Production Readiness Checklist

This checklist is your guide to the best practices for deploying secure, scalable, and highly available infrastructure in AWS. Before you go live, go through each item, and make sure you haven't missed anything important!

Your checklist state will be saved to Local Storage.

Server-side

Build AMIs

If you want to run your apps directly on EC2 Instances, you should package them as Amazon Machine Images (AMIs) using a tool such as Packer. Although we recommend Docker for all stateless apps (see below), we recommend directly using AMIs and EC2 Instances for all stateful apps, such as any data store (MySQL, MongoDB, Kafka), and app that writes to its local disk (e.g., WordPress, Jenkins).

Deploy AMIs using Auto Scaling Groups

The best way to deploy an AMI is typically to run it as an Auto Scaling Group.

This will allow you to spin up multiple EC2 Instances that run your AMI, scale the number of Instances up and down in response to load, and automatically replace failed Instances.

Build Docker images

If want to run your apps as containers, you should package your apps as Docker images and push those images to Amazon's Elastic Container Registry (ECR). We recommend Docker for all stat eless apps and for local development (along with Docker Compose).

Deploy serverless apps using Lambda and API Gateway

If you want to build serverless apps, you should package them as deployment packages for AWS Lambda. You can expose your Lambda functions as HTTP endpoints using API Gateway.

Configure hard drives

Configure the root volume on each EC2 Instance with enough space for your app and log files. Note that root volumes are deleted when an Instance is terminated, so if you are running stateful apps that need to persist data between redeploys (or between crashes), attach one or more EBS Volumes.

Client-side

Pick a JavaScript framework

If you are building client-side applications in the browser, you may wish to use a JavaScript framework such as React, Angular, or Ember. You'll need to update your build system to build and package the code appropriately(see continuous integration).

Pick a compile-to-JS language

JavaScript has a number of problems and limitations, so you may wish to use a compile-to-JS language, such as TypeScript, Scala.js, PureScript, Elm, or ClojureScript. You'll need to update your build system to build and package the code appropriately (see continuous integration).

Pick a compile-to-CSS language

CSS has a number of problems and limitations, so you may wish to use a compile-to-CSS language,

such asSASS, less, cssnext, or postcss. You'll need to update your build system to build and package the code appropriately (see continuous integration).

Use a static content server

You should serve all your static content (CSS, JS, images, fonts) from a static content server so that your dynamic web framework (e.g., from Rails, Node.js, or Django) can focus solely on processing dynamic requests. The best static content host to use with AWS is S3.

Use a CDN

Use CloudFront as a Content Distribution Network (CDN) to cache and distribute your content across servers all over the world. This signicantly reduces latency for users and is especially effective for static content.

Configure caching

Think carefully about versioning, caching, and cache-busting for your static content. One option is to put the version number of each release directly in the URL (e.g., /static/v3/foo.js), which is easy to implement, but means 100% of your content is "cache buested" each release. Another option is "asset fingerprinting," where the build system renames each static content file with a hash of that files contents (e.g., foo.js becomes 908e25f4bf641868d8683022a5b62f54.js), which is more complicated to implement (note: many build systems have built-in support), but ensures that only content that has changed is ever cache busted.

Data storage

Deploy relational databases

Use Amazon's Relational Database Service (RDS) to run MySQL, PostgreSQL, Oracle, SQL Server, or MariaDB. Consider Amazon Aurora as a highly scalable, cloud-native, MySQL and PostgreSQL compatible database. Both RDS and Aurora support automatic failover, read replicas, and automated backup.

Deploy NoSQL databases

Use Elasticache if you want to use Redis or Memcached for key-value storage. Use DynamoDB if you need a managed, eventually consistent document store. If you need other NoSQL databases, such as MongoDB or Couchbase, you'll need to run it yourself (see the Gruntwork Library).

Deploy queues

Use Amazon Simple Queue Service (SQS) as a managed, distributed queue.

Deploy search tools

Use Amazon Elasticsearch for log analysis and full text search. Note that Amazon Elasticsearch has somesignificant limitations, so if you need to work around those, you'll need to run the ELK stack yourself (see the Gruntwork Library).

Deploy stream processing tools

Use Amazon Kinesis to process streaming data. Note that Kinesis has some significant limitations, so if you need to work around those, you'll need to instead run Kafka yourself (see the Gruntwork Library).

Deploy a data warehouse

Use Amazon Redshift for data warehousing.

Deploy big data systems

Use Amazon EMR to run Hadoop, Spark, HBase, Presto, and Hive.

Set up cron jobs

Use AWS Lambda Scheduled Events or ECS Scheduled Tasks to reliably run background jobs on a schedule (cron jobs). Look into AWS Step Functions to build reliable, multi-step, distributed workflows.

Configure disk space

Configure enough disk space on your system for all the data you plan to store. If you are running a data storage system yourself, you'll probably want to store the data on one or more EBS Volumes that can be attached and detached as Instances are replaced. Note: using EBS Volumes with Auto Scaling Groups (ASGs) is very tricky, as ASGs can launch an Instance in any Availability Zone, but an EBS Volume can only be attached from the same Availability Zone (see the Gruntwork Library for solutions).

Configure backup

Configure backup for all of your data stores. Most Amazon-managed data stores, such as RDS and Elasticache, support automated nightly snapshots. For backing up EC2 Instances and EBS Volumes, consider running ec2-snapper on a scheduled basis.

Configure cross-account backup

Copy all of your backups to a separate AWS account for extra redundancy. This ensures that if a disaster happens in one AWS account—e.g., an attacker gets in or someone accidentally deletes all the backups—you still have a copy of your data available elsewhere.

Test your backups

If you never test your backups, they probably don't work. Create automated tests that periodically restore from your backups to check they are actually working.

Set up schema management

For data stores that use a schema, such as relational databases, define the schema in schema migration files, check those files into version control, and apply the migrations as part of the deployment process.

See Flyway and Liquibase.

Scalability and High Availability

Choose between a Monolith and Microservices

Ignore the hype and stick with a monolithic architecture as long as you possibly can. Microservices have massive costs (operational overhead, performance overhead, more failure modes, loss of transactions/atomicity/consistency, difficulty in making global changes, backwards compatibility requirements), so only use them when your company grows large enough that you can't live without one of the benefits they provide (support for different technologies, support for teams working more independently from each other). See Don't Build a Distributed Monolith, Microservices — please, don't, and Microservice trade-offs for more info.

Configure service discovery

If you do go with microservices, one of the problems you'll need to solve is how services can discover the IPs and ports of other services they depend on. Some of the solutions you can use include Load Balancers, ECS Service Discovery, and Consul.

Use multiple Instances

Always run more than one copy (i.e., more than one EC2 Instance or Docker container) of each stateless application. This allows you to tolerate the app crashing, allows you to scale the number of copies up and down in response to load, and makes it possible to do zero-downtime deployments.

Use multiple Availability Zones

Configure your Auto Scaling Groups and Load Balancers to make use of all Availability Zones (AZs) in your AWS account so you can tolerate the failure of an entire AZ.

Set up load balancing

Distribute load across your apps and Availability Zones using Amazon's managed Load Balancers, which are designed for high availability and scalability. Use the Application Load Balancer (ALB) for all HTTP/HTTPS traffic and the Network Load Balancer (NLB) for everything else.

Use Auto Scaling

Use auto scaling to automatically scale the number of resources you're using up to handle higher load and down to save money when load is lower.

Configure Auto Recovery

Configure a process supervisor such as systemd or supervisord to automatically restart failed processes. Configure your Auto Scaling Groups to use a Load Balancer for health checks and to automatically replace failed EC2 Instances. Use your Docker orchestration tool to monitor the health of your Docker containers and automatically restart failed ones (e.g., ECS Health Checks).

Configure graceful degradation

Handle failures in your dependencies (e.g., a service not responding) by using graceful degradation patterns, such as retries (with exponential backoff and jitter), circuit breaking, timeouts, deadlines, and rate limiting.

Perform load tests and use chaos engineering

Run load tests against your infrastructure to figure out when it falls over and what the bottlenecks are. Usechaos engineering to continuously test the resilience of your infrastructure (see also chaos monkey).

Continuous Integration

Pick a Version Control System

Check all code into a Version Control System (VCS). The most popular choice these days is Git. You can useGitHub, GitLab, or BitBucket to host your Git repo.

Do code reviews

Set up a code review process in your team to ensure all commits are reviewed. Pull requests are an easy way to do this.

Configure a build system

Set up a build system for your project, such as Gradle (for Java), Rake (for Ruby), or Yarn (for Node.js). The build system is responsible for compiling your app, as well as many other tasks described below.

Use dependency management

Your build systems should allow you to explicitly define all the of the dependencies for your apps.
Each dependency should be versioned, and ideally, the versions of all dependencies, including transitive dependencies, are captured in a lock file (e.g., read about Yarn's lock file and Go's dep lock file.

Configure static analysis

Configure your build system so it can run static analysis tools on your code, such as linters and code coverage.

Set up automatic code formatting

Configure your build system to automatically format the code according to a well-defined style (e.g., with Go, you can run go fmt; with Terraform, you can run terraform fmt). This way, all your code has a consistent style, and your team doesn't have to spend any time arguing about tabs vs spaces or curly brace placement.

Set up automated tests

Configure your build system so it can run automated tests on your code, with tools such as JUnit (for Java), RSpec (for Ruby), or Mocha (for Node.js).

Publish versioned artifacts

Configure your build system so it can package your app into a deployable "artifact," such as an AMI or Docker image. Each artifact should be immutable and have a unique version number that makes it easy to figure out where it came from (e.g., tag Docker images with the Git commit ID). Push the artifact to an artifact repository (e.g., ECR for Docker images) form which it can be deployed.

Set up a build server

Set up a server to automatically run builds, static analysis, automated tests, etc. after every commit. You can use a hosted system such as CircleCI or Travis CI, or run your a build server yourself with a tool such as Jenkins.

Continuous Delivery

Create deployment environments

Define separate "environments" such as dev, stage, and prod. Each environment can either be a separate AWS account (recommended for larger teams and security-sensitive and compliance use cases) or a separate VPC within a single AWS account (recommended only for smaller teams).

Set up per-environment configuration

Your apps may need different configuration settings in each environment: e.g., different memory settings, different features on or off. Define these in config files that get checked into version control (e.g., dev-config.yml, stage-config.yml, prod-config.yml) and packaged with your app artifact (i.e., packaged directly into the Docker image for your app), and have your app boot up code pick the proper config file for the current environment during boot.

Define your infrastructure as code

Do not deploy anything by hand, by using the AWS Console, or the AWS CLI. Instead, define all of your infrastructure as code using tools such as Terraform, Packer, and Docker.

Test your infrastructure code

If all of your infrastructure is defined as code, you can create automated tests for it. The goal is to verify your infrastructure works as expected after every single commit, long before those infrastructure changes affect prod. See Terratest for more info.

Set up immutable infrastructure

Don't update EC2 Instance or Docker containers in place. Instead, launch completely new EC2 Instances and new Docker containers and, once those are up and healthy, remove the old EC2 Instances and Docker images. Since we never "modify" anything, but simply replace, this is known as immutable infrastructure, and it makes

it easier to reason about what's deployed and to manage that infrastructure.

Promote artifacts

Deploy immutable artifacts to one environment at a time, and promote it to the next environment after testing. For example, you might deploy v0.3.2 to dev, and test it there. If it works well, you promote the exact same artifact, v0.3.2, to stage, and test it there. If all goes well, you finally promote v0.3.2 to prod. Since it's the exact same code in every environment, there's a good chance that if it works in one environment, it'll also work in the others.

Roll back in case of failure

If you use immutable, versioned artifacts as your unit of deployment, then any time something goes wrong, you have the option to roll back to a known-good state by deploying a previous version. If your infrastructure is defined as code, you can also see what changed between versions by looking at the diffs in version control

Automate your deployments

One of the advantages of defining your entire infrastructure as code is that you can fully automate the deployment process, making deployments faster, more reliable, and less stressful.

Do zero-downtime deployments

There are several strategies you can use for Zero-downtime deployments, such as blue-green deployment (works best for stateless apps) or rolling deployment (works best for stateful apps).

Use canary deployments

Instead of deploying the new version of your code to all servers, and risking a bug affecting all users at once, you limit the possible damage by first deploying te new code to a single "canary" server. You then compare the canary to a "control" server running the old code and make sure there are no unexpected errors, performance issues, or other problems. If the canary looks healthy, roll out the new version of your code to the rest of the servers. If not, roll back the canary.

Use feature toggles

Wrap all new functionality in an if-statement that only evaluates to true if a the feature toggle is enabled. By default, all feature toggles are disabled, so you can safely check in and even deploy code that isn't completely finished (as long as it compiles!), and it won't affect any user. When the feature is done, you can

use a UI to gradually enable the feature toggle for specific users: e.g., initially just for your company's employees, then for 1% of all users, then 10% of all users, and so on. At any stage, if anything goes wrong, you can turn the feature toggle off again. Feature toggles allow you to separate deployment of new code from the release of new features in that code. They also allow you to do bucket testing. See LaunchDarkly,Split, and Optimizely for more info.

Networking

Set up VPCs

Don't use the Default VPC, as everything in it is publicly accessible by default. Instead, create one or more custom Virtual Private Clouds (VPC), each with their own IP address range (see VPC and subnet sizing), and deploy all of your apps into those VPCs

Set up subnets

Create three "tiers" of subnets in each VPC: public, private-app, private-persistence. The public subnets are directly accessible from the public Internet and should only be used for a small number of highly locked down, user-facing services, such as load balancers and Bastion Hosts. The private-apps subnets are only accessible from within the VPC from the public subnets and should be used to run your apps (Auto Scaling Groups, Docker containers, etc.). The private-persistence subnets are also only accessible from within the VPC from the private-app subnets (but NOT the public subnets) and should be used to run all your data stores (RDS, ElastiCache, etc.). See A Reference VPC Architecture.

Configure Network ACLs

Create Network Access Control Lists (NACLs) to control what traffic can go between different subnets. We recommend allowing the public subnets to receive traffic from anywhere, the private-app subnets to only receive traffic from the public subnets, and the private-persistence subnets to only receive traffic from the private-app subnets.

Configure Security Groups

Every AWS resource (e.g., EC2 Instances, Load Balancers, RDS DBs, etc.) has a Security Group that acts as a firewall, controlling what traffic is allowed in and out of that resource. By default, no traffic is allowed in

or out. Follow the Principle of Least Privilege and open up the absolute minimum number of ports you can for each resource. When opening up a port, you can also specify either the CIDR block (IP address range) or ID of another Security Group that is allowed to access that port. Reduce these to solely trusted servers where possible. For example, EC2 Instances should only allow SSH access (port 22) from the Security Group of a single, locked-down, trusted server (the Bastion Host).

Configure Static IPs

By default, all AWS resources (e.g., EC2 Instances, Load Balancers, RDS DBs, etc.) have dynamic IP addresses that could change over time (e.g., after a redeploy). When possible, use Service Discovery to find the IPs of services you depend on. If that's not possible, you can create static IP addresses that can be attached and detached from resources using Elastic IP Addresses (EIPs) for public IPs or Elastic Network Interfaces (ENIs) for private IPs.

Configure DNS using Route 53

Manage DNS entries using Route 53. You can buy public domain names using the Route 53 Registrar or create custom private domain names, accessible only from within your VPC, using Route 53 Private Hosted Zones.

Security

Configure encryption in transit

Encrypt all network connections using TLS. Many AWS services support TLS connections by default (e.g., RDS) or if you enable them (e.g., ElastiCache). You can get free, auto-renewing TLS certificates for your public domain names from AWS Certificate Manager (ACM). You can also use the ACM Private Certificate Authority to get auto-renewing TLS certificates for private domain names within your VPC.

Configure encryption at rest

Encrypt the root volume of each EC2 Instance by using the encrypt_boot setting in Packer. Enable encryption for each EBS Volume too. Many AWS services optionally support disk encryption: e.g., seeEncrypting Amazon RDS Resources and ElastiCache for Redis At-Rest Encryption.

Set up SSH access

Do NOT share EC2 KeyPairs with your team! Otherwise, everyone will be using the same username and key

for server acesss (so there's no audit trail), the key may easily be compromised, and if it is, or someone leaves the company, you'll have to redeploy ALL your EC2 Instances to change the KeyPair. Instead, configure your EC2 Instances so that each developer can use their own username and SSH key, and if that developer leaves the company, the key can be invalidated immediately (see the Gruntwork Library for solutions).

Deploy a Bastion Host

All EC2 Instances should be in a private subnet and NOT accessible directly from the public Internet. Only a single, locked-down EC2 Instance, known as the Bastion Host, should run in the public subnets. You must first connect to the Bastion Host, which gets you "in" to the network, and then you can use it as a "jump host" to connect to the other EC2 Instances.

Deploy a VPN Server

We typically recommend running a VPN Server as the entrypoint to your network (as the Bastion Host). OpenVPN is the most popular option for running a VPN server.

Set up a secrets management solution

NEVER store secrets in plaintext. Developers should store their secrets in a secure secrets manager, such aspass, 1Password, or LastPass. Applications should store all their secrets (such as DB passwords and API keys) either in files encrypted with KMS or in a secret store such as Vault.

Use server hardening practices

Every server should be hardened to protect against attackers. This may include: running CIS Hardened Images, fail2ban to protect against malicious access, unattended upgrades to automatically install critical security patches, firewall software, anti-virus software, and file integrity monitoring software. See also My First 10 Minutes On a Server and Guide to User Data Security.

Go through the OWASP Top 10

Browse through the Top 10 Application Security Risks list from the Open Web Application Security Project (OWASP) and check your app for vulnerabilities such as injection attacks, CSRF, and XSS.

Go through a security audit

Have a third party security service perform a security audit and do penetration testing on your services.

Fix any issues they uncover.

Sign up for security advisories

Join the security advisory mailing lists for any software you use and monitor those lists for announcements of critical security vulnerabilities.

Create IAM Users

Create an IAM User for each developer. The developer will have a web console login for accessing AWS from a web browser and a set of API keys for accessing AWS from the CLI. Note that the root user on your AWS account should only be used to create an initial admin IAM User; after that, do all your work from that IAM user account and never use the root user account again!

Create IAM Groups

Manage permissions for IAM users using IAM Groups. Follow the Principle of Least Privilege, assigning the minimum permissions possible to each IAM Group and User.

Create IAM Roles

Give your AWS resources (e.g., EC2 Instances, Lambda Functions) access to other resources by attachingIAM Roles. All AWS SDK and CLI tools automatically know how to use IAM Roles, so you should never have to copy AWS access keys to a server.

Create cross-account IAM Roles

If you are using multiple AWS accounts (e.g., one for dev and one for prod), you should define all of the IAM Users in one account, and use IAM Roles to provide access to the other AWS accounts. This way, developers have only one set of credentials to manage, and you can have very fine-grained permissions control over what IAM Users can do in any given account.

Create a password policy and enforce MFA

Set a password policy that requires a long password for all IAM users and require every user to enableMulti-Factor Authentication (MFA).

Record audit Logs

Enable CloudTrail to maintain an audit log of all changes happening in your AWS account.

Monitoring

Track availability metrics

The most basic set of metrics: can a user access your product or not? Useful tools: Route 53 Health Checksand Pingdom.

Track business metrics

Metrics around what users are doing with your product, such as what pages they are viewing, what items they are buying, and so on. Useful tools: Google Analytics, Kissmetrics, and Mixpanel.

Track application metrics

Metrics around what your application is doing, such as QPS, latency, and throughput. Useful tools:CloudWatch, DataDog, and New Relic.

Track server metrics

Metrics around what your hardware is doing, such as CPU, memory, and disk usage. Useful tools:CloudWatch, DataDog, New Relic, Nagios, Icinga, and collectd.

Configure services for observability

Record events and stream data from all services. Slice and dice it using tools such as Kafka and KSQL, Honeycomb, and OpenTracing.

Store logs

To prevent log files from taking up too much disk space, configure log rotation on every server using a tool such as logrotate. To be able to view and search all log data from a central location (i.e., a web UI), set up log aggregation using tools such as CloudWatch Logs, Filebeat, Logstash, Loggly, and Papertrail.

Set up alerts

Configure alerts when critical metrics cross pre-defined thresholds, such as CPU usage getting too high or available disk space getting too low. Most of the metrics and log tools listed earlier in this section support

alerting. Set up an on-call rotation using tools such as PagerDuty and VictorOps.

Cost optimization

Pick proper EC2 Instance types and sizes

AWS offers a number of different Instance Types, each optimized for different purposes: compute, memory, storage, GPU, etc. Use EC2Instances.info to slice and dice the different Instance Types across a variety of parameters. Try out a variety of Instance sizes by load testing your app on each type and picking the best balance of performance and cost. In general, running a larger number of smaller Instances ("horizontal scaling") is going to be cheaper, more performant, and more reliable than a smaller number of larger Instances ("vertical scaling"). See also How Netflix Tunes EC2 Instances for Performance.

Use Spot EC2 Instances for background jobs

EC2 Spot Instances allow you to "bid" a much lower price for EC2 Instances than what you'd pay on-demand (as much as 90% lower!), and when there is capacity to fulfill your request, AWS will give you the EC2 Instances at that price. Note that if AWS needs to relcaim that capacity, it may terminate the EC2 Instance at any time with a 2-minute notice. This makes Spot Instances a great way to save money on any workload that is non-urgent (e.g., all background jobs, machine learning, image processing) and pre-production environments (e.g., run an ECS cluster on spot instances by just setting a single extra param!).

Use Reserved EC2 Instances for dedicated work

EC2 Reserved Instances allow you to reserve capacity ahead of time in exchange for a significant discount (up to 75%) over on-demand pricing. This makes Reserved Instances a great way to save money when you know for sure that you are going to be using a certain number of Instances consistently for a long time period. For example, if you knew you were going to run a 3-node ZooKeeper cluster all year long, you could reserve three `r4.large` Instances for one year, at a discount of 75%. Reserved Instances are a billing optimization, so no code changes are required: just reserve the Instance Type, and next time you use it, AWS will charge you less for it.

Shut down EC2 Instances and RDS DBs when not using them

You can shut down (but not terminate!) EC2 Instances and RDS DBs when you're not using them, such as in your pre-prod environments at night and on weekends. You could even create a Lambda function that does this on a regular schedule. For more info, see AWS Instance Scheduler.

Use Auto Scaling

Use Auto Scaling to increase the number of EC2 Instances when load is high and then to decrease it again—and thereby save money—when load is low.

Use Docker when possible

If you deploy everything as an AMI directly on your EC2 Instances, then you will typically run exactly one type of app per EC2 Instance. If you use a Docker orchestration tool (e.g., ECS), you can give it a cluster of EC2 Instances to manage, and it will deploy Docker containers across the cluster as efficiently as possible, potentially running multiple apps on the same Instances when resources are available.

Use Lambda when possible

For all short (5 min or less) background jobs, cron jobs, ETL jobs, event processing jobs, and other glue code, use AWS Lambda. You not only have no servers to manage, but AWS Lambda pricing is incredibly cheap, with the first 1 million requests and 400,000 GB-seconds per month being completely free! After that, it's just $0.0000002 per request and $0.00001667 for every GB-second.

Clean up old data with S3 Lifecycle settings

If you have a lot of data in S3, make sure to take advantage of S3 Object Lifecycle Management to save money. You can configure the S3 bucket to move files older than a certain age either to cheaper storage classes or to delete those files entirely.

Clean up unused resources

Use tools such as cloud-nuke and Janitor Monkey to clean up unused AWS resources, such as old EC2 Instances or ELBs that no one is using any more. You can run these tools on a regular schedule by using your CI server or scheduled lambda functions.

Learn to analyze your AWS bill

Learn to use tools such as Cost and Usage Report, Cost Explorer, Ice, Trusted Advisor, and Cost Optimization Monitor to understand where you're spending money. Make sure you understand what each category means (e.g., the delightfully vague "EC2 Other" often means EBS Volumes, AMIs, and Load Balancers). If you find something you can't explain, reach out to AWS Support, and they will help you track it down. Using multiple AWS accounts with AWS Organizations and consolidated billing can make it easier to isolate certain types

of costs from others (e.g., break down costs by environment or team).

Create billing alarms

Create billing alerts to notify you when your AWS bill crosses important thresholds. Make sure to have several levels of alerts: e.g., at the very least, one when the bill is a little high, one when it's really high, and one when it is approaching bankruptcy levels.