SpiritDogs

Ahmer Butt
ab15015; 21452

Louis Wyborn
lw15771; 25955

***Abstract*—This report details the features and implementation of the SpiritDogs web application, A quantitative study is performed to assess the implementation's availability and scaling limits, finding it is limited only by AWS account limits or extreme usage spikes. The repository is found at github.com/ahmerb/SpiritDogs. The application can be run online at https://spiritdogs.co.uk.**[1]

***Index Terms*—Serverless, Amazon Web Services, TensorFlow**

I. Introduction

SpiritDogs is a web application for people to discover the dogs that match their spirit. Users upload photographs of themselves and receive their SpiritDog for the day. Behind the scenes, a state-of-the-art neural network trained to classify dog breeds processes the user images. Users can view and track their SpiritDogs over time.

SpiritDogs is a Single Page Web Application build using ReactJS. It works seamlessly on both desktop and mobile. The backend is a serverless application hosted on Amazon Web Services (AWS).

II. Application Features

Upon launching the application, users are presented with a landing page. The user can use the navigation bar to either log in or sign up. Upon logging in, users are presented with the 'Home Page'. A list of previous SpiritDog classifications are shown. Clicking a previous SpiritDog opens the 'View SpiritDog' page. This shows the breed of the SpiritDog, the picture uploaded by the user and a picture of the user's SpiritDog. The Home Page also presents an option to create a new SpiritDog. Clicking this navigates to the 'New SpiritDog' page. There is a 'Choose file' button that allows the user to select a photograph of themselves to upload. Once chosen, the user presses a 'Create' button. Loading indicators show the progress as the image is uploaded and then classified. Upon success, the 'View SpiritDog' page is shown with the results.

III. Dog Breed Classifier

The dog breed classifier is a convolutional neural network (CNN) training using transfer learning. We take a pre-trained InceptionV3 network and re-architect and retrain the top layers with a dog breed dataset. This fine-tunes the complex features of the network to learn to classify dog breeds instead of general image classification. The SpiritDogs classifier is inspired by [1]. The dataset is available at [2].

The model is implemented in Python using Keras with a TensorFlow backend. Keras includes an implementation of InceptionV3 pre-trained on ImageNet, with the top layers removed. A 2d average pooling layer is added, followed by a fully connected layer with L2 regularization, Relu activations and 40% dropout, followed by a softmax-cross-entropy output layer. Merging the new model in Keras found to be a complicated task.

[1]SpiritDogs is available at both www.spiritdogs.co.uk and at spiritdogs.co.uk. HTTP users are automatically redirected to the HTTPS website.

IV. System Architecture

A. Serving the Website

The website is decoupled from the API and is hosted as a static website in an **S3** bucket. A **CloudFront** distribution is configured to serve the application. CloudFront is an Amazon hosted Content Delivery Network (CDN). A CDN is a distribution of servers that provides high availability and performance by distributing the service geographically. CloudFront is globally distributed, with 160 points of presence in 29 countries. CloudFront also increases availability by caching content in 'edge locations' and serving content directly from these caches, reducing load on the server. This also decreases response time for users located further from Ireland, the AWS region where the SpiritDogs S3 bucket is hosted. Finally, **Route 53** is a Domain Name System (DNS) web service. It points the custom domain to the CloudFront distribution.

The frontend is a single page web application built using ReactJS. React-bootstrap UI Kit is used to provide primitive React components such as buttons and form fields. React Router is used to help configure frontend routing. The frontend resizes to fit desktop, tablet and mobile screens.

B. Machine Learning Model Serving

It is possible to deploy the SpiritDogs classifier as a Python Lambda function triggered on '/classify' API requests. However, Lambda has a strict 250 MB limit on the deployment package size, including code dependencies. TensorFlow is 330 MB [3]. Many suggestions are found online for circumventing this limit. For example, it is possible to compress Python dependencies and import dependencies from ZIP files. However, this does not work with libraries that include shared object files. We could not sufficiently reduce the size.

As a result, the classifier is not deployed as a serverless function. An alternate offering that still minimises developer time spent and handles auto-scaling is desirable. **Amazon SageMaker** is a fully managed platform for building, training and deploying machine learning models. Amazon handles running the model on EC2 instances. The SpiritDogs classifier is running on an ml.t2.medium instance.

TensorFlow Serving is a library for high-performance production serving of a trained model. TensorFlow Serving builds

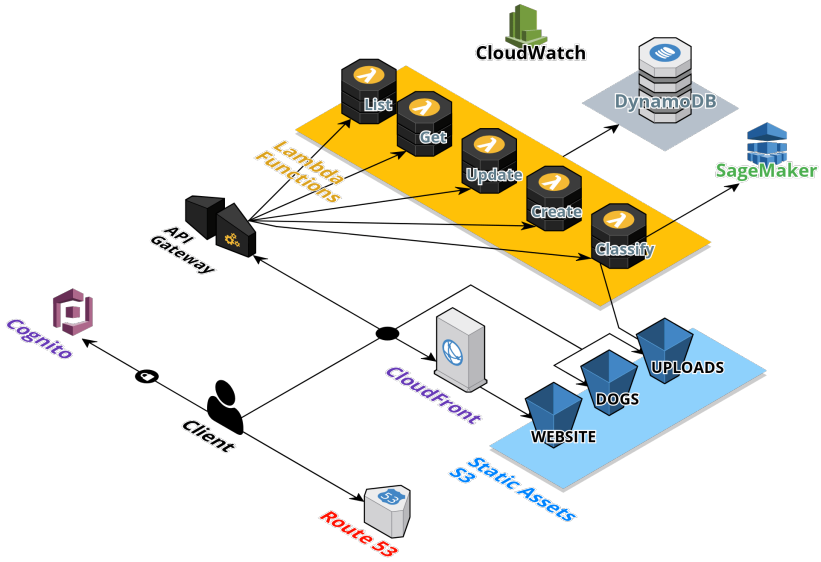

Fig. 1: The architecture of SpiritDogs. There are five Lambda functions, triggered by API Gateway when requests are received. The function 'Classify' is triggered by a request to `GET /classify/[id]`. It retrieves the note for the corresponding ID from DynamoDB, pulls the uploaded image for this classification from S3, preprocesses the image into a format which the TensorFlow model accepts, sends the image to SageMaker, decodes the classification response, saves response to DynamoDB and finally returns the result. 'List', 'Get', 'Update' and 'Create' are triggered by `GET /notes`, `GET /notes/[id]`, `PUT /notes/[id]` and `POST /notes` respectively. They interact with DynamoDB and act as CRUD (create-read-update-delete) operations for the `notes` table. For discussion of the remaining components, see the text.

a **Docker** container that exposes an HTTP endpoint which accepts model inputs. Hosting a TensorFlow Serving model built outside of SageMaker requires additional setup. An Amazon SageMaker endpoint needs to handle requests to '/invocations' and `/ping` on port 8080. However, a TensorFlow Serving server takes requests on /models/[model_name]:predict on port 8501. To solve this problem, SpiritDogs creates a custom Docker image. The Dockerfile uses the TensorFlow Serving image as a base image and installs **NGINX**. NGINX is configured inside the container to set up a reverse proxy which forwards requests from SageMaker to TensorFlow Serving. The container is then pushed via shell script to an Amazon **Elastic Container Registry** (ECR). Other shell scripts then create the SageMaker model by pointing to the container in ECR; create a SageMaker endpoint configuration and then deploy the SageMaker endpoint.

C. Database

SpiritDogs requires a database which keeps a retrievable history of a users classifications. In SpiritDogs, classification results are encapsulated in a *note*. A note contains the classification result, a reference to the user uploaded image and the ID of the user who created the note. This can be represented in a single table. As neither relational queries nor cross-table atomic operations are required, a NoSQL database is possible. This is typically easier to scale and cheaper.

SpiritDogs uses **DynamoDB**. It is fully managed NoSQL database from Amazon and automatically scales. It has a serverless pricing model where you are only billed for the read and write requests performed (plus storage).

D. Storage of User and SpiritDog Images

Simple Storage Service (S3) buckets are key-value stores. When a user uploads an image, the frontend directly makes a PUT request to an uploads bucket. SpiritDog images are downloaded by making a GET request to another bucket. User uploaded images are stored in a private folder in the uploads bucket. A client must authenticate as a user to access the user's folder. Uploaded images are stored with a Unix timestamp prepended to the file name. This is typically not a good strategy to ensure uniqueness at scale, but it suffices here as the file name need only be unique per user.

By default S3 does not allow access from a different domain, but the application is served through a custom domain. A Cross-Origin Resource Sharing (CORS) configuration is added to the buckets which allows access from other origins.

E. Authentication

SpiritDogs uses AWS **Cognito** for authentication. A Cognito User Pool is a sign-in provider. It is a directory of users and provides sign-up endpoints, sign-in endpoints, password management and email verification out-of-the-box. SpiritDogs uses a Cognito User Pool as an identity provider for a Cognito Identity Pool. A user firsts authenticates with the User Pool to get tokens to access the SpiritDogs Identity Pool. The identity pool accepts these tokens and provides credentials to access other AWS services.

To authenticate, the Cognito User Pool uses the Secure Remote PAssword protocol. To generate an authentication header for further requests, the client uses the access tokens to create a signature using the AWS4 Signature algorithm [4]. AWS **Amplify** is a code library used by the frontend to access AWS resources. It reduces the complexity to the developer of authenticating with Cognito.

F. Serverless API

An **API Gateway** is configured which exposes HTTP endpoints. When an endpoint is contacted, it triggers a **Lambda** function which processes the request and generates the correct HTTP response. The Lambda functions are written in JavaScript and are executed by Node.js. Figure 1 diagrams the entire cloud architecture of SpiritDogs and details the Lambda functions with their event sources.

V. Scalability and Availability

The SpiritDogs architecture has been designed to let AWS be responsible for scaling as much as possible. We adopted services designed to scale with serverless, including many Backend-as-a-Service (BaaS) offerings. This is where a back-end, such as an API Gateway, is configured by the developer but the cloud vendor handles managing and scaling. All AWS services used are highly available [5] [6]. In all cases SpiritDogs should scale to meet high demand limited only by AWS account usage limits.

AWS Lambda dynamically scales function execution in response to increased traffic. The scaling behaviour is limited but it is possible to request increases. In the EU (Ireland) region, Lambda allows an initial burst of up to 3000 concurrent executions. API Gateway automatically scales without any additional configuration. It is noted that Lambda has a 15 minute timeout and 3008 MB memory allocation limit. However, these are both far in excess of SpiritDogs' requirements.

S3 provides four 9s availability and eleven 9s durability. Each bucket is an unlimited size key-value data store and automatically scales. It is resilient against entire availability zone (AZ) failures. AWS redundantly stores data in a minimum of three AZs in one region. It provides automatic detection and repair of lost redundancies. AWS allows 5 GB per upload and a max file size of 5 TB. This far exceeds the requirements of SpiritDogs.

DynamoDB also provides high availability and replications across AZs. Maximum throughput rate is unlimited, but with an account limit of 10k read and write capacity units per table. One capacity unit is measured as one strongly consistent or two eventually consistent reads per second, with up to 4 KB items. SpiritDogs' database has an autoscaling policy enabled. It tracks consumed capacity and provisions more or less capacity dynamically to maintain a target tracking rate.

CloudFront is a massively scaled and globally distributed CDN. It has a usage peak of 10Gbps or 15k requests per second. However, this is another soft AWS account limit. Route53's DNS servers are distributed to ensure high availability to route end users to the application. AWS has an upper limit of 10k queries per second per IP address in an endpoint.

Cognito has soft limits of 1000 users and 10 sign up, sign in and forgot password requests per second. It is also possible to request higher account limits if required.

SageMaker is not a serverless offering. However, it is fully-managed. It is possible to configure an autoscaling policy that provisions more instances as demand increases. The server only needs to perform an execution of a stateless mathematical function, making scale-out easy. SageMaker models are hosted by Amazon on EC2 instances. The SpiritDogs classifier is running on an ml.t2.medium instance. A t2 instance only provides short-lived 'burstable' scaling, not automatic scale-out. We have submitted requests to AWS Support to remove account limits and allow autoscaling up to three m1.m4.xlarge instances.

VI. Development Infrastructure

SpiritDogs is developed using the Serverless Framework [7]. It allows managing cloud infrastructure as code. The SpiritDogs API is given in a configuration file. It specifies which JavaScript functions are to be deployed as Lambda functions and the events that trigger them. It also specifies **Identity and Access Management** (IAM) permissions for the Lambda functions when they execute. They are granted access to DynamoDB, S3 and SageMaker.

Serverless deploys the entire service, as specified by the configuration, via AWS **CloudFormation**. The CloudFormation stack creates: the Lambda functions; an API gateway; the IAM roles; and log groups in **CloudWatch** for all the AWS resources created. A Serverless plugin is configured to compile and package the JavaScript files using Babel and Webpack.

VII. Logging and Metrics

CloudWatch is used to log all API Gateway executions and Lambda function invocations. Alarms are configured that track DynamoDB consumed read and write capacity and trigger autoscaling. CloudWatch tracks 196 metrics. This consists of invocation counts, latencies and error counts for every API Gateway endpoint, Lambda function, DyanamoDB read and write operation and the SageMaker endpoint. It also tracks S3 storage used.

VIII. Load Testing

A. Methodology and Test Design

SpiritDogs has been load tested using the Python-based Locust framework [8]. Locust is chosen as the load testing scripts are entirely code. This makes working with SpiritDog's complicated authentication system easier.

Our load testing methodology is to test the entire system end-to-end rather than individual pieces. Designing the testing environment as close as possible to the production environment minimises the possibility of seeing false negatives in testing and helps identify bottlenecks that may arise only when the entire ecosystem of components needs to collaborate. The test involves test users communicating with API Gateway, S3 and Cognito exactly as real users would. Each test user is a unique user in the Cognito User Pool. Users first contact Cognito to gain credentials to access the API and to access their private S3 uploads bucket directory. Suitable authentication headers are provided with all requests, reflecting the production environment.

Test users do not target a particular requests per second (RPS) for each endpoint. Instead, they aim to mimic real user sessions. Figure 2 shows a diagram of the behaviour of a user in a typical session. Each state represents an action a user can take. Arrows specify the actions a user can choose next. The behaviour is entirely encoded into Locust scripts.

Locust is used to spawn 100 test users who follow the state diagram. They select each path with the given execution ratio and with random wait times between actions. The test starts with a ramp up period where 5 new users are spawned every

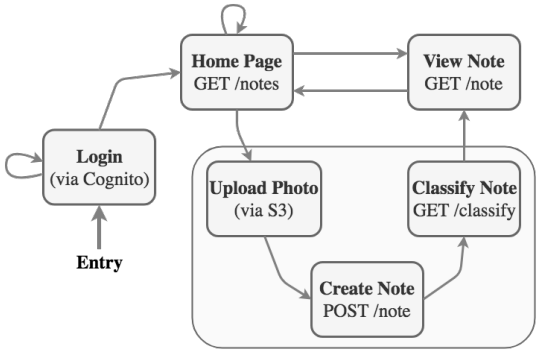

Fig. 2: Transition matrix describing the behaviour of a real user. Locust test users mimic this. After authenticating users perform the 'Home Page' action first, reflecting a real session. From here, users can either refresh, view a note, or begin a sequence of actions to create a new note.

second. It takes 20 seconds to reach 100 users. This load is sustained for 20 minutes, long enough to exhaust 'burstable' scaling and make AWS provision extra resources.

B. Results and Analysis

The test averaged 12.6 requests per second. Figure 3 shows a percentile plot of request latencies. This plot does not take into account the differing load on each endpoint. From all requests, S3 uploads were 28%, 'GET /classify' 23%, 'POST /notes' 24%, 'GET /notes' 17% and 'GET /notes/[id]' 8%. The test is purposefully designed this way to reflect the endpoint usage that arises from real user behaviour rather than artificial endpoint targeting.

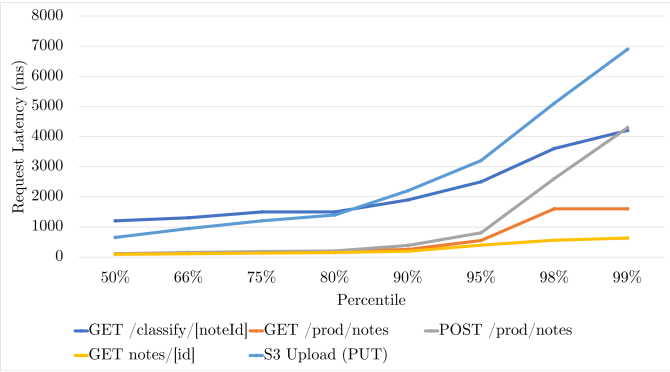

Fig. 3: Percentile plot illustrating the milliseconds required to include a given percentage of all requests.

Figure 3 shows the percentile times rising. Further investigation revealed that after a sustained load of 5 minutes, POST requests to `/notes` began to respond with 502 Bad Gateway errors. CloudWatch logs reveal that DynamoDB is throttling. DynamoDB triggers provisioning more capacity by using CloudWatch alarms. However, these alarms only fired after 2 minutes. Before this time, DynamoDB used bursting to cope with increased demand. The load test creates an extremely rapid increase in usage from zero to 12.6 requests per second. Bursting capacity is quickly exhausted and DynamoDB fails to provision more resources fast enough. There was a 12.72% failure rate for the `POST /notes` endpoint. In production, it

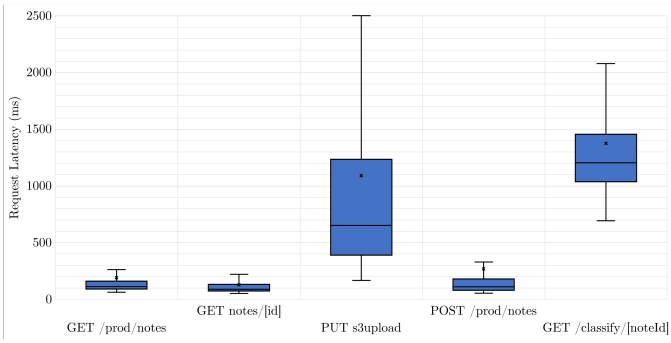

Fig. 4: Box-and-whisker plot showing interquartile range for the latency of each request. Quartiles are calculated exclusive of outliers. To collect this data, custom event hooks were added to Locust.

is unlikely that SpiritDogs will be subject to such sharp usage spikes. If they did occur, a solution is to increase the initial provisioned capacity. If spikes are predictable, it is possible to schedule increases in capacity. Requests to `/classify` also began to return 502 errors, with a 6.79% failure rate. This is because SageMaker also exhausts burstable capacity and does not autoscale due to the AWS account limits. A solution is to request resource limit increases from AWS, which we have done.

The boxplot in Figure 4 illustrates the request latency per action. As expected, the S3 file upload and neural network evaluation are much slower than other requests. However, their summed median is 1850 ms, which can be considered a reasonable time for users to wait for results.

The load test was also run with 800 users and a hatch rate of 20 users per second. Load quickly spiked to 60 RPS. However, AWS Lambda has a 1000 concurrent request limit. The limit was exceeded and Lambda began to throttle. The RPS decreased to 10 requests per second. Due to the volume of concurrent users, request latency became large. The 99th percentile was 24 seconds. If SpiritDogs reached this scale in production the problem is easily solved by requesting larger account limits.

During the ramp-up period, each new concurrent function invocation will encounter a cold start. Although the function may have been invoked many times recently, the warm containers are still busy processing the other requests. When load is increasing, all warm containers will already be preoccupied. When load settles, there will be as many warm containers as concurrent users, allowing new requests to be mapped to warm containers.

IX. Conclusion

In this report, the SpiritDogs application was detailed. We demonstrated a serverless, highly scalable, highly available and production-grade implementation hosted on Amazon Web Services. A state-of-the-art machine learning model was developed and deployed to Amazon SageMaker. Quantitative test results indicated that scaling was only insufficient under extreme usage spikes or when constrained by AWS account limits.

References

[1] J. Jordan, "github.com/jeremyjordan/dog-breed-classifier," 2017.

[2] "https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/dogimages.zip."

[3] Google, "Tensorflow https://www.tensorflow.org/."

[4] Amazon, "Signature version 4 signing process https://docs.aws.amazon.com/general/latest/gr/signature-version-4.html."

[5] ——, "Aws global infrastructure. https://aws.amazon.com/about-aws/global-infrastructure/ [accessed 2 jan 2019]," 2018.

[6] ——, "Pricing. https://aws.amazon.com/pricing/. [accessed 2 jan 2019]," 2018.

[7] Serverless, "serverless.com/framework/."

[8] J. Heyman, "Locust.io."

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 205–220. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294281

[10] A. Innovations, "serverless-stack.com," 2018.