TextToMP3

Nicholas Elison & Jalen Rodgers

To demonstrate our ability to provision and manage cloud resources, we've built a very simple web application that allows users to convert text to speech. After creating an account, a user may enter some text into a text box and click a button to convert that text into an audio file. Users can listen to the audio files they have created, and may also download them or delete them. This web app is currently accessible at https://slangz.com.[1] The cloud service provider used was AWS, and much of our focus was directed toward the architecture, networking and security aspects of our cloud infrastructure setup.

In this report, we will examine the following:

I. Project Goals & Background

We had originally set out to build something with a considerably more complicated architecture, but after encountering a number of early issues, we decided we needed to scale back our aspirations. The new goal was simply to get a very basic web application running on a highly scalable and secure cloud infrastructure setup. As the purpose of this project was not to write an application, we wanted to spend as little time on this as possible, so we defaulted to a language (Python) and framework (Flask) with which we both have some familiarity. Because one of us had used AWS in the past (but still had — and still has — a LOT to learn), we chose to use AWS rather than Azure or Google Cloud Platform.

A secondary goal of this project was to learn how to automate the process of the provisioning infrastructure. Specifically, we wanted to not spend all of our time clicking around the AWS console. Also, while we didn't want to be limited by the free tier and were willing to spend a little bit of money, it would be more economical for us to have the ability to easily tear down (and set up) all of our resources. For this, we decided to use Terraform, and used the following AWS tutorial as a sort of jumping-off point for the project: https://github.com/aws-samples/deploy-python-flask-microservices-to-aws-using-open-source-tools-part2.

[1] Domain name registered years ago for a project that was never started.

We also quickly realized that deploying our app to multiple EC2 instances manually would be quite challenging, and that the way any sensible person would go about doing this would be to containerize the application and use a container orchestration service. For this, we chose to use Docker — a tool with which neither of us actually had much familiarity — and AWS Elastic Container Service (ECS) and Elastic Container Repository (ECR). Learning how to use both Docker and Terraform steepened the already steep AWS learning curve, and made initial project progress almost dispiritingly slow.

II. Application

As the point of this project was not to write an application, we will keep this brief. Aside from what was mentioned in the introduction, we'll note a few facts about the application:

- It's a Flask app using a PostgreSQL database.
- It performs basic CRUD operations.
- User authentication is done in-app (i.e., not with AWS Cognito as we'd originally planned).
- The boto3 library is used to store and fetch objects from an S3 bucket, as well as to invoke an AWS Lambda function (API Gateway was not used).
    - Audio files are stored in an S3 bucket by the Lambda function, but are retrieved and removed from the S3 bucket by the application.
    - Profile picture images are stored in and retrieved from an S3 bucket by the application.

III. Architecture

We opted to go with a traditional three-tier architectural pattern where the client is on the first tier (presentation tier or presentation layer), the web app server is on the second tier (logic tier or application layer), and the database is on the third tier (data tier or data layer). We chose this architecture in part because it is commonly used and so there is no shortage of answers to questions on StackOverflow regarding common problems encountered when working with it, but also because it is easy to scale (e.g., just add more EC2 or RDS instances, which are not dependent on each other), easy to maintain, and the clear separation of concerns is good for security (and also makes creating an architecture diagram easier).

The presentation layer of our application deals with the user interface and user experience. This comes in the form of static files served either from our EC2 instances or from an S3 bucket. At the time of writing, some of our static files (CSS and JS files) are hosted on our EC2 instances, while other files (images and mp3 files) are stored in an S3 bucket. Additionally, at this time, we do not have CloudFront set up.[2]

[2] See conclusion section.

For the application layer, we're using two EC2 instances to run the Flask application. Each EC2 instance is located in a different availability zone (AZ), so that in the event of an issue in one AZ, our application is still accessible. Traffic is distributed across the instances by an application load balancer (ALB). The instances themselves are part of an ECS cluster and auto scaling group, enabling us to add (or remove) instances should traffic or load increase (or decrease). There is a "serverless" component to our application layer as well: a Lambda function that is responsible for calling AWS Polly to generate the MP3 files that users of our service are expecting to get. Although this particular case *could* be handled by the application running on our EC2 instances, it may sometimes be economical to outsource particularly compute-intensive operations to some sort of function-as-a-service solution like Lambda in the event that your traffic profile is spikey and/or unpredictable, and you don't want to pay for all the compute you *may* need when you don't *actually* need it *most* of the time.

For the data layer, we're using a single Relational Database Service (RDS) PostgreSQL instance with a read replica in a different AZ.

The point of spreading our resources across different AZs is to mitigate the impact of localized issues, such as power outage or hardware failures that may only affect resources in a single AZ. In the event of such an issue, we want our service to continue running smoothly and with minimal downtime.

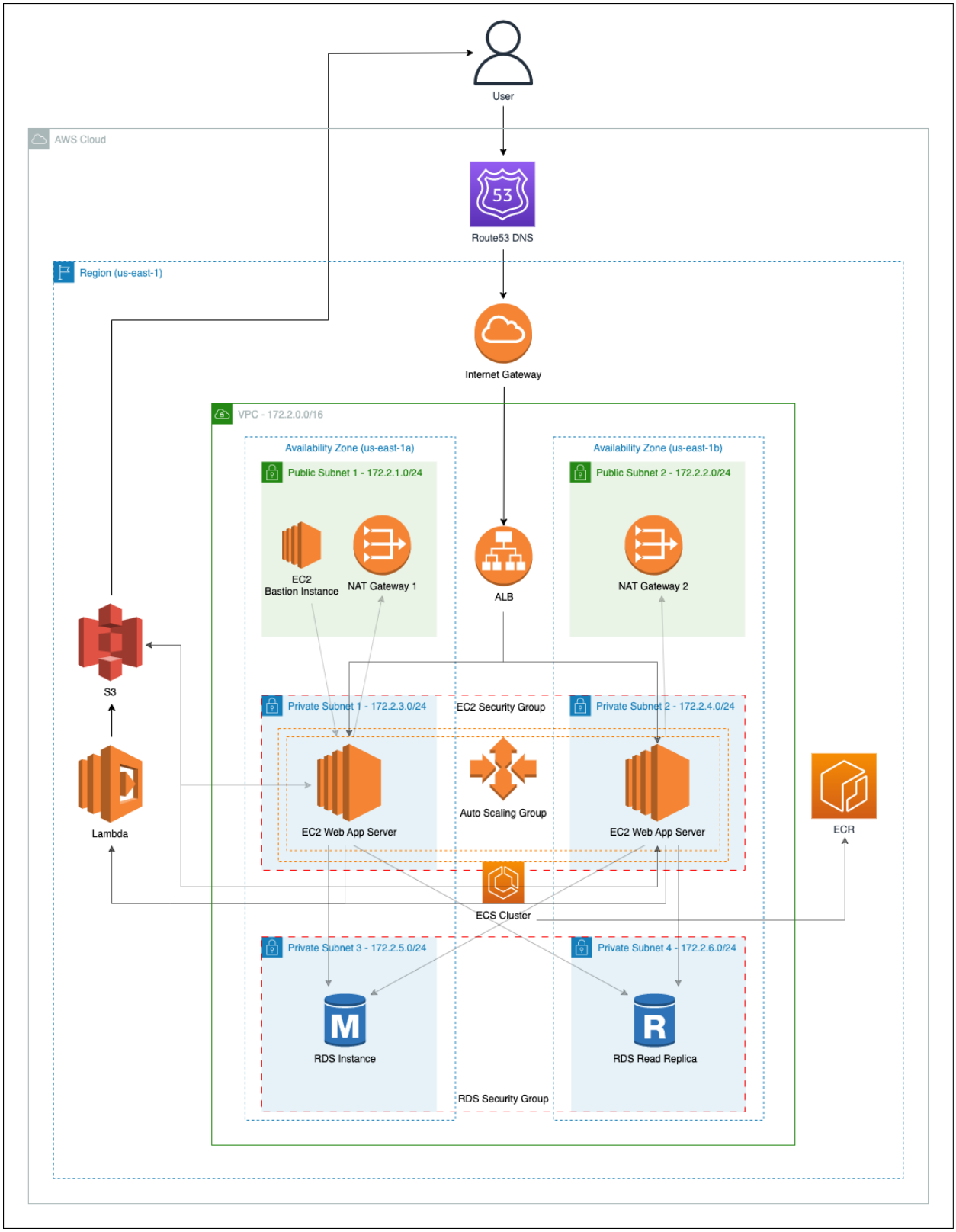

Fig 1. Web application cloud infrastructure architecture. The us-east-1b AZ is now us-east-1e.

IV. Networking & Security

We created a VPC in the region in which we're operating (us-east-1) with a CIDR range of 172.2.0.0/16. This gives our VPC a private IPv4 address space ranging from 172.2.0.0 to 172.2.255.255 (a total of 65,536 addresses). In our VPC, we have a total of six subnets between two AZs (us-east-1a and us-east-1e), with each AZ having one public subnet and two private subnets. The CIDR range of these subnets is 172.2.n.0/24, where n is an integer between 1 and 6, allowing us 251 IP addresses[3] for the resources in each subnet.

The purpose of placing our resources into different subnets is for security. Although outside traffic does ultimately need to reach our EC2 instances in order for our application to work, we don't want to allow just any traffic into the network. By setting up network access control lists (NACLs), security groups and route tables, we are able to fully control what kind of traffic comes into and out of the network as well as how traffic moves around within the network between subnets. This kind of setup also helps mitigate potential damage should a bad actor gain access to one the resources in our network; because we are able to control the flow of traffic throughout our network, this bad actor may not necessarily have access to all of the resources in our network in the same way they would if everything were in a single subnet.

Our public subnet NACL allows:

- Inbound traffic on ports 80 (HTTP) and 443 (HTTPS) from anywhere (0.0.0.0/0)
- Inbound traffic on port 443 (SSH) from anywhere[4]
- Outbound traffic on all ports to anywhere

Our private subnet NACL allows:

- Inbound traffic on ports 80 and 443 from anywhere
- Inbound traffic on port 22 from anywhere
- Inbound traffic on port 5000 (application) from anywhere
- Inbound traffic on port 5432 (PostgreSQL) from anywhere
- Outbound traffic on ports 1024-65535 to anywhere[5]

Our EC2 and RDS security groups have similar rules, but are more specifically tailored to the resources to which these security groups are attached. For example, the RDS security group only allows inbound traffic on port 5432.

Additionally, route tables allow us to direct traffic between subnets and our internet gateway. For example, we have a "public route table" with routes that direct traffic from public subnets

---

[3] Technically 256 IP addresses, but the first four addresses as well as the final address in the CIDR block are reserved by AWS, leaving 251 IP addresses.
[4] It'd be better to restrict this to e.g., just our IP address, but for development purposes we did not do this.
[5] ECS container instances seem to run on ports anywhere in this range; at the time of writing, for one of our container instances, the following ports are in use: 22, 2376, 2375, 51678, 51679.

destined for outside the VPC (0.0.0.0/0) to the internet gateway, thereby allowing resources in the public subnets to access the internet. The same is true for private subnets, only outbound traffic from these subnets is directed to network address translation (NAT) gateways in the public subnets.

For security purposes, our EC2 and RDS instances are placed in private subnets and are not directly reachable through the internet. In Fig. 1, you may notice an additional EC2 instance in one of the public subnets. This EC2 instance serves as a bastion host, enabling us to SSH into the EC2 instances in the private subnets. SSH access is very important for development purposes, but it is also a huge security vulnerability, so we want to limit it where possible. Here, we're using SSH forwarding to prevent having to store a private key on the bastion host. There are also additional advantages to this setup apart from simply enabling SSH access to private instances. For example, it is easier to aggressively patch and update software on a single machine than it is for multiple machines, which effectively reduces the attack surface for bad actors who may be scanning the internet looking for machines with an open port 22 that may be running applications with specific vulnerabilities. Similarly, patches and software updates can be applied without worrying about how they'll affect the machines running the application (e.g., potentially breaking the app).

We're also using Secrets Manager to securely store sensitive information like database credentials. While it is obviously a horrible idea to hardcode something like a database password into an application, it is also not a great idea to store it as an environment variable on the server. For one, if an attacker were to gain access to an EC2 instance running our application, having database credentials stored in environment variables would then give them access to the database. These environment variables could also inadvertently be leaked in logs.

V. DevOps

This section will be short as it is kind of outside the scope of the project. Thanks to Terraform, the provisioning of AWS resources has been almost entirely automated. The setup of the following things was done manually:

- The creation of an AWS user and the managed policy for that user.
- The transfer of domain name from another AWS account and the creation of the Route 53 hosted zone.[6]
- The storing of secrets in Secrets Manager.
- The creation of the Lambda function.[7]

Apart from these things, we can go from having almost no AWS resources to having a running web application by running just a few commands and waiting about 10 minutes. Similarly, we can tear everything down very quickly, preventing us from leaving EC2 and RDS instances

[6] The creation/deletion of the A record pointing to the ALB's public DNS address is done with Terraform.
[7] We could containerize this code and use ECR/ECS.

running when not actively working on the project and allowing us to save a very small amount of money.

VI. Conclusion

One of our biggest regrets with this project was not keeping track of all the small issues that took an inordinately long amount of time to solve as we encountered them. As alluded to in section I, we encountered many of these small but time consuming issues early on, and they had more to do with Docker and Terraform and less to do with cloud infrastructure.

We had some difficulty getting our RDS instance working properly with our application, and as a result, our database setup is not quite as robust as we'd like. We wanted to set up one of AWS' multi-AZ DB clusters which consists of one reader instance and two writer instances (all in different AZs), but we never got around to it. In fact, we're not even really sure how or if our read replica works properly; it was almost added as an afterthought so we had something to put in the empty fourth private subnet, and we'd actually commented it out in the Terraform configuration because we were frequently creating and destroying all of our resources and its setup was taking too long.

If we were to do something else with this project, creating a better database setup is probably the second thing we would do. The first thing we would do is finish setting up CloudFront. While we are able to create a CloudFront distribution with (what we believe to be) the correct configuration settings (while disallowing public access to the S3 bucket), we were unable to avoid an "Access Denied" error when trying to access resources. After having tried seemingly every suggestion found on StackOverflow, we considered the possibility that one StackOverflow user's answer[8] may actually be correct, and it may simply take more time than we were willing to wait for the S3 bucket to finish "setting up" and for CloudFront to work. Because we were continually tearing everything down and setting it up again, we may not have been giving it enough time to work. As we will now have everything up and running for at least a week (between the time of this document's submission and our presentation), we may try to get CloudFront working.

We'd also like to get continuous integration and continuous delivery working. For this, we would use CodePipeline and CodeDeploy. CodePipeline would be used to build and deploy our Docker image to ECR, and CodeDeploy would be used for Blue/Green deployment which is something we admittedly don't fully understand at this time despite recalling that it was mentioned in class. There is a seemingly helpful AWS tutorial explaining how to do something like this with Terraform.[9] This would probably go hand-in-hand with automated testing: another thing we completely neglected to do, but which would be a great addition to this project.

Something severely neglected in this project is logging. Although we have actually set up a CloudWatch log group for checking on the results of the database migrations that are run based

[8] https://stackoverflow.com/a/42285049
[9] https://aws.amazon.com/blogs/opensource/automate-python-flask-deployment-to-the-aws-cloud/

on an ECS task definition we've created, we're not getting any kind of information about how our application is actually running in terms of performance, resource utilization or resource health. We're also not getting notifications about important events (e.g., abnormally high CPU or memory usage). Having this kind of information is actually a critical part of setting up a robust cloud infrastructure, as this is what allows you to make decisions about how to scale. By failing to put any effort into this, we're not only flying blind, but we're missing out on one of the features that makes modern cloud service providers so useful.

As we were putting together this report, we realized some of our NACL rules and security rules could be improved.[10] To do this, we set up our infrastructure again for what may have been the 100th time, and in doing so we encountered a new issue: our EC2 instances were being launched in the same AZ, when they should have been launched in different AZs. This was interesting because our Terraform `aws_autoscaling_group` resource had not changed — it still specified the correct IDs of the private subnets in which the EC2 instances were to be created, and these subnets were still in the correct AZs. In fact, none of our Terraform configuration had changed since we had last set up and destroyed all of our AWS resources. We were also certain we would have noticed this issue if it had occurred previously, as after running `terraform apply` to set up our infrastructure, we then run a script which updates the `~/.ssh/config` files on our machines with the new public DNS addresses of our EC2 instances. This is how we noticed the issue in the first place: because our script depends on the EC2 instances in the private subnets being in different AZs, it did not work correctly and we had an error when trying to SSH into them.

At such a late stage in the project, this was quite an irritating issue to have. A cursory Google search yielded no useful answers or suggestions, but poking around the AWS console and checking our auto scaling group's activity history did lead us to the following: our EC2 instance failed to launch in us-east-1b because AWS does "not have sufficient t2.medium capacity in the Availability Zone [we] requested." This is why we are using us-east-1a and us-east-1e instead of us-east-1a and us-east-1b.

We will conclude by saying that we found this project to be equal arts fun, challenging and rewarding. While it is easy to develop a kind of myopic focus on writing application code, this project forced us to focus on a new-to-us aspect of software development: cloud infrastructure. Simply navigating the AWS management console and trying to get a sense of the lay of the land can be overwhelming, to say nothing of actually provisioning and properly configuring the necessary resources to get an application running, so doing this involved an appreciable (but not overly steep) learning curve. The project also served as an introduction to DevOps; deploying our application without the use of a container likely would have been tremendously challenging, and the manual provisioning of AWS resources would have been an incredibly tedious and untenable process. In this way, we gained a lot of familiarity not just with AWS, but with popular tools like Docker and Terraform. Although we could have taken this project a bit further, we are happy with what we've built, and we've learned a lot in building it.

[10] We're pretty sure there's still room for improvement.