The cheap, easy way to deploy an app to AWS

bluematador

aws

Introduction

There are a million things to do when you're deploying an app to AWS. If you're launching this app for the very first time, you might think you have a laundry list of items you must tick off before your app can be successful.

We're here to tell you that's just not true. By jumping the gun, you'll be wasting resources on stuff you just don't need yet.

In this ebook, we'll list what you MUST do and what you don't need to do yet as you're launching your app for the first time.

Hopefully, we can help you save a bunch of time and effort—and maybe even some cash, too—and you'll still have a running, fully functional app.

About Blue Matador

Blue Matador is easy, out-of-the-box AWS cloud monitoring.

Set up Blue Matador in about 5 minutes and start getting valuable insights about your AWS cloud infrastructure immediately. Super-fast setup, super-fast results.

Learn more about how it works on our website.

Table of contents

Table of contents

Table of contents

Step 1: Prepare

To be honest, this is the hardest part, and thus this chapter will be the longest. But if you make sure you have all your ducks in a row early on, you should be well prepared for success.

Let's dive in.

Is AWS the right call for your app?

Long story short: yes.

There are a few other options out there, but they are dwarfed in size and capability by Amazon. AWS covers compute, database, IoT, analytics, DevOps, CDN, access controls, and more. It has over 175 services. And in terms of spread, it has 22 regions (with 5 more announced at the time we write this), including GovCloud, which is specifically for U.S. government software, covering 6 continents, 70 availability zones, and 205 edge locations.

That said, not all services are provided in all regions, and the prices vary, so choose carefully.

What you need to know about your app first

It's wise to know your app back to front before you start implementing anything. Here's a checklist to help you take stock of everything you need to deploy.

For server-side applications:

- How is the application built?
- Is there a script to release and run the application?
- What ports need to be open?
- What services does this application rely on?
- What configuration does it need?
- Can it work behind a load balancer?
- What libraries or local packages does it need?
- Does it have cron jobs, background jobs, or scheduled maintenance?
- What compute infrastructure was it built for? Serverless, containers, or virtual machines?
- If EC2, does the operating system matter?
- Is there a minimum memory requirement?
- Are there any health checks or health endpoints to call?
- Are there any special release instructions from any other developers?

Other stuff to think about:

- What databases (type and version) do you need?
- Are there any static files to serve?
- What domain names and subdomains will you use, and where will they go?
- Do you own the domain names already?
- Is there already an AWS account set up?

Choosing your AWS services

Here are most of the services you'll likely need, short of those specific to your application.

AWS compute services

- **Amazon Elastic Compute Cloud (EC2) / Virtual Private Cloud (VPC).** These are your standard virtual Amazon Elastic Compute Cloud (EC2) machines and the default option. In general, if you use one, you'll use both. EC2 is the compute for servers, disks, IP addresses, load balancers, and firewalls, whereas VPC is the networking layer for EC2.
- **AWS Elastic Beanstalk / Amazon Lightsail.** Beanstalk is Amazon's app deployment service and is compatible with a wide variety of languages and servers. Lightsail is one of the easiest-to-use AWS cloud platforms. It's great for devs just starting in the cloud or who have simple workloads.
- **AWS Lambda / Application Gateway.** This option is gaining in popularity, but to use this pair, you have to have built your application for it specifically, so if you didn't... well, you'll have to choose different compute services. Lambda is the place your code runs, and Application Gateway is your load balancer.
- **Amazon Elastic Kubernetes Service (EKS) / Amazon Elastic Container Service (ECS) / AWS Fargate.** To use these services, your application must have been built in containers. Container services are a good next choice if you don't want to use EC2. ECS is the old-school container option, EKS is hosted Kubernetes, and Fargate is a mix of the two. If you're using containers and haven't chosen which container infrastructure to run on yet, choose EKS.

AWS database services

- **Amazon Relational Database Service (RDS).** RDS is a fully managed cloud database service that supports MySQL, Postgres, MariaDB, SQL Server, Oracle, and the AWS cloud-native database Aurora. At this point, you should know which engine you need, and it's probably not worth changing it. If you use MySQL, consider Aurora—it's faster, built for AWS, and fully MySQL compliant.
- **Amazon DynamoDB.** Globally distributed and available, fully managed NoSQL database. If you need NoSQL, this is probably what you're looking for.

- **Amazon Elasticsearch Service.** Fully managed Elasticsearch cluster. If your application uses Elasticsearch, you could roll your own cluster or use this managed one.

AWS content delivery services

- **Amazon Simple Storage Service (S3).** S3 is Amazon's cloud object storage service. You'll use this one, no doubt about it. It's great for hosting HTML, Javascript, and CSS. Also great for getting file uploads, PDFs, and large documents out of your database.
- **Amazon Route 53.** Another one you'll definitely use. Amazon Route 53 is your managed DNS. If you already bought your domain from some other provider, and they offer free DNS, switch to Route53. It's $1 per month and integrates much better with AWS services (which stands to reason).
- **AWS Certificate Manager.** This one you might not have considered, but it's one you should definitely use. A couple years back, AWS started offering free SSL certificates. Certificate Manager is where you get them. It will do a validation check on your domain (easier with Route53), and then grant you a free certificate to use in your AWS services.

Other important AWS stuff

- **AWS Identity and Access Management (IAM).** This is where you'll secure your account and grant permissions. Whether you go there or not, it's running and controlling your environment.
- **Simple Email Service (SES).**

Making sure your app is compliant

It seems like every day there's a new rule to comply with. It's not critical that you're compliant right when you're launching since you don't have users yet, but you should have a pretty good understanding of what's necessary just in case. We go over the basics below, but this is not legal advice— you should consult a

lawyer to ensure you're fully compliant.

Personally identifiable information (PII)

If you collect any personal data from your users, you should be compliant with data privacy rules.

Ways to get by for now (and that are just a good idea in general): specify a privacy policy, refrain from sharing and selling information, and secure all endpoints and access to PII using credentials.

Payment Card Industry Data Security Standard (PCI DSS)

PCI DSS was created to protect payment data. The PCI Security Standards Council recommends all businesses that accept or process payment cards do the following:

- Create a firewall to protect cardholder data
- Encrypt transmission of cardholder data
- Use and regularly update anti-virus software
- Implement strong access control measures
- Regularly test security systems and processes
- Maintain an information security policy

Even though PCI compliance is not technically a law, it is mandated by credit card companies and overseen by the FTC.

Basically, anyone who handles payment should adhere to PCI rules.

Ways to get by for now:

- Avoid accepting credit card or bank information on your website directly, but go through a third-party processor like Stripe. If they handle all the payments, you're secure.

- Never give out payment information, even after identity verification.
- Train your team to never take payment over the phone, email, chat, fax, mail, etc. Always use either checks or a third-party payment processor.

General Data Protection Regulation (GDPR)

You know how every single website these days has an opt-in cookie policy? Yeah, that's because of GDPR.

Basically, if anyone from the EU ever uses your app, you're subject to GDPR rules. Again, this might not be a concern for you at the moment, but considering its broad scope and potential fines (up to $20 million for small businesses), it's smart to keep GDPR in the back of your mind.

The 7 principles for GDPR compliance are:

- Lawfulness, fairness and transparency
- Purpose limitation
- Data minimization
- Accuracy
- Storage limitation
- Integrity and confidentiality (security)
- Accountability

When you're ready to tackle this, we like the Golden Data writeup on the 7 principles for GDPR compliance.

HIPAA

If you're handling patients' personal health information (PHI, the combination of PII and health information) specifically for a health services provider, you need to invest time into making sure you're compliant with HIPAA rules.

If you're handling PHI for an individual, e.g., if your app counts steps or collects

heart rates, you don't need to be HIPAA compliant.

If you fall into the "needs to be compliant" group, there are services out there you can outsource health information to for a relatively low fee.

Privacy policies and terms and conditions

Even though these aren't always mandated, it is vital that you have them in place, as they may still have legal implications.

There really are no good shortcuts here. To minimize risk, you should create these yourself with the help of an attorney, then post them and require acceptance from your users. It's worth it, as having them just might save your bacon.

To get an idea of what you'll need for yours, check out the privacy policies of your competitors or others in your space.

Privacy policy tips from the Better Business Bureau

- Make sure your privacy policy is visible.
- Keep the language simple.
- Only make promises you are prepared to keep.
- Keep the policy up to date.

Special actions that might be irreversible

Before you launch anything, think carefully about some of the actions you're about to take: some of them may be irreversible, or at least costly to change.

Encryption

Encryption can be a pain. Fortunately, AWS has an out-of-the-box AWS encryption tool, Key Management Store (KMS).

AWS KMS is available for a bunch of services, most of which we've listed below. I

recommend that you enable it on all the ones you use.

AWS KMS for EBS volumes

KMS encryption only works with EBS volumes. For this reason (and a host of others), I recommend using only EBS volumes.

EBS encryption must happen at creation time. **You cannot change this setting later.**

The key used to encrypt and decrypt your volumes is managed in Amazon's KMS. The service rotates your keys, keeps backups, and ensures your keys are safe and durably stored. It also works seamlessly with the block device on your EC2 instance—you just mount the volume like you normally would with a non-encrypted EBS volume.

You can opt to encrypt when creating an EBS volume or when creating an EC2 instance in the AWS console.

Similar options are available in the aws-cli using the --kms-key-id argument.

KMS for S3 buckets

You can't actually encrypt an entire S3 bucket, but you can encrypt individual objects in S3. That's why this is an irreversible action—adding encryption to all your S3 objects requires iterating through them all and reuploading to S3.

To enable encryption on a specific object, just set the option during upload.

KMS stores these keys, too.

You can use the AWS console or the aws-cli to encrypt your objects. But, more likely, you'll want to update your code to encrypt objects.

KMS encryption with other stuff

Encryption is available as a built-in, manually enabled option for the

following services:

- EFS file systems
- SQS messages
- DynamoDB
- RDS
- Kinesis streams
- Cloudfront distributions
- ELB

In every case, the encryption has to be enabled at creation time.

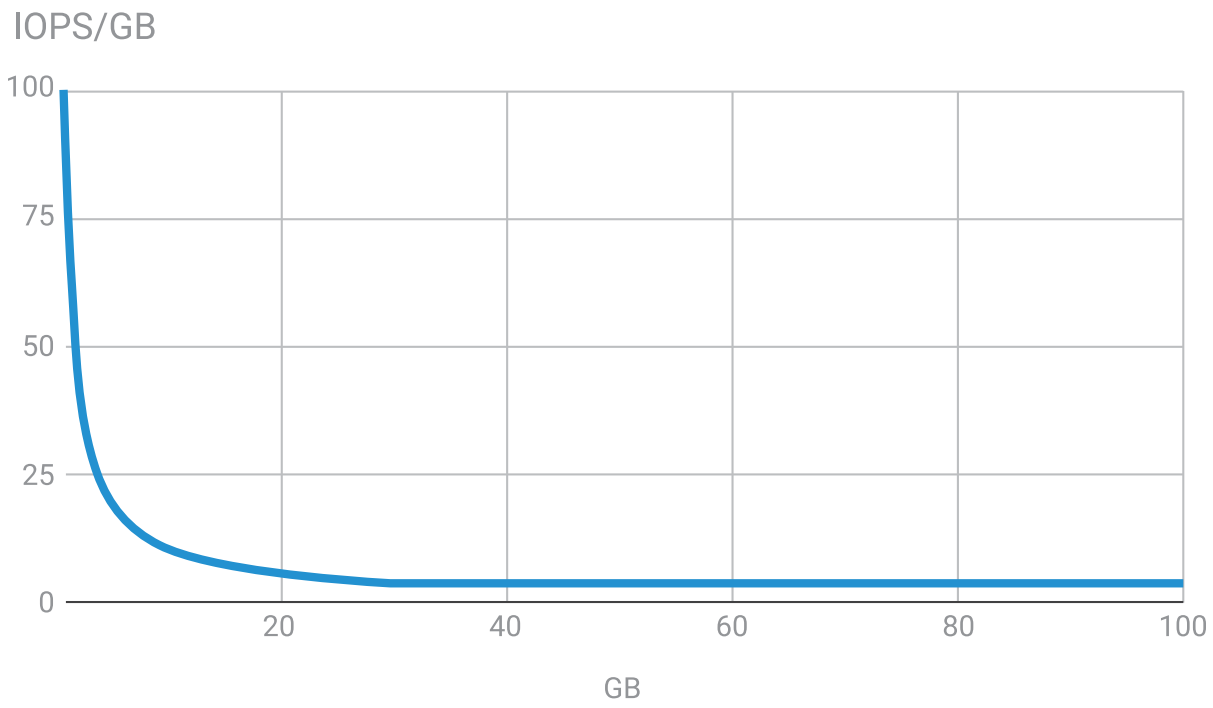
EBS volume provisioning

We recommend that you under-provision your EBS volumes. Hear us out.

Here's why you shouldn't over-provision your EBS volumes: The first problem is that the cost of the EBS volume is proportional to the size of the volume. If you over-provision, you're spending money. This is simple math. The second problem is that any snapshots (which you'll likely take) will incur a proportional price, too.

Then it gets more complicated.

For gp2 volumes (the primary type), you don't get more IOPS as you grow from 1GB to 33GB. The next graph displays this inverse proportionality. The decrease is due to the minimum of 100 IOPS for all gp2 volumes. Read another way, you've got to buy 34x what you'll probably need in your first 6 months to get any additional throughput, which you probably won't need for your first year.



When you over-provision, not only do you not get more throughput or need more size, but you also increase the time it takes to do snapshots. It also increases the chance that you exhaust your burst balance, exceed the hour threshold, and run multiple snapshots at the same time (assuming you're taking one an hour, but the principle applies regardless of your snapshot frequency). When multiple snapshots are running simultaneously, it's a downward spiral from there on performance.

These prior issues aren't even the worst—it's the management of a large volume.

If you start small and decide to grow the volume, it's not that big of a deal.

Going from big to small takes much more time and complexity. The only way to shrink a disk is to create a new disk of the appropriate size, and then copy or rsync over all the files. This takes much more time and is incredibly error-prone.

Volume management

If you're using EBS volumes and Linux, use LVM2.

Here's the simple way to get started.

```
#!/bin/bash
sudo apt-get update
sudo apt-get install lvm2
# mark your "physical volumes" (your EBS volumes)
sudo pvcreate /dev/xvdf
# create a group of physical volumes called a "volume group". Name it "main"
sudo vgcreate main /dev/xvdf
# create a block device (like a pretend EBS volume) called a "logical volume". Name it "vol1"
sudo lvcreate -l 100%FREE -n vol1 main
#### LVM DONE! ####
# create an xfs filesystem and mount it
sudo apt-get install xfsprogs
sudo mkfs.xfs /dev/main/vol1
sudo mkdir /vol1
sudo mount /dev/main/vol1 /vol1
#### LVM DONE! ####
# create an xfs filesystem and mount it
sudo apt-get install xfsprogs
sudo mkfs.xfs /dev/main/vol1
sudo mkdir /vol1
sudo mount /dev/main/vol1 /vol1
```

This code assumes Debian, an XFS file system, and that your EBS volume was mounted to /dev/sdf (converted to /dev/xvdf in the OS).

That's it! From this point on, don't even worry about LVM. It will just work. Then, in the future, if you need to update anything about your disks, look through the LVM docs to see if you can do it easily (chances are the answer is"yes").

Security keys

There are three types of keys that you should pay particular attention to when

dealing with AWS. This isn't rocket science, but it's also too easy to forget.

EC2 key pairs

Don't lose them. If you lose them, you'll have to terminate the machine. AWS will not help you regain access to the server. Doesn't matter if the server is a random job server or your primary database. It will be gone.

Root account multifactor authentication (MFA)

It's highly recommended that you enable MFA on your root AWS account credentials.

Access IDs and secret keys

It's far more likely that you'll share these too much than share them too little. Don't commit these to code.

Separate AWS accounts

Having few accounts limits your exposure to MFA, credentials, key pairs, VPNs, traffic routing, and more. Yet, with this number, you gain all the benefits of security, stability, disparate service limits, and simple cost-allocation reports.

To start, I would create a root billing account, a production account, and one development account per developer. The root billing account won't have any infrastructure in it but will serve as the parent account and cost reporting center. Lock this account down to a select few. The production account will host everything necessary for production, including shared services like email and DNS.

Reproducibility

Instead of setting up a tool like Terraform, Saltstack, or Chef, just create a new git repo and make a file for every type of server. Every command you run on the server gets copied from the repo.

Without this git repo, if anything were to happen to your infrastructure, you would

start from scratch all over again. At least this method gives you a leg up. It also helps your team members understand what you did, even if it's not a perfect history.

How to prep for the future

Thus far, we've talked about what you need to do now and what you can put off. In this section, we'll talk about those things you do need to consider about the future of your app. Some decisions you're making now can't be altered in the future, i.e., you will be "locked in." Here's how to plan ahead.

Request AWS service quotas now

You should request a pretty high quota right now. You won't fill it all for a little while, but you'll get capped if you hit the limit. It takes a long time for quota increase requests to go through, so it's definitely good to have a pretty big quota in place beforehand. Reach out to AWS support for a limit increase once you get any real amount of traffic.

Set up VPC

AWS VPC logically separates your application from the rest of AWS. AWS accounts have a default VPC, but you'll want to create a new VPC with a private subnet to run your application in so individual resources are not accessible over the internet. Then you'll create a public subnet that's accessible to the internet and can access your application's public interface. Using this configuration is considered a security best practice because it allows your users to access your app, but not its internals.

You'll want to set up your VPC before you create any resources for your application because those resources cannot be moved between subnets later.

Pick a region and services

You might be tempted to go for the region that's the cheapest, but resist that impulse. By going with a region that's not the one you're in, you are introducing

higher latency.

Start with the one you're closest to, and just the one you're closest to. You don't need to deal with additional regions when you're just starting out. You can always expand later on.

As for services, the same story here: choose the ones you need to get the job done and not any more than that. The more services you choose, the more setup you have to do, and the more you have to learn.

Once you choose a service, it's really hard to switch because your code is written to work with the service you've chosen. Keep that in mind as you're choosing your services. That sounds like a downside to AWS, but AWS isn't going away (and there's a large community of helpful people built around it), so just do it. If you decide to move later on, you can put the time in, but for now, just use AWS.

Vendor vs. open-source services

If you really want to keep things light, you could just use EC2 instances running open-source software rather than using a lot of AWS services—we don't recommend that, though. Open source is tempting due to costs or because the code is flexible. However, it's not that cost-effective. You still have to pay for the server it's on and someone to manage it, and you run the risk of not knowing how to manage or scale it, which could result in downtime. It's nice to have flexibility in code, but it's almost always more effort to change it than it would be to adapt your code to a paid AWS service.

Step 2: Secure

AWS security is an ongoing battle that you must address during every release, every change, and every CVE.

At minimum, you should complete the security assessment in the IAM console, set up and use private subnets in VPC, and enable Amazon's automated security agent, GuardDuty.

Take the security assessment in the AWS IAM console.

To see the security assessment, navigate in the console to the IAM service. On this dashboard, if you haven't completed the assessment, you'll see a section for "Security Status" with the following five items:

Delete your root access keys.

With your root keys, a hacker could delete your entire infrastructure, create a new user account for later access, launch new bitcoin mining resources in a different region, and more. You cannot restrict root access keys, but you can replace them with specific IAM users (addressed two sections down) or temporary keys using IAM.

To delete your root access keys, navigate to the root security credentials page and click "Access keys." Take a moment to make sure that no currently running production application depends on them by looking at the "Last Used" column in the table. Then, delete every row in the table. You won't need a single root access key.

Activate MFA on your root account.

Your root account should be protected by multi-factor authentication (MFA).

To enable MFA on the AWS root account, navigate to the root security credentials

page and click “Multi-factor authentication (MFA).” Once there, click “Activate MFA” and follow the steps in the wizard. If you want multiple people to have access to the root account (for disaster recovery), make sure all the people are there during the activation process, as they’ll all need to scan the QR code.

Create individual IAM users.

Every new person who needs access to the AWS console, AWS resources, or the aws-cli should receive it by creating a new IAM user for them specifically.

To create IAM users, navigate to AWS IAM users and click “Add User.” Then, follow the wizard to specify their username, console vs. API access, and then the exact permissions. The next step will be to place them in user groups, which give them permissions that are more manageable at scale.

Use groups to assign permissions.

You can specify permissions on individuals, but you’ll likely have an ops group and a dev group. Getting more granular is better, but also takes more time. It’s sufficient to create the groups, grant necessary permissions for now, and manage it later as your application grows.

Keep in mind: Only users can belong to groups. Roles must have their own specific permissions set.

To create a group, navigate to AWS IAM groups and click “Create New Group.” Then, follow the wizard to specify the group name and permission policies of that group. After it’s created, you’ll need to go back to your list of users and add them to the newly created group.

Apply an IAM password policy.

To set an IAM password policy, go to the IAM Account Settings page and click

"Set password policy." Choose from the list of options and click "Save changes."

Set up private subnets in VPC.

VPC makes it possible to specify public or private addressability on the public internet.

It's very difficult to change the public/private options once you've selected them. That's why I recommend setting up and using private subnets in VPC from the beginning. If you don't, it's a large security hole with a terrible migration process ahead.

The setup is a little more involved than the IAM security, and won't fit in this blog post without excessively bloating the topic. If you're familiar with networking, you could get through it without a tutorial. If you're new to networking, the whole thing, start to finish, will take less than 20 minutes with the right tutorial. Until I write one, here's the recommended guide from AWS on how to create a VPC with a public and private subnet.

Bastion hosts

A bastion host is a simple EC2 instance that lives in a public subnet and has SSH access open to the world (or your specific IP) and can connect on RDP or SSH to any other instance in your VPC.

How to set up a bastion host

Here's a checklist of things you'll need to do to set up a bastion host.

- Create your public/private subnets in a VPC (check the AWS docs linked above).
- Launch an EC2 instance in the public subnet. This is your bastion host.
- Adjust security groups to open SSH (port 22) to the bastion host from your local IP.
- Test SSH to the bastion host. If it doesn't work, check your route tables, IP address, selected subnet, and ssh daemon.
- Launch an EC2 instance in the private subnet. This is your private server.

- Adjust security groups to open SSH (port 22) to the private server from the bastion host.
- SSH to the bastion host, and then test SSH from the bastion host to the private server. If it doesn't work, check your private route tables, IP address, selected subnet, and username/keypair.

Here's a helpful tutorial on how to use a bastion host to connect to a private server using RDP.

Activate AWS GuardDuty.

Amazon's GuardDuty is the easiest security monitoring tool you'll ever enable in AWS. Its purpose is to protect your AWS accounts and workloads with intelligent threat detection and continuous monitoring. While I doubt that it's as good as other third-party tools, it will be good enough to warrant 1 minute of clicking and a negligible price per month.

Here's how to enable AWS GuardDuty from start to finish.

- Step one: Navigate to the tool.
- Step two: Make sure your primary region is selected (GuardDuty must be enabled on a region-by-region basis).
- Step three: Click "Get started"
- Step four: Click "Enable GuardDuty."

That's it!

There are a couple things to note.

- You need to enable GuardDuty once for every region you want it to run in.
- You need to enable GuardDuty in every account you want it to run in. If you have multiple production accounts, enable it in all applicable regions in all applicable production accounts.
- GuardDuty does not need VPC flow logs, DNS logs, or CloudTrail logs to be specifically enabled. It works on the underlying data.
- GuardDuty sends findings to CloudWatch Events.

It’s important to set up GuardDuty as well as get notifications. If you’re not addressing notifications, you may as well not set it up.

Testing

I know it's tempting to start playing with automation right off the bat, but you should first make sure your application actually works by running it manually.

When you first start running your application on real infrastructure, you'll want to test along the way in as small of increments as possible. That way if something goes wrong, you know where it happened. Start by running your application manually on the command line. Test it and make sure everything works before turning it into a service or daemon.

Domains

As you're deploying your web app, you will inevitably use DNS. If you're deploying on AWS, then you should be using Route 53 for a couple of reasons: it uses IAM for authentication; it tightly integrates with EC2, S3, CloudFront, and more; and it has smart mechanisms for global failover and health checks. Even if you don't use all of those features now, it's so simple to switch and cheap to run that you should do it anyway.

DNS explained

DNS stands for "domain name system." Here are the basics: All computer networks use IP addresses (numbers) as unique locators. DNS is the system that converts domain names and subdomains into those IP addresses. DNS is usually available on port 53 over TCP and UDP but runs as a normal process on a normal server.

Route 53 is simply Amazon's implementation of DNS, including configuration and integration with other AWS services. Just like every other DNS provider, AWS Route 53 allows you to manage domain names, subdomains, and other types of records. You can manage Route 53 in the AWS console, through API calls, and

through the aws-cli command-line utility.

How to make Route 53 authoritative

A DNS provider needs to be authoritative. If Route 53 is not authoritative, then your changes will not be reflected on the public internet.

There are only three ways to make your Route 53 domains authoritative.

Option 1: Buy your Route 53 domain

If you haven't already bought your domain, then just buy it straight from Route 53. It's straightforward to search for domains and purchase them.

Go to the domain registration page, pick your domain, and buy it. And you're done! It will automatically get set up in Route 53 in the following 24 hours, where it will be authoritative and ready for action.

Option 2: Transfer your domain to Route 53's registrar

Amazon has a wizard for requesting a domain transfer. To initiate a transfer, go to the Registered Domains tab on Route 53 and click "Transfer Domain."

The wizard will check for things like transfer locks, which can be adjusted in your current registrar.

Make sure you copy all your DNS records over to the new hosted zone as soon as you can. If you don't, you'll have some downtime.

Option 3: Update the NS records in your existing registrar

Skip this section if you can do either of the previous two—they're MUCH better!

1. Create a new Hosted Zone in Amazon Route 53.
2. Duplicate the DNS in your registrar to Route 53.
3. Change the NS records.

4. Wait for about 24 hours.
5. Verify that it's working.
6. Delete all old records in the registrar.

If you need help, contact the registrar.

Hosted zones and DNS records

Every DNS record has 4 parts: the name (app.bluematador.com), the type (A), the value (3.219.17.34), and the TTL in seconds (59).

Let's go over the most-common DNS record types.

- A specifies an IPv4 address like "3.219.17.34". If multiple A records with the same name are found, the client will select one at random.
- AAAA specifies an IPv6 address like "2601:681:8100:8f0:8dd5:3d1c:7f1f:62ff". If multiple AAAA records are found, the client will select one at random.
- CNAME specifies another CNAME, A, or AAAA DNS record where the correct values can be found. Think of CNAME as a pointer or a redirect. An example record would be "app2.bluematador.com CNAME app.bluematador.com."

There are more DNS record types than this, but those are the ones you'll primarily use in deploying your application.

As for the name of records, it will always be a subdomain, like "dev.bluematador.com" or "www.bluematador.com." Every subdomain can point at a single application, and there is no cost to the number of subdomains hosted.

Finally, the TTL will seem arbitrary. In large part, it is, except that low values are required by AWS to allow changing infrastructure. I recommend using 1 minute in all A, AAAA, and CNAME records.

Route 53 alias records

DNS doesn't allow CNAMEs to be used in the apex record. You can't make a CNAME from "bluematador.com" (the apex—meaning "no subdomain") to "www.

bluematador.com" unless you use an alias record.

An alias record is a symlink, which is a self-resolving record that doesn't take an extra hop by the client to resolve.

You will use alias records for static website hosting in S3, CloudFront distributions, ELBs, and apex domains.

How to use Amazon Route 53

The nice thing about Route 53 is that it's so simple if you use just its core feature set—DNS records for hosted zones. Integrating it with other services is straightforward and fast to do.

Let me give you some examples.

- For an EC2 instance, use an A record with the instance's public IP address as the value. The TTL should be 60 seconds. If you're using an elastic IP address (you probably should), then the TTL can be 3600 seconds (an hour).
- For an S3 bucket, use a CNAME record to "<bucketname>.s3.<region>.amazonaws.com". The TTL may be 3600 seconds. Your bucket name must match exactly the name of the subdomain—if your bucket name is something like "mybucket," then it won't work. For a subdomain of "cdn.bluematador.com", your bucket name must be "cdn.bluematador.com", and your CNAME would be "cdn.bluematador.com.s3.us-east-1.amazonaws.com". Keep in mind that SSL will not work on that URL (the certificate won't match the URL). To get SSL with your subdomain on S3, you'll have to use CloudFront in front of S3.
- For an S3 bucket with website hosting, use an A record with alias to the bucket. Using an alias record will provide a drop-down option in Route 53. If it doesn't appear, refresh your page. The name of the bucket must be identical to the subdomain you're using, same as the S3 bucket in the paragraph above. Alias entries don't need a TTL.
- For an ELB, ALB, CLB, or NLB, use an A record with an alias to the ELB. Again, an alias won't require a TTL and will provide a dropdown option for the ELB's alias target.
- For a CloudFront distribution, use an A record with alias to the distribution.
- For an API gateway, use an A record with alias to the gateway.

Other Route 53 features

You probably won't use these now, but I want to bring it up so you can be aware of it and use it later.

Other DNS record types

- MX stands for "Mail eXchange" and is used to identify an email server. Your email provider will have documentation around how to set these up.
- TXT is plain text, and is only really good for validating domain ownership with tools like Google Analytics or email validation.
- SPF stands for "Sender Policy Framework" and can help increase your email deliverability. I won't go into this here, but since you're making a web app, you should know what this is.
- SOA stands for "Start of Authority" and looks cryptic. You'll have to look up the exact syntax of this value, but it's helpful to reduce negative TTLs—the time you spend waiting for a new record to appear on the public internet. The default is usually a day, meaning that if you query "abc.bluematador.com" and then add it to Route 53, it will take a day to show up. If you reduce the negative TTL in the SOA record, you can reduce that time spent waiting.

Private hosted zones

If you want an IP address for an internal database server, but don't want that IP available to the public, use an internal zone, and associate it with your VPC.

DNS health checks

You can (and should) also use Route 53 to manage traffic between the same application hosted in different regions. You can route traffic by latency, shortest path, or health. This is all set up in Route 53.

Connecting to data stores

In order to connect your app to your data stores, you'll need to set up IAM roles that have the right permissions. It's pretty easy to set up IAM in RDS, and Amazon has a bunch of resources to help you do that. You'll also want to configure your

application with connection information for your data stores.

Serving static assets

At Blue Matador, our static assets are sitting in S3 so they're secure, quick to grab, less vulnerable to data loss, and scalable.

In front of that, we've set up CloudFront as our CDN to improve availability and reduce latency.

This setup works really well for us, but when you're just starting out, we recommend that you set up S3 on its own first. You can always set up CloudFront later—here's a handy guide to do that.

Load balancing

Instead of pointing incoming requests directly at a single server, point them at an elastic load balancer (ELB). Your load balancer will then redistribute that traffic across all your server instances to make sure no one instance gets overwhelmed.

Even if you don't have a lot of instances right now, it's a good idea to set up your ELB now so it's ready when you add more servers.

Database migrations

Your databases are constantly changing—whether you're updating existing tables or adding new ones—and you need a way to keep them up to date in production.

The simplest way is just to keep those migrations in a SQL file that's versioned. In that file, just make your notes, e.g., "here are the commands you run to make this work." You don't need a complicated system—just keep a file with the commands handy.

Step 4: Optimize

Starting with automation

It's incredibly helpful to have an automated build and deploy system that consistently builds correctly.

AWS has a build and deployment automation tool built-in: AWS CodePipeline.

It includes:

- CodeCommit (a git repository)
- CodeBuild (build tool)
- CodeDeploy (deployment tool)
- CodePipeline (ties all the tools together; adds steps, approvals, staging, etc.)

On top of all that, you can also integrate any of your existing tools with CodePipeline.

AWS cost management vs. availability

You're going to be drawn to adding a bunch of resources in multiple AWS availability zones and maybe even regions. You'll be tempted to use the costliest tools to get the job done because they seem better. Skip spending time on this in favor of better protecting your data (which we'll discuss below).

- You don't need high availability, so don't launch more resources than you need.
- You don't need hot backups, so don't make big database clusters.
- You're not sure what instance sizes you'll need, so don't buy any reservations that last longer than three months. (I've been guilty of this one myself.)
- You don't need all the cost classes in CloudFront. You probably don't even need CloudFront. Ignore CDN for now.

Disaster prevention and recovery

One area where you can't skimp is safeguarding customer data. You can lose servers, databases, and active traffic—a pain, but ultimately not catastrophic. But losing customer data will cost you customers.

So it's a good idea to have a backup plan. Take a look at every place you're storing data and make an AWS disaster recovery strategy. Here are some ideas.

S3

- Replicate your data across regions.
- Consider restricting deletes in IAM or S3 configuration.
- Implement S3 object versioning.

EC2 instances and EBS volumes

- Only put data on EBS volumes. EC2 instances do come with a little bit of memory, but if you get rid of the instance, you lose everything on it.
- Create snapshots as required.
- But first, make sure you pause all writes, flush data to disk, sync file system, and lock the file system. Only then create your snapshot. Failure to take any of those measures may result in an unrecoverable snapshot.
- Use EBS snapshot lifecycles to manage backups.

RDS

- Configure your snapshots.

DynamoDB

- Configure backups.

SQS and Kinesis

- Keep messages long enough to catch up.
- Consider streaming to S3 as a failsafe.

Everything else

- Read the documentation for every service you're using.
- Configure backups.

This all feels like no-brainer information, but it's all too easy to let it slip until something goes wrong. Take the time to prevent that from happening.

Configuration management

For every type of server (database, Kubernetes, etc.), have the following:

- AMI
- IAM role/permissions
- Security group, subnet
- Instance type and size
- Launch script or copy/paste commands to run for configuration, programs, etc.

For every Lambda:

- Language and version
- Environment variables

For every other service:

- Critical configuration
- Versions
- Permissions

Basically what you want at the end of this is a way for someone else to reproduce the environment if they need to (as mentioned in the Security chapter). Only record the things that matter, and keep them up to date: If you change a script, configuration, or value, update your repo. If you need a new resource, refer to the repo.

Step 5: Monitor

Hearing that you're down from an end-user has to be one of the worst feelings ever. Ideally, if something is going to go wrong, you want to know about it first.

The cheap option

There are a bunch of services out there that will monitor different aspects of your infrastructure. CloudWatch is AWS's proprietary monitoring tool, so if you don't want to spend time looking into other tools, stick with that.

You can set up custom alerts in CloudWatch for pretty much anything you want. For now, focus on what's most critical and expand your coverage as you go.

The easy option

If you'd rather go for an automated monitoring solution, Blue Matador monitors tons of AWS services without any alarm creation required on your part.

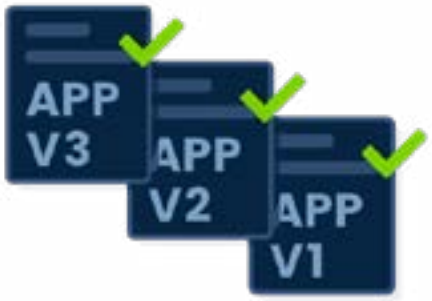
bluematador

Alerts without the toil.

The typical, manual monitoring approach requires significant time and toil to manage every alert on every resource. Blue Matador eliminates the need to manually set up alerts by automatically configuring full monitoring coverage out-of-the-box.

Know about critical production issues.

Issues in your infrastructure can pop-up at any time and could be caught unaware. Blue Matador identifies previously unknown issues, ensuring you see the problems first—instead of hearing about them from your customers.



Deploy faster, rest easier.

Agile teams are looking to rapidly deliver features and delight customers, but that leaves little time to configure proper alerting. Blue Matador supports agile teams deploying multiple times per day by ensuring that they will be alerted of any potential issues.

MAKE MONITORING EASIER

START YOUR FREE TRIAL TODAY

START YOUR FREE TRIAL


bluematador