**AWS**

1. **ELASTIC SEARCH:**

Amazon Elasticsearch Service makes it easy to deploy, operate, and scale Elasticsearch for log analytics, full text search, application monitoring. Amazon Elasticsearch Service is a fully managed service that delivers Elasticsearch’s easy-to-use APIs and real-time capabilities along with the availability, scalability, and security required by production workloads. The service offers built-in integrations with [Kibana](https://aws.amazon.com/elasticsearch-service/kibana/), [Logstash](https://aws.amazon.com/elasticsearch-service/logstash/), and AWS services including [Amazon Kinesis Firehose](https://aws.amazon.com/kinesis/firehose/), [AWS Lambda](https://aws.amazon.com/lambda/), and [Amazon CloudWatch](https://aws.amazon.com/cloudwatch/) so that you can go from raw data to actionable insights quickly.

It’s easy to get started with Amazon Elasticsearch Service. You can set up and configure your Amazon Elasticsearch Service domain in minutes from the AWS Management Console. Amazon Elasticsearch Service provisions all the resources for your domain and launches it. The service automatically detects and replaces failed Elasticsearch nodes, reducing the overhead associated with self-managed infrastructure and Elasticsearch software. Amazon Elasticsearch Service allows you to easily scale your cluster via a single API call or a few clicks in the console. With Amazon Elasticsearch Service, you get direct access to the Elasticsearch open-source API so that code and applications you’re already using with your existing Elasticsearch environments will work seamlessly.

1. **CLOUD FORMATION:**

AWS CloudFormation gives developers and systems administrators an easy way to create and manage a collection of related AWS resources, provisioning and updating them in an orderly and predictable fashion.

You can use AWS CloudFormation’s [sample templates](https://aws.amazon.com/cloudformation/aws-cloudformation-templates/) or create your own templates to describe the AWS resources, and any associated dependencies or runtime parameters, required to run your application. You don’t need to figure out the order for provisioning AWS services or the subtleties of making those dependencies work. CloudFormation takes care of this for you. After the AWS resources are deployed, you can modify and update them in a controlled and predictable way, in effect applying version control to your AWS infrastructure the same way you do with your software. You can also visualize your templates as diagrams and edit them using a drag-and-drop interface with the [AWS CloudFormation Designer](https://aws.amazon.com/cloudformation/details/#designer).

You can deploy and update a template and its associated collection of resources (called a stack) by using the AWS Management Console, AWS Command Line Interface, or APIs. CloudFormation is available at no additional charge, and you pay only for the AWS resources needed to run your applications.

1. **Terraform:**

Terraform is used for building, changing, and versioning infrastructure safely and efficiently. Terraform can manage existing and popular service providers as well as custom in-house solutions.

Configuration files describe to Terraform the components needed to run a single application or your entire datacenter. Terraform generates an execution plan describing what it will do to reach the desired state, and then executes it to build the described infrastructure. As the configuration changes, Terraform is able to determine what changed and create incremental execution plans which can be applied.

The infrastructure Terraform can manage includes low-level components such as compute instances, storage, and networking, as well as high-level components such as DNS entries, SaaS features, etc.

**key features:**

**Infrastructure as Code**

Infrastructure is described using a high-level configuration syntax. This allows a blueprint of your datacenter to be versioned and treated as you would any other code. Additionally, infrastructure can be shared and re-used.

**Execution Plans**

Terraform has a "planning" step where it generates an execution plan. The execution plan shows what Terraform will do when you call apply. This lets you avoid any surprises when Terraform manipulates infrastructure.

**Resource Graph**

Terraform builds a graph of all your resources, and parallelizes the creation and modification of any non-dependent resources. Because of this, Terraform builds infrastructure as efficiently as possible, and operators get insight into dependencies in their infrastructure.

**Change Automation**

Complex change sets can be applied to your infrastructure with minimal human interaction. With the previously mentioned execution plan and resource graph, you know exactly what Terraform will change and in what order, avoiding many possible human errors.

1. **AWS CLI/API:**

AWS Command Line Interface (CLI) is a unified tool to manage your AWS services. you can control multiple AWS services from the command line and automate them through scripts.

* you can view the contents of your S3 buckets in a directory-based listing.

$ aws s3 ls s3://mybucket

* You can perform recursive uploads and downloads of multiple files in a single folder-level command. The AWS CLI will run these transfers in parallel for increased performance.

$ aws s3 cp myfolder s3://mybucket/myfolder --recursive

* A sync command makes it easy to synchronize the contents of a local folder with a copy in an S3 bucket.

$ aws s3 sync myfolder s3://mybucket/myfolder --exclude \*.tmp

$ aws ec2 describe-instances

$ aws ec2 start-instances --instance-ids i-1348636c

$ aws sns publish --topic-arn arn:aws:sns:us-east-1:546419318123:OperationsError

--message "Script Failure"

$ aws sqs receive-message --queue-url

https://queue.amazonaws.com/546419318123/Test

1. **LAMBDA:**

AWS Lambda lets you run code without provisioning or managing servers. You pay only for the compute time you consume - there is no charge when your code is not running. With Lambda, you can run code for virtually any type of application or backend service - all with zero administration. Just upload your code and Lambda takes care of everything required to run and scale your code with high availability. You can set up your code to automatically trigger from other AWS services or call it directly from any web or mobile app.

1. How Lambda works?

* Upload your code to AWS lambda
* Set up your code to trigger from other AWS services, HTTP endpoints, or in-app activity.
* Lambda runs your code only when triggered, using only the compute resources needed. Pay just for the compute time you use.

b) What can you build with AWS Lambda?

* **Data processing**: You can use AWS Lambda to execute code in response to triggers such as changes in data, shifts in system state, or actions by users. Lambda can be directly triggered by AWS services such as S3, DynamoDB, Kinesis, SNS, and CloudWatch, allowing you to build a variety of real-time [serverless](https://aws.amazon.com/serverless/) data processing systems. You can process more complex jobs by coordinating multiple Lambda functions with [AWS Step Functions](https://aws.amazon.com/step-functions/).
* **Real-time file processing**: You can use Amazon S3 to trigger AWS Lambda to process data immediately after an upload. For example, you can use Lambda to thumbnail images like whenever you upload the images to s3, lambda runs image resizing code to generate web, mobile and tablet sizes, transcode videos, index files, process logs, validate content, and aggregate and filter data in real-time.
* **Real-time stream processing:** you can use AWS Lambda and **Amazon Kinesis** to process real-time streaming data like social media stream is loaded into Kinesis in real-time, whenever the data is loaded Lambda will be triggered and Lambda runs code that generate hashtag trend and stores it in DyanamoDB, after this action social media trend data immediately available for business users to query or for application activity tracking, transaction order processing, click stream analysis, data cleansing, metrics generation, log filtering, indexing, social media analysis, and IoT device data telemetry and metering.
* **Extract, Transform, Load:** You can use AWS Lambda to perform data validation, filtering, sorting, or other transformations for every data change in a DynamoDB table and load the transformed data to another data store. Ex: Retail data warehouse ETL like whenever your order is placed, order is stored in operational db like dynamoDB, it triggers the Lambda, It runs data transformation code and loads into data warehouse thenafter analytics generated from data.
* **Web-applications: B**y combining AWS Lambda with other AWS services, developers can build powerful web applications that automatically scale up and down and run in a highly available configuration across multiple data centers – with zero administrative effort required for scalability, back-ups or multi-data center redundancy. Ex: weather app: front-end code for weather app hosted in s3, user clicks to get local weather info, app makes REST API call to endpoint through API GATEWAY, here Lambda is triggered and Lambda runs code to retrieve local weather info and returns data back to user.

1. **AWS KINESIS:**

Amazon Kinesis makes it easy to collect, process, and analyze [real-time, streaming data](https://aws.amazon.com/streaming-data/) so you can get timely insights and react quickly to new information. Amazon Kinesis offers [key capabilities](https://aws.amazon.com/kinesis/#kinesis-capabilities) to cost effectively process streaming data at any scale, along with the flexibility to choose the tools that best suit the requirements of your application. With Amazon Kinesis, you can ingest real-time data such as application logs, website clickstreams, IoT telemetry data, and more into your databases, data lakes and data warehouses, or build your own real-time applications using this data. Amazon Kinesis enables you to process and analyze data as it arrives and respond in real-time instead of having to wait until all your data is collected before the processing can begin.

**Kinesis capabilities:**

### [Amazon Kinesis Firehose](https://aws.amazon.com/kinesis/firehose/): Easily load streaming data into AWS

### [Amazon Kinesis Analytics](https://aws.amazon.com/kinesis/analytics/): Easily process and analyze streaming data with standard SQL

### [Amazon Kinesis Streams](https://aws.amazon.com/kinesis/streams/): Build custom applications that process and analyze streaming data.

1. **ELASTIC BEANSTALK:**

AWS Elastic Beanstalk is an easy-to-use service for deploying and scaling web applications and services developed with Java, [.NET](https://aws.amazon.com/net/), PHP, Node.js, Python, Ruby, Go, and [Docker](https://aws.amazon.com/docker/) on familiar servers such as Apache, Nginx, Passenger, and [IIS](https://aws.amazon.com/windows/).

You can simply upload your code and Elastic Beanstalk automatically handles the deployment, from capacity provisioning, load balancing, auto-scaling to application health monitoring. At the same time, you retain full control over the AWS resources powering your application and can access the underlying resources at any time.

There is no additional charge for Elastic Beanstalk - you pay only for the AWS resources needed to store and run your applications.

1. **AWS EC2 CONTAINER SERVICE:**

Amazon EC2 Container Service (ECS) is a highly scalable, high performance [container](https://aws.amazon.com/containers/) management service that supports [Docker](https://aws.amazon.com/docker/) containers and allows you to easily run applications on a managed cluster of Amazon EC2 instances. Amazon ECS eliminates the need for you to install, operate, and scale your own cluster management infrastructure. With simple API calls, you can launch and stop Docker-enabled applications, query the complete state of your cluster, and access many familiar features like security groups, Elastic Load Balancing, EBS volumes, and IAM roles. You can use Amazon ECS to schedule the placement of containers across your cluster based on your resource needs and availability requirements. You can also integrate your own scheduler or third-party schedulers to meet business or application specific requirements.

There is no additional charge for Amazon EC2 Container Service. You pay for AWS resources (e.g. EC2 instances or EBS volumes) you create to store and run your application.

1. **Migration apps to aws:**

Phases Benefits:

a)Cloud Assessment

• Financial Assessment (TCO calculation) • Security and Compliance Assessment • Technical Assessment (Classify application types) • Identify the tools that can be reused and the tools that need to be built • Migrate licensed products • Create a plan and measure success Business case for migration (Lower TCO, faster time to market, higher flexibility & agility, scalability + elasticity) Identify gaps between your current traditional legacy architecture and next -generation cloud architecture

b)Proof of Concept

• Get your feet wet with AWS • Build a pilot and validate the technology • Test existing software in the cloud Build confidence with various AWS services Mitigate risk by validating critical pieces of your proposed architecture

c)Moving your Data

• Understand different storage options in the AWS cloud • Migrate fileservers to Amazon S3 • Migrate commercial RDBMS to EC2 + EBS • Migrate MySQL to Amazon RDS Redundancy, Durable Storage, Elastic Scalable Storage Automated Management Backup

d)Moving your Apps

• Forklift migration strategy • Hybrid migration strategy • Build “cloud-aware” layers of code as needed • Create AMIs for each component Future-proof scaled-out service-oriented elastic architecture

e)Leveraging the Cloud

• Leverage other AWS services • Automate elasticity and SDLC • Harden security • Create dashboard to manage AWS resources • Leverage multiple availability zones Reduction in CapEx in IT Flexibility and agility Automation and improved productivity Higher Availability (HA)

f)Optimization

• Optimize usage based on demand • Improve efficiency • Implement advanced monitoring and telemetry • Re-engineer your application • Decompose your relational databases Increased utilization and transformational impact in OpEx Better visibility through advanced monitoring and telemetry

1. **KUBERNETES:**

[Kubernetes](https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/) is an open-source system for automating deployment, scaling, and management of containerized applications.

It groups containers that make up an application into logical units for easy management and discovery. Kubernetes builds upon [15 years of experience of running production workloads at Google](http://queue.acm.org/detail.cfm?id=2898444), combined with best-of-breed ideas and practices from the community.

* Kubernetes can scale without increasing your ops team.
* Whether testing locally or running a global enterprise, Kubernetes flexibility grows with you to deliver your applications consistently and easily no matter how complex your need is.
* Kubernetes is open source giving you the freedom to take advantage of on-premises, hybrid, or public cloud infrastructure, letting you effortlessly move workloads to where it matters to you.

### Kubernetes Features:

* **Automatic binpacking**: Automatically places containers based on their resource requirements and other constraints, while not sacrificing availability. Mix critical and best-effort workloads in order to drive up utilization and save even more resources.

#### [**Self-healing**](https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/#what-is-a-replicationcontroller)**:** Restarts containers that fail, replaces and reschedules containers when nodes die, kills containers that don't respond to your user-defined health check, and doesn't advertise them to clients until they are ready to serve.

#### [**Horizontal scaling**](https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/)**:** Scale your application up and down with a simple command, with a UI, or automatically based on CPU usage.

#### [**Service discovery and load balancing**](https://kubernetes.io/docs/concepts/services-networking/service/)**:** No need to modify your application to use an unfamiliar service discovery mechanism. Kubernetes gives containers their own IP addresses and a single DNS name for a set of containers, and can load-balance across them.

#### [**Automated rollouts and rollbacks**](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#what-is-a-deployment)**:** Kubernetes progressively rolls out changes to your application or its configuration, while monitoring application health to ensure it doesn't kill all your instances at the same time. If something goes wrong, Kubernetes will rollback the change for you. Take advantage of a growing ecosystem of deployment solutions.

#### [**Secret**](https://kubernetes.io/docs/concepts/configuration/secret/)**and**[**configuration**](https://kubernetes.io/docs/tasks/configure-pod-container/configmap/)**management:** Deploy and update secrets and application configuration without rebuilding your image and without exposing secrets in your stack configuration.

#### [**Storage orchestration**](https://kubernetes.io/docs/concepts/storage/persistent-volumes/)**:** Automatically mount the storage system of your choice, whether from local storage, a public cloud provider such as [GCP](https://cloud.google.com/storage/) or [AWS](https://aws.amazon.com/products/storage/), or a network storage system such as NFS, iSCSI, Gluster, Ceph, Cinder, or Flocker.

#### [**Batch execution**](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/)**:** In addition to services, Kubernetes can manage your batch and CI workloads, replacing containers that fail, if desired.

1. What is SSL certificate ?

When it is installed on a web server it activates the padlock and the https protocol and then make a secure connection from the web server to the browser. Mostly SSL is used to secure the credit card transactions. Usually SSL binds the domain name, server name, company’s name together. Every organization must install the SSL on its web server to securely connect to the browser.

1. How to create a SSL ?
2. Generate a private key- openssl tool kit is used to generate private key and CSR. This private key is 1024 bit key and is stored in pem format.

* **openssl genrsa -des3 -out server.key 1024**

1. Generate a CSR- Generally this CSR is sent to Certificate Authority, who will verify the identity of the requestor and issues a certificate.

* **openssl req -new -key server.key -out server.csr**

1. Remove passphrase from key- Important reason for the removal of passphrase is APACHE will ask for the passphrase every time you start the webserver.

* **cp server.key server.key.org  
  openssl rsa -in server.key.org -out server.key**

1. Generating a self signed certificate- The below command creates a SSL certificate which is temporary and good for 365 days

* **openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt**

1. Installing the private key and certificate-

* **cp server.crt /usr/local/apache/conf/ssl.crt  
  cp server.key /usr/local/apache/conf/ssl.key**

1. Configuring SSL enabled virtual hosts
2. Restart apache and Test

### pplication Development

**Store no application state on your servers.**  
The reason for this is so that if you server gets killed, you won't lose any application state. To that end, sessions should be stored in a database (or some other sort of central storage; memcached, redis, etc.), not on the local filesystem. Logs should be handled via syslog (or similar) and sent to a remote store. Uploads should go direct to S3 (don't store on local filesystem and have another process move to S3 for example). And any post-processing or long running tasks should be done via an asynchronous queue (SQS is great for this).  
  
**Edit:** For S3 uploads, HN user [*krallin pointed out*](https://news.ycombinator.com/item?id=7172583) that you can bypass your server entirely and use [*pre-signed URLs*](http://docs.aws.amazon.com/AmazonS3/latest/dev/PresignedUrlUploadObject.html) to let your users upload directly to S3.  
  
**Store extra information in your logs.**  
Log lines normally have information like timestamp, pid, etc. You'll also probably want to add instance-id, region, availability-zone and environment (staging, production, etc), as these will help debugging considerably. You can get this information from the [instance metadata service](http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AESDG-chapter-instancedata.html). The method I use is to grab this information as part of my bootstrap scripts, and store it in files on the filesystem (/env/az, /env/region, etc). This way I'm not constantly querying the metadata service for the information. You should make sure this information gets updated properly when your instances reboot, as you don't want to save an AMI and have the same data persist, as it will then be incorrect.  
  
**If you need to interact with AWS, use the SDK for your langauge.**  
Don't try to roll your own, I did this at first as I only needed a simple upload to S3, but then you add more services and it's just an all around bad idea. [The AWS SDKs](http://aws.amazon.com/tools/) are well written, handle authentication automatically, handle retry logic, and they're maintained and iterated on by Amazon. Also, if you use EC2 IAM roles (which you absolutely should, more on this later) then the SDK will automatically grab the correct credentials for you.  
  
**Have tools to view application logs.**  
You should have an admin tool, syslog viewer, or something that allows you to view current real-time log info without needing to SSH into a running instance. If you have centralised logging (which you really should), then you just want to be sure you can read the logs there without needing to use SSH. Needing to SSH into a running application instance to view logs is going to become problematic.

### Operations

*If you have to SSH into your servers, then your automation has failed.*

**Disable SSH access to all servers.**  
This sounds crazy, I know, but port 22 should be disallowed for everyone in your security group. If there's one thing you take away from this post, this should be it: **If you have to SSH into your servers, then your automation has failed**. Disabling it at the firewall level (rather than on the servers themselves) will help the transition to this frame of thinking, as it will highlight any areas you need to automate, while still letting you easily re-instate access to solve immediate issues. It's incredibly freeing to know that you never need to SSH into an instance. This is both the most frightening and yet most useful thing I've learned.  
  
**Edit:** A lot of people are concerned about this particular tip (there's some [*good discussion over on Hacker News*](https://news.ycombinator.com/item?id=7173361)), so I'd like to expand on it a little. Disabling inbound SSH has just been a way for me to stop myself cheating with automation (Oh, I'll just SSH in and fix this one thing). I can still re-enable it in the security group if I need to actively debug something on an instance, since sometimes there really is no other way to debug certain issues. It also depends on your application; If your application relies on you being able to push things to a server via SSH, then disabling it might be a bad idea. Blocking inbound SSH worked for me, and forced me to get my automation into a decent state, but it might not be for everyone.  
  
**Servers are ephemeral, you don't care about them. You only care about the service as a whole.**  
If a single server dies, it should be of no big concern to you. This is where the real benefit of AWS comes in compared to using physical servers yourself. Normally if a physical server dies, there's panic. With AWS, you don't care, because auto-scaling will give you a fresh new instance soon anyway. Netflix have taken this several steps further with their [simian army](http://techblog.netflix.com/2011/07/netflix-simian-army.html), where they have things like [Chaos Monkey](http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html), which will kill random instances in production (they also have Chaos Gorilla to kill AZs and I've heard rumour of a Chaos Kong to kill regions...). The point is that servers will fail, but this shouldn't matter in your application.  
  
**Don't give servers static/elastic IPs.**  
For a typical web application, you should put things behind a load balancer, and balance them between AZs. There are a few cases where Elastic IPs will probably need to be used, but in order to make best use of auto-scaling you'll want to use a load balancer instad of giving every instance their own unique IP.  
  
**Automate everything.**  
This is more of general operations advice than AWS specific, but everything needs to be automated. Recovery, deployment, failover, etc. Package and OS updates should be managed by something, whether it's just a bash script, or Chef/Puppet, etc. You shouldn't have to care about this stuff. As mentioned earlier, you should also make sure to disable SSH access, as this will pretty quickly highlight any part of your process that isn't automated. Remember the key phrase from earlier, if you have to SSH into your servers, then your automation has failed.  
  
**Everyone gets an IAM account. Never login to the master.**  
Usually you'll have an "operations account" for a service, and your entire ops team will have the password. With AWS, you definitely don't want to do that. Everyone gets an IAM user with just the permissions they need (least privilege). An IAM user can control everything in the infrastructure. At the time of writing, the only thing an IAM user can't access are some parts of the billing pages.  
  
If you want to protect your account even more, make sure to [enable multi-factor authentication](http://aws.amazon.com/iam/details/mfa/) for everyone (you can use Google Authenticator). I've heard of some users who give the MFA token to two people, and the password to two others, so to perform any action on the master account, two of the users need to agree. This is overkill for my case, but worth mentioning in case someone else wants to do it.

*The last time I had an actionable alert from CloudWatch was about a year ago...*

**Get your alerts to become notifications.**  
If you've set everyting up correctly, your health checks should automatically destroy bad instances and spawn new ones. There's usually no action to take when getting a CloudWatch alert, as everything should be automated. If you're getting alerts where manual intervention is required, do a post-mortem and figure out if there's a way you can automate the action in future. The last time I had an actionable alert from CloudWatch was about a year ago, and it's extremely awesome not to be woken up at 4am for ops alerts any more.

### Billing

**Set up granular billing alerts.**  
You should always have at least one billing alert set up, but that will only tell you on a monthly basis once you've exceeded your allowance. If you want to catch runaway billing early, you need a more fine grained approach. The way I do it is to set up an alert for my expected usage each week. So the first week's alert for say $1,000, the second for $2,000, third for $3,000, etc. If the week-2 alarm goes off before the 14th/15th of the month, then I know something is probably going wrong. For even more fine-grained control, you can set this up for each individual service, that way you instantly know which service is causing the problem. This could be useful if your usage on one service is quite steady month-to-month, but another is more erratic. Have the indidividual weekly alerts for the steady one, but just an overall one for the more erratic one. If everything is steady, then this is probably overkill, as looking at CloudWatch will quickly tell you which service is the one causing the problem.

### Security

**Use EC2 roles, do not give applications an IAM account.**  
If your application has AWS credentials baked into it, you're "doing it wrong". One of the reasons it's important to use the AWS SDK for your language is that you can really easily use EC2 IAM roles. The idea of a role is that you specify the permissions a certain role should get, then assign that role to an EC2 instance. Whenever you use the AWS SDK on that instance, you don't specify any credentials. Instead, the SDK will retrieve temporary credentials which have the permissions of the role you set up. This is all handled transparently as far as you're concerned. It's secure, and extremely useful.  
  
**Assign permissions to groups, not users.**  
Managing users can be a pain, if you're using Active Directory, or some other external authentication mechanism which you've integrated with IAM, then this probably won't matter as much (or maybe it matters more). But I've found it much easier to manage permissions by assigning them only to groups, rather than to individual users. It's much easier to rein in permissions and get an overall view of the system than going through each individual user to see what permissions have been assigned.  
  
**Set up automated security auditing.**  
It's important to keep track of changes in your infrastructure's security settings. One way to do this is to first set up a security auditer role ([JSON template](https://s3.amazonaws.com/reinvent2013-sec402/secaudit.json)), which will give anyone assigned that role read-only access to any security related settings on your account. You can then use this rather [fantastic Python script](https://s3.amazonaws.com/reinvent2013-sec402/SecConfig.py), which will go over all the items in your account and produce a canonical output showing your configuration. You set up a cronjob somewhere to run this script, and compare its output to the output from the previous run. Any differences will show you exactly what has been changed in your security configuration. It's useful to set this up and just have it email you the diff of any changes. (Source: Intrusion Detection in the Cloud - [Video](https://www.youtube.com/user/AmazonWebServices/Cloud?x=us-en_reinvent_1878_35) & [Presentation](http://awsmedia.s3.amazonaws.com/SEC402.pdf))  
  
**Use CloudTrail to keep an audit log.**  
CloudTrail will log any action performed via the APIs or web console into an S3 bucket. Set up the bucket with versioning to be sure no one can modify your logs, and you then have a complete audit trail of all changes in your account. You hope that you will never need to use this, but it's well worth having for when you do.

### S3

**Use "-" instead of "." in bucket names for SSL.**  
If you ever want to use your bucket over SSL, using a "." will cause you to get certificate mismatch errors. You can't change bucket names once you've created them, so you'd have to copy everything to a new bucket.

*I've found them to be about as reliable as a large government department...*

**Avoid filesystem mounts (FUSE, etc).**  
I've found them to be about as reliable as a large government department when used in critical applications. Use the SDK instead.  
  
**You don't have to use CloudFront in front of S3 (but it can help).**  
**Edit:** Based on some [*excellent*](https://news.ycombinator.com/item?id=7172936) [*feedback*](https://news.ycombinator.com/item?id=7176110) from Hacker News users, I've made some modifications to this tip.  
If all you care about is scalability, you can link people directly to the S3 URL instead of using CloudFront. S3 can scale to any capacity (although [some users](https://news.ycombinator.com/item?id=7173464) have reported that it doesn't scale instantly), so is great if that's all your care about. Additionally, updates are available quickly in S3, yet you have to wait for the TTL when using a CDN to see the change (although I believe you can set a 0s TTL in CloudFront now, so this point is probably moot).  
  
If you need speed, or are handling very high bandwidth (10TB+), then you might want to use a CDN like CloudFront in front of S3. CloudFront can dramatically [speed up access](http://www.quora.com/What-are-typical-latencies-for-static-content-in-S3-vs-Cloudfront) for users around the globe, as it copies your content to edge locations. Depending on your use case, this can also work out slightly cheaper if you deal with very high bandwidth (10TB+) with lower request numbers, as it's about $0.010/GB cheaper for CloudFront bandwidth than S3 bandwidth once you get above 10TB, but the cost per request is slightly higher than if you were to access the files from S3 directly. Depending on your usage pattern, the savings from bandwidth could outweigh the extra cost per request. Since content is only fetched from S3 infrequently (and at a much lower rate than normal), your S3 cost would be much smaller than if you were serving content directly from S3. The [AWS documentation on CloudFront](http://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/MigrateS3ToCloudFront.html) explains how you can use it with S3.  
  
**Use random strings at the start of your keys.**  
This seems like a strange idea, but one of the implementation details of S3 is that Amazon use the object key to determine where a file is physically placed in S3. So files with the same prefix might end up on the same hard disk for example. By randomising your key prefixes, you end up with a better distribution of your object files. (Source: [S3 Performance Tips & Tricks](http://aws.typepad.com/aws/2012/03/amazon-s3-performance-tips-tricks-seattle-hiring-event.html))

### EC2/VPC

**Use tags!**  
Pretty much everything can be given tags, use them! They're great for organising things, make it easier to search and group things up. You can also use them to trigger certain behaviour on your instances, for example a tag of env=debug could put your application into debug mode when it deploys, etc.

*I've had it happen, it sucks, learn from my mistake!*

**Use termination protection for non-auto-scaling instances. Thank me later.**  
If you have any instances which are one-off things that aren't under auto-scaling, then you should probably enable termination protection, to stop anyone from accidentally deleting the instance. I've had it happen, it sucks, learn from my mistake!  
  
**Use a VPC.**  
VPC either wasn't around, or I didn't notice it when I got started with AWS. It seems like a pain at first, but once you get stuck in and play with it, it's suprising easy to set up and get going. It provides all sorts of extra features over EC2 that are well worth the extra time it takes to set up a VPC. First, you can control traffic at the network level using ACLs, you can modify instance size, security groups, etc. without needing to terminate an instance. You can specify egress firewall rules (you cannot control outbound traffic from normal EC2). But the biggest thing is that you have your own private subnet where your instances are completely cut off from everyone else, so it adds an extra layer of protection. Don't wait like I did, use VPC straight away to make things easy on yourself.  
  
If you're interested in the internals of VPC, I highly recommend watching [A Day in the Life of Billion Packets](http://www.youtube.com/watch?v=Zd5hsL-JNY4) ([Slides](http://www.slideshare.net/AmazonWebServices/a-day-in-the-life-of-a-billion-packets-cpn401-aws-reinvent-2013)).  
  
**Use reserved instances to save big $$$.**  
Reserving an instance is just putting some money upfront in order to get a lower hourly rate. It ends up being a lot cheaper than an on-demand instance would cost. So if you know you're going to be keeping an instance around for 1 or 3 years, it's well worth reserving them. Reserved instances are a purely logical concept in AWS, you don't assign a specific instance to be reserved, but rather just specify the type and size, and any instances that match the criteria will get the lower price.  
  
**Lock down your security groups.**  
Don't use 0.0.0.0/0 if you can help it, make sure to use specific rules to restrict access to your instances. For example, if your instances are behind an ELB, you should set your security groups to only allow traffic from the ELBs, rather than from 0.0.0.0/0. You can do that by entering "amazon-elb/amazon-elb-sg" as the CIDR (it should auto-complete for you). If you need to allow some of your other instances access to certain ports, don't use their IP, but specify their security group identifier instead (just start typing "sg-" and it should auto-complete for you).  
  
**Don't keep unassociated Elastic IPs.**  
You get charged for any Elastic IPs you have created but not associated with an instance, so make sure you don't keep them around once you're done with them.

### ELB

**Terminate SSL on the load balancer.**  
You'll need to add your SSL certificate information to the ELB, but this will take the overhead of SSL termination away from your servers which can speed things up. Additionally, if you upload your SSL certificate, you can pass through the HTTPS traffic and the load balancer will add some extra headers to your request (x-forwarded-for, etc), which are useful if you want to know who the end user is. If you just forward TCP, then those headers aren't added and you lose the information.  
  
**Pre-warm your ELBs if you're expecting heavy traffic.**  
It takes time for your ELB to scale up capacity. If you know you're going to have a large traffic spike (selling tickets, big event, etc), you need to "warm up" your ELB in advance. You can inject a load of traffic, and it will cause ELB to scale up and not choke when you actually get the traffic, however AWS suggest you contact them instead to prewarm your load balancer. (Source: [Best Practices in Evaluating Elastic Load Balancing](http://aws.amazon.com/articles/1636185810492479#pre-warming)). Alternatively you can install your own load balancer software on an EC2 instance and use that instead (HAProxy, etc).

### ElastiCache

**Use the configuration endpoints, instead of individual node endpoints.**  
Normally you would have to make your application aware of every Memcached node available. If you want to dynamically scale up your capacity, then this becomes an issue as you will need to have some way to make your application aware of the changes. An easier way is to use the configuration endpoint, which means using an AWS version of a Memcached library that abstracts away the auto-discovery of new nodes. The [AWS guide to cache node auto-discovery](http://docs.aws.amazon.com/AmazonElastiCache/latest/UserGuide/AutoDiscovery.html) has more information.

### RDS

**Set up event subscriptions for failover.**  
If you're using a Multi-AZ setup, this is one of those things you might not think about which ends up being incredibly useful when you do need it.

### CloudWatch

**Use the CLI tools.**  
It can become extremely tedious to create alarms using the web console, especially if you're setting up a lot of similar alarms, as there's no ability to "clone" an existing alarm while making a minor change elsewhere. Scripting this using the CLI tools can save you lots of time.  
  
**Use the free metrics.**  
CloudWatch monitors all sorts of things for free (bandwidth, CPU usage, etc.), and you get up to 2 weeks of historical data. This saves you having to use your own tools to monitor you systems. If you need longer than 2 weeks, unfortunately you'll need to use a third-party or custom built monitoring solution.  
  
**Use custom metrics.**  
If you want to monitor things not covered by the free metrics, you can send your own metric information to CloudWatch and make use of the alarms and graphing features. This can not only be used for things like tracking diskspace usage, but also for custom application metrics too. The AWS page on [publishing custom metrics](http://docs.aws.amazon.com/AmazonCloudWatch/latest/DeveloperGuide/publishingMetrics.html) has more information.  
  
**Use detailed monitoring.**  
It's ~$3.50 per instance/month, and well worth the extra cost for the extra detail. 1 minute granularity is much better than 5 minute. You can have cases where a problem is hidden in the 5 minute breakdown, but shows itself quite clearly in the 1 minute graphs. This may not be useful for everyone, but it's made investigating some issues much easier for me.

### Auto-Scaling

**Scale down on INSUFFICIENT\_DATA as well as ALARM.**  
For your scale-down action, make sure to trigger a scale-down event when there's no metric data, as well as when your trigger goes off. For example, if you have an app which usually has very low traffic, but experiences occasional spikes, you want to be sure that it scales down once the spike is over and the traffic stops. If there's no traffic, you'll get INSUFFIFIENT\_DATA instead of ALARM for your low traffic threshold and it won't trigger a scale-down action.  
  
**Use ELB health check instead of EC2 health checks.**  
This is a configuration option when creating your scaling group, you can specify whether to use the standard EC2 checks (is the instance connected to the network), or to use your ELB health check. The ELB health check offers way more flexibility. If your health check fails and the instance gets taken out of the load balancing pool, you're pretty much always going to want to have that instance killed by auto-scaling and a fresh one take it's place. If you don't set up your scaling group to use the ELB checks, then that won't necessarily happen. The [AWS documentation on adding the health check](http://docs.aws.amazon.com/AutoScaling/latest/DeveloperGuide/as-add-elb-healthcheck.html) has all the information you need to set this up.  
  
**Only use the availability zones (AZs) your ELB is configured for.**  
If you add your scaling group to multiple AZs, make sure your ELB is configured to use all of those AZs, otherwise your capacity will scale up, and the load balancer won't be able to see them.  
  
**Don't use multiple scaling triggers on the same group.**  
If you have multiple CloudWatch alarms which trigger scaling actions for the same auto-scaling group, it might not work as you initially expect it to. For example, let's say you add a trigger to scale up when CPU usage gets too high, or when the inbound network traffic gets high, and your scale down actions are the opposite. You might get an increase in CPU usage, but your inbound network is fine. So the high CPU trigger causes a scale-up action, but the low inbound traffic alarm immediately triggers a scale-down action. Depending on how you've set your cooldown period, this can cause quite a problem as they'll just fight against each other. If you want multiple triggers, you can use multiple auto-scaling groups.

### IAM

**Use IAM roles.**  
Don't create users for application, always use IAM roles if you can. They simplify everything, and keeps things secure. Having application users just creates a point of failure (what if someone accidentally deletes the API key?) and it becomes a pain to manage.  
  
**Users can have multiple API keys.**  
This can be useful if someone is working on multiple projects, or if you want a one-time key just to test something out, without wanting to worry about accidentally revealing your normal key.  
  
**IAM users can have multi-factor authentication, use it!**  
[Enable MFA](http://aws.amazon.com/iam/details/mfa/) for your IAM users to add an extra layer of security. Your master account should most definitely have this, but it's also worth enabling it for normal IAM users too.

### Route53

**Use ALIAS records.**  
An ALIAS record will link your record set to a particular AWS resource directly (i.e. you can map a domain to an S3 bucket), but the key is that you don't get charged for any ALIAS lookups. So whereas a CNAME entry would cost you money, an ALIAS record won't. Also, unlike a CNAME, you can use an ALIAS on your zone apex. You can read more about this on [the AWS page for creating alias resource record sets](http://docs.aws.amazon.com/Route53/latest/DeveloperGuide/CreatingAliasRRSets.html).

### Elastic MapReduce

**Specify a directory on S3 for Hive results.**  
If you use Hive to output results to S3, you must specify a directory in the bucket, not the root of the bucket, otherwise you'll get a rather unhelpful NullPointerException with no real explanation as to why.

### Miscellaneous Tips

**Scale horizontally.**  
I've found that using lots of smaller machines is generally more reliable than using a smaller number of larger machines. You need to balance this though, as trying to run your application from 100 t1.micro instances probably isn't going to work very well. Breaking your application into lots of smaller instances means you'll be more resiliant to failure in one of the machines. If you're just running from two massive compute cluster machines, and one goes down, things are going to get bad.  
  
**Your application may require changes to work on AWS.**  
While a lot of applications can probably just be deployed to an EC2 instance and work well, if you're coming from a physical environment, you may need to re-architect your application in order to accomodate changes. Don't just think you can copy the files over and be done with it.  
  
**Always be redundant across availability zones (AZs).**  
AZs can have outages, it's happened in the past that certain things in an AZ have gone down. Spreading your application into multiple AZs is as simple as adding a new AZ to your load balancer and starting an instance in that AZ. You should spread your load over two AZs at the very least! If you can afford it, being redundant across regions can also be well worth it, but this generally has a more complex set up cost and isn't always necessary. You can now copy AMIs between regions, and you can set up your Route53 records to balance traffic between regions, but you can't use a single ELB across regions.  
  
**Be aware of AWS service limits before you deploy.**  
Various service limits are enforced which aren't highlighted until you're actually trying to deploy your application and get the error notification. These limits can easily be increased by making a request to AWS support, however that can involve a significant turn around time (as low as a few minutes, up to a few days, based on past experience), during which you won't be able to finish deploying. A few days before deploying, you should consult the [service limits](http://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html) page to see if you think you're going to exceed any of them, and make your support request ahead of time. You will need to make a separate request to each department where you need a limit increased. It's also worth pointing out that some limits are global, while others are per-region.  
  
**Decide on a naming convention early, and stick to it.**  
There's a lot of resources on AWS where you can change the name later, but there's equally a lot where you cannot (security group names, etc). Having a consistent naming convention will help to self-document your infrastructure. Don't forget to make use of tags too.  
  
**Decide on a key-management strategy from the start.**  
Are you going to have one key-pair per group of instances, or are you going to have one key-pair you use for your entire account? It's easy to modify your authorized-keys file with a bootstrap script of course, but you need to decide if you want to manage multiple key-pairs or not, as things will get complicated later on if you try to change your mind.  
  
**Make sure AWS is right for your workload.**  
User [mbreese](https://news.ycombinator.com/user?id=mbreese) on Hacker News makes the very good point that you should make sure that using AWS is correct for your particular workload. If you have a steady load and 24/7 servers, it's possible there are cheaper providers you can use, or it might even be cheaper to use dedicated hardware of your own. One of the big benefits of AWS is the ability to scale up and down rapidly in response to load, but not everyone needs that feature. As when purchasing anything, you should shop around a bit first to make sure you're getting the best deal for what you need.