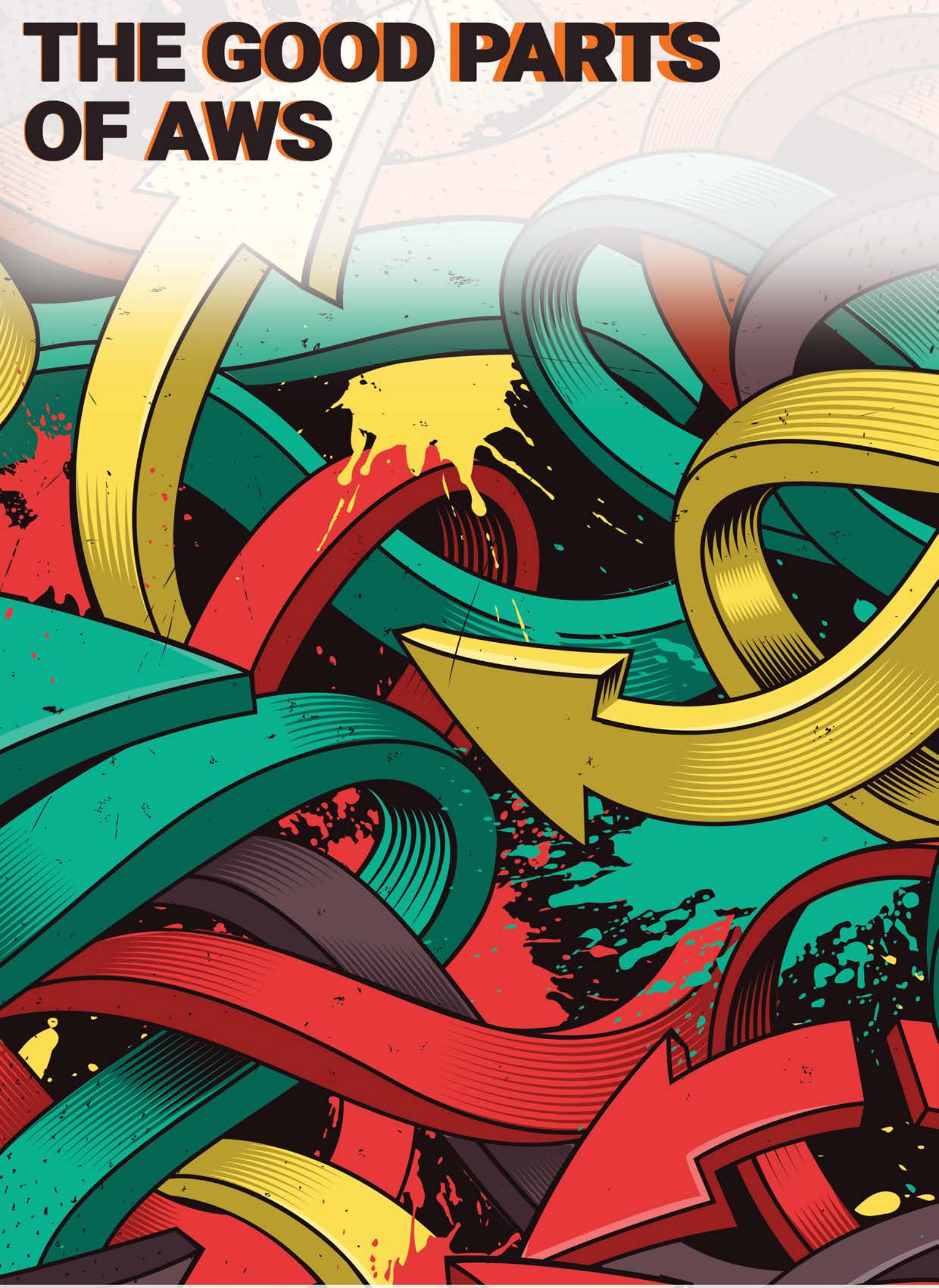
THE GOOD PARTS OF AWS



DANIEL VASSALLO & JOSH PSCHORR

The Good Parts of AWS

Daniel Vassallo, Josh Pschorr

Version 1.1, 2019-12-29

Table of Contents

Preface

This is not your typical reference book. It doesn't cover all of AWS or all its quirks. Instead, we want to help you realize which AWS features you'd be foolish not to use. Features for which you almost never need to consider alternatives. Features that have passed the test of time by being at the backbone of most things on the internet.

Making technical choices can be overwhelming. As developers, we have to make many choices from what seems like unlimited options. We have to choose a programming language, an application framework, a database, a cloud vendor, software dependencies, and many other things. AWS alone offers about 150 services, all with their own set of options and features, sometimes even overlapping with each other. In this book, we will reveal a technique we use ourselves to help make reliable technical choices without getting paralyzed in the face of so many options. This technique will be presented in the context of AWS, but it can be generalized to any other technical decision.

This is a book by Daniel Vassallo and Josh Pschorr. Between us, we have worked with AWS for 15 years, including 11 years working inside AWS. We have worked

on all sorts of web applications, from small projects to massive web services running on thousands of servers. We have been using AWS since it was just three services without a web console, and we even got to help build a small part of AWS itself.

This is an opinionated book. We only cover topics we have significant first-hand experience with. You won't find most of the knowledge we share here in the AWS docs.

Part 1: The Good Parts

The Default Heuristic

Chasing the best tool for the job is a particularly insidious trap when it comes to making progress—especially at the beginning of a project. We consider the relentless search for the best tool to be an optimization fallacy—in the same category as any other premature optimization.

Searching for the optimal option is almost always expensive. Any belief that we can easily discover the best option by exhaustively testing each one is delusional. To make matters worse, we developers tend to enjoy tinkering with new technology and figuring out how things work, and this only amplifies the vicious cycle of such a pursuit.

Instead of searching for the best option, we recommend a technique we call the default heuristic. The premise of this heuristic is that when the cost of acquiring new information is high and the consequence of deviating from a default choice is low, sticking with the default will

likely be the optimal choice.

But what should your default choice be? A default choice is any option that gives you very high confidence that it will work. Most of the time, it is something you've used before. Something you understand well. Something that has proven itself to be a reliable way for getting things done in the space you're operating in.

Your default choice doesn't have to be the theoretical best choice. It doesn't have to be the most efficient. Or the latest and greatest. Your default choice simply needs to be a reliable option to get you to your ultimate desirable outcome. Your default choice should be very unlikely to fail you; you have to be confident that it's a very safe bet. In fact, that's the only requirement.

With this heuristic, you start making all your choices based on your defaults. You would only deviate from your defaults if you realize you absolutely have to.

When you start with little experience, you might not have a default choice for everything you want to do. In this book we're going to share our own default choices when it comes to AWS services and features. We're going to explain why some things became our defaults, and why other things we don't even bother with. We hope this

information will help you build or supplement your basket of default choices, so that when you take on your next project you will be able to make choices quickly and confidently.

DynamoDB

Amazon describes DynamoDB as a database, but it's best seen as a highly-durable data structure in the cloud. A partitioned B-tree data structure, to be precise.

DynamoDB is much more similar to a Redis than it is to a MySQL. But, unlike Redis, it is immediately consistent and highly-durable, centered around that single data structure. If you put something into DynamoDB, you'll be able to read it back immediately and, for all practical purposes, you can assume that what you have put will never get lost.

It is true that DynamoDB can replace a relational database, but only if you think you can get away with storing all your data in a primitive B-tree. If so, then DynamoDB makes a great default choice for a database.

Compared to a relational database, DynamoDB requires you to do most of the data querying yourself within your application. You can either read a single value out of DynamoDB, or you can get a contiguous range of data. But if you want to aggregate, filter, or sort, you have to do that yourself, after you receive the requested data range.

Having to do most query processing on the application side isn't just inconvenient. It also comes with performance implications. Relational databases run their queries close to the data, so if you're trying to calculate the sum total value of orders per customer, then that rollup gets done while reading the data, and only the final summary (one row per customer) gets sent over the network. However, if you were to do this with DynamoDB, you'd have to get all the customer orders (one row per order), which involves a lot more data over the network, and then you have to do the rollup in your application, which is far away from the data. This characteristic will be one of the most important aspects of determining whether DynamoDB is a viable choice for your needs.

Another factor to consider is cost. Storing 1 TB in DynamoDB costs $256/month. For comparison, storing 1 TB in S3 costs $23.55/month. Data can also be compressed much more efficiently in S3, which could make this difference even bigger. However, storage cost

is rarely a large factor when deciding whether DynamoDB is a viable option. Instead, it's generally request pricing that matters most.

By default, you should start with DynamoDB's on-demand pricing and only consider provisioned capacity as a cost optimization. On-demand costs $1.25 per million writes, and $0.25 per million reads. Now, since DynamoDB is such a simple data structure, it's often not that hard to estimate how many requests you will need. You will likely be able to inspect your application and map every logical operation to a number of DynamoDB requests. For example, you might find that serving a web page will require four DynamoDB read requests. Therefore, if you expect to serve a million pages per day, your DynamoDB requests for that action would cost $1/day.

Then, if the performance characteristics of DynamoDB are compatible with your application and the on-demand request pricing is in the ballpark of acceptability, you can consider switching to provisioned capacity. On paper, that same workload that cost $1/day to serve 1 million pages would only cost $0.14/day with provisioned capacity, which seems like a very spectacular cost reduction. However, this calculation assumes both that requests are evenly distributed over the course of the day and that there is absolutely zero capacity headroom. (You

would get throttled if there were a million and one requests in a day.) Obviously, both of these assumptions are impractical. In reality, you're going to have to provision abundant headroom in order to deal with the peak request rate, as well as to handle any general uncertainty in demand. With provisioned capacity, you will have the burden to monitor your utilization and proactively provision the necessary capacity.

In general, you will almost always want to start with on-demand pricing (no capacity management burden). Then, if your usage grows significantly, you will almost always want to consider moving to provisioned capacity (significant cost savings). However, if you believe that on-demand pricing is too expensive, then DynamoDB will very likely be too expensive, even with provisioned capacity. In that case, you may want to consider a relational database, which will have very different cost characteristics than DynamoDB.

It is important to note that, with on-demand pricing, the capacity you get is not perfectly on-demand. Behind the scenes, DynamoDB adjusts a limit on the number of reads and writes per second, and these limits change based on your usage. However, this is an opaque process and, if you want to ensure that you reserve capacity for big fluctuations in usage, you may want to consider using

provisioned capacity for peace of mind.

A final word about DynamoDB indexes. They come in two flavors: local and global. Local indexes came first in early 2013, and global indexes were added just a few months later. The only advantage of local indexes is that they're immediately consistent, but they do come with a very insidious downside. Once you create a local index on a table, the property that allows a table to keep growing indefinitely goes away. Local indexes come with the constraint that all the records that share the same partition key need to fit in 10 GB, and once that allocation gets exhausted, all writes with that partition key will start failing. Unless you know for sure that you won't ever exceed this limit, we recommend avoiding local indexes.

On the other hand, global indexes don't constrain your table size in any way, but reading from them is eventually consistent (although the delay is almost always unnoticeable). Global indexes also have one insidious downside, but for most scenarios it is much less worrisome than that of local indexes. DynamoDB has an internal queue-like system between the main table and the global index, and this queue has a fixed (but opaque) size. Therefore, if the provisioned throughput of a global index happens to be insufficient to keep up with updates on the main table, then that queue can get full. When that

happens, disaster strikes: all write operations on the main table start failing. The most problematic part of this behavior is that there's no way to monitor the state of this internal queue. So, the only way to prevent it is to monitor the throttled request count on all your global indexes, and then to react quickly to any throttling by provisioning additional capacity on the affected indexes. Nevertheless, this situation tends to only happen with highly active tables, and short bursts of throttling rarely cause this problem. Global indexes are still very useful, but keep in mind the fact that they're eventually consistent and that they can indirectly affect the main table in a very consequential manner if they happen to be underprovisioned.

S3

If you're storing data—whatever it is—S3 should be the very first thing to consider using. It is highly-durable, very easy to use and, for all practical purposes, it has infinite bandwidth and infinite storage space. It is also one of the few AWS services that requires absolutely zero capacity management.

Pages 11-34 are not included in this preview.

Part 2: The Bootstrap Guide

In this part, we will walk you through getting a basic web application running in the cloud on AWS. We will start with a blank project, and build the application and its infrastructure step by step. Each step focuses on a single aspect of the infrastructure, and we will try to explain in detail what's happening, and why.

When interacting with AWS, we will use both the AWS console and the AWS CLI. In addition, we will also make use of the following tools:

- GitHub as the source code repository for our application and infrastructure code.
- node.js and npm to build our application.
- git for version control.
- curl to interact with our application.

Starting from Scratch

Objective

Get a simple web application running on a single EC2 instance.

Steps

1. Write a basic "hello world" web application.
2. Manually create basic AWS infrastructure to host our application.
3. Manually install our application on an EC2 instance.

In this section we will create a tiny web application and we will get it running on an EC2 instance in an AWS account. We will start by performing all the steps manually, and we will automate them in later sections. Our application is not very interesting, and the way it will be hosted is far from ideal, but it will allow us to spend the rest of this book building a well-contained AWS setup, step by step.

Pages 37-52 are not included in this preview.

Infrastructure as Code

Objective

Recreate our infrastructure using CloudFormation.

Steps

1. Configure the AWS CLI.
2. Create a CloudFormation Stack.
3. Deploy the CloudFormation Stack.

In this section, we'll recreate the same infrastructure we set up in the previous section, but this time, we'll use CloudFormation to automate the process, instead of setting everything up manually through the AWS console.

Configuring the AWS CLI

We're going to use the AWS CLI to access AWS resources from the command line rather than the AWS console. If you don't already have it installed, follow the official directions for your system. Then, configure a profile named `awsbootstrap` using a newly generated Access Key ID and Secret Access Key, as described in the AWS

Pages 54-65 are not included in this preview.

Automatic Deployments

Objective

Automatically update our application when a change gets pushed to GitHub.

Steps

1. Get GitHub credentials.
2. Install the CodeDeploy agent on our EC2 instance.
3. Create a CodePipeline.

In this section, we're going to use CodeBuild, CodeDeploy, and CodePipeline so that our application gets updated automatically as soon as we push a change to GitHub.

GitHub access token

We will need a GitHub access token to let CodeBuild pull changes from GitHub. To generate an access token, go to https://github.com/settings/tokens/new and click *Generate new token*. Give it *repo* and *admin:repo_hook* permissions, and click *Generate token*.

Pages 67-84 are not included in this preview.

Load Balancing

Objective

Run our application on more than one EC2 instance.

Steps

1. Add a second EC2 instance.
2. Add an Application Load Balancer.

Currently, our application is running on a single EC2 instance. To allow our application to scale beyond the capacity of a single instance, we need to introduce a load balancer that can direct traffic to multiple instances.

Adding a second instance

We could naively add a second instance by simply duplicating the configuration we have for our existing instance. But that would create a lot of configuration duplication. Instead, we're going to pull the bulk of the EC2 configuration into an EC2 launch template, and then we'll simply reference the launch template from both instances.

Pages 86-102 are not included in this preview.

Scaling

Objective

Replace explicit EC2 instances with Auto Scaling.

Steps

1. Add an Auto Scaling Group.
2. Remove `Instance` and `Instance2`.

Thus far, we have created our two EC2 instances explicitly. Doing this means that we need to update the CloudFormation template and do an infrastructure deployment just to add, remove, or replace an instance.

In this section, we'll replace our two EC2 instances with an auto scaling group (ASG). We can then easily increase or decrease the number of hosts running our application by simply changing the value of desired instances for our ASG. In addition, we will configure our ASG to place our instances evenly across our two availability zones. This ensures that if one availability zone has an outage, our application would still have half of its capacity online.

Pages 104-112 are not included in this preview.

Production

Objective

Create separate environments for staging and production.

Steps

1. Extract common resources out of `main.yml`.
2. Create separate stacks for staging and production.

In the real world, we will generally want to have at least one environment to test our application and infrastructure before rolling changes out to production. A common model is to call the testing environment 'staging' and production 'prod'. We'll set these up next.

In this section, we will decommission our existing infrastructure and replace it with two CloudFormation nested stacks representing our staging and prod environments. We will also update our CodePipeline so that it will promote updates to prod only after they've successfully passed through staging.

Pages 114-128 are not included in this preview.

Custom Domains

Objective

Access our application from a custom domain.

Steps

1. Register a domain with Route 53.
2. Create a DNS hosted zone for our domain.
3. Map our domain to the load balancers.

In this section, we will walk through the process of registering a domain with Route 53, and making our application use it. If you already own a domain that you want to use, you can migrate your DNS to Route 53 or completely transfer control of your domain and DNS to Route 53.

Domain name registration costs on the order of tens of dollars yearly, and Route 53 hosted zones are $0.50 per month. Promotional credits are not applicable to either expense. If you don't want to incur these fees just for experimentation, you can skip this section and the next one (HTTPS) and continue using the AWS-provided DNS names of your load balancers.

Pages 130-136 are not included in this preview.

HTTPS

Objective

Migrate our endpoint from HTTP to HTTPS.

Steps

1. Manually create a TLS certificate.
2. Add an HTTPS endpoint.
3. Make the application speak HTTPS.
4. Remove the HTTP endpoint.

As things stand, our application is responding to unencrypted HTTP traffic. In the real world, we want to protect any data as it traverses the network. To do that, we must encrypt our traffic and serve it over HTTPS.

We'll also take this as an opportunity to practice the two-phase change process discussed in Multi-phase deployments to give the chance to anyone using our HTTP endpoint to migrate to HTTPS before we turn off HTTP.

Pages 138-158 are not included in this preview.

Network Security

Objective

Make our instances inaccessible from the internet.

Steps

1. Add private subnets with a NAT gateway.
2. Switch our ASGs to use the private subnets.
3. Only allow the HTTPS port in the public subnets.

In this section, we're going to make our EC2 instances inaccessible from the internet. The instances will be able to reach the internet using a NAT gateway, but the network will not allow anything from the internet to reach the instances without going through the load balancer.

Pages 160-173 are not included in this preview.