new instances with updated code – or to update the live instances manually! Utilising a service such as *GitHub*'s web-hooks could facilitate auto-updating: when a new release is tagged on the master branch relevant repositories software is notified and can respond accordingly, by performing a *pull* of the latest code, then restarting the software.

V. Conclusion

The application I envisioned was a complex, enterprise-scale project that scaled automatically and responsively, refused to 'go down' under heavy loads and provided a solution to a yet-unsolved problem: the identification of fonts. The vision was ambitious and not entirely realised, however, the application delivered is capable of scaling both the front-and back-ends automatically, monitoring the health of all the nodes and the routes, redirecting the user to a static, failover site in the event of emergency and performing image classification in the cloud.

The major drawback of the application is its inability to identify fonts, something it is clearly designed to do. It does, however, identify faces and bananas, tasks which are computationally identical. The task of training classifiers for each font has been tackled in the undertaking of this project, and the relevant tools have been created, however the scope of creating and testing the classifiers seems beyond that of creating a cloud application.

To take the application further would involve automating the update process, as specified in the Considerations section, and to design and build an interface to facilitate classifier training, as described in the Instances sub-section. Then, with the addition of some font classifiers, the application would be complete and, in principal, highly functional.

If you wish to see more, full code listings are provided at the 'font-detective' *GitHub* organization, available at https://github.com/font-detective. This includes the front-end and processing node software, instructions for deploying the application, examples and the *CloudFormation* template.

References

[1] Herlihy, P. (2013). *How many people are missing out on JavaScript enhancement?* [online] Government Digital Service. Available at: https://gds.blog.gov.uk/2013/10/21/how-many-people-are-missing-out-on-javascript-enhancement/ [Accessed 8 Jan. 2016].

[2] P.A. Viola, M.J. Jones, Rapid object detection using a boosted cascade of simple features, in: CVPR, issue 1, 2001, pp. 511–518.

[3] Pulli, K., Baksheev, A., Kornyakov, K. and Eruhimov, V., 2012. Real-time computer vision with OpenCV. *Communications of the ACM*, *55*(6), pp.61-69.

[4] Ball, T. (2014). *mrnugget/opencv-haar-classifier-training*. [online] GitHub. Available at: https://github.com/mrnugget/opencv-haar-classifier-training [Accessed 8 Jan. 2016].

[5] Tarreau, W. (2015). *The PROXY protocol*. [online] Haproxy.org. Available at: http://www.haproxy.org/download/1.5/doc/proxy-protocol.txt [Accessed 8 Jan. 2016].

[6] Docs.aws.amazon.com, (2016). *X-Forwarded Headers for Elastic Load Balancing - Elastic Load Balancing*. [online] Available at: http://docs.aws.amazon.com/ElasticLoadBalancing/latest/DeveloperGuide/x-forwarded-headers.html [Accessed 8 Jan. 2016].