# Best Practices for the Cloud

Design For Failure & Nothing Will Fail, Decouple your components, Implementing elasticity, Automating Your Infrastructure, Thinking parallel.

In this section, you will learn about designing the best practices that will help you build an application in the cloud.

### Design For Failure & Nothing Will Fail

Rule of thumb: Be a pessimist when designing architectures in the cloud; assume things will fail. In other words, always design, implement and deploy for automated recovery from failure.

In particular, assume that your hardware will fail. Assume that outages will occur. Assume that some disaster will strike your application. Assume that you will be slammed with more than the expected number of requests per second some day.

If you realize that things will fail over time and incorporate that thinking into your architecture, build mechanisms to handle that failure before disaster strikes to deal with a scalable infrastructure, you will end up creating a fault-tolerant architecture that is optimized for the cloud.

### Questions that you need to ask yourself:

What happens if a node in your system fails? How do you recognize that failure? How do I replace that node? What kind of scenarios do I have to plan for? What are my single points of failure? If a load balancer is sitting in front of an array of application servers, what if that load balancer fails? If there are master and slaves in your architecture, what if the master node fails? How does the failover occur and how is a new slave instantiated and brought into sync with the master? Just like designing for hardware failure, you have to also design for software failure.

### Questions that you need to ask:

What happens to my application if the dependent services changes its interface? What if downstream service times out or returns an exception? What if the cache keys grow beyond memory limit of an instance? Build mechanisms to handle that failure. For example, the following strategies can help in event of failure:

1. Have a coherent backup and restore strategy for your data and automate it.
2. Build process threads that resume on reboot.
3. Allow the state of the system to re-sync by reloading messages from queues.
4. Keep pre-configured and pre-optimized virtual images to support on launch/boot.
5. Avoid in-memory sessions or stateful user context, move that to data stores. Good cloud architectures should be impervious to reboots and re-launches. You can do this using a combination of Amazon SQS and Amazon SimpleDB, the overall controller architecture is very resilient to the types of failures listed in this section.

For instance, if the instance on which controller thread was running dies, it can be brought up and resume the previous state as if nothing had happened. This was accomplished by creating a pre-configured Amazon Machine Image, which when launched dequeues all the messages from the Amazon SQS queue and reads their states from an Amazon SimpleDB domain on reboot.

Designing with an assumption that underlying hardware will fail, will prepare you for the future when it actually fails. This design principle will help you design operations-friendly applications.

If you can extend this principle to proactively measure and balance load dynamically, you might be able to deal with variance in network and disk performance that exists due to multi-tenant nature of the cloud.

Tactics for implementing the above best practice:

1. Failover Gracefully Using Elastic IPs: Elastic IP is a static IP that is dynamically re-mappable. You can quickly remap and failover to another

set of servers so that your traffic is routed to the new servers. It works great when you want to upgrade from old to new versions or in case of hardware failures.

2. Utilize Multiple Availability Zones: Availability Zones / Availability Domains are conceptually like logical data centers. By deploying your architecture to multiple availability zones, you can ensure highly availability. Utilize Amazon RDS Multi-AZ deployment functionality to automatically replicate database updates across multiple Availability Zones.
3. Maintain a Machine Image so that you can restore and clone environments very easily in a different Availability Zone; Maintain multiple Database slaves across Availability Zones and setup hot replication.
4. Utilize CloudWatch to get more visibility and take appropriate actions in case of hardware failure or performance degradation. Setup an Auto scaling group to maintain a fixed fleet size so that it replaces unhealthy EC2 instances by new ones.
5. Utilize EBS and set up cron jobs so that incremental snapshots are automatically uploaded to Amazon S3 and data is persisted independent of your instances.
6. Utilize RDS and set the retention period for backups, so that it can perform automated backups.

### Decouple your components

The cloud reinforces the SOA design principle that the more loosely coupled the components of the system, the bigger and better it scales. The key is to build components that do not have tight dependencies on each other, so that if one component were to fail, not respond or slow to respond for some reason, the other components in the system are built to continue to work as if no failure is happening. In essence, loose coupling isolates the various layers and components of your application so that each component interacts asynchronously with the others and treats them as a “black box”.

For example, in the case of web application architecture, you can isolate the

app server from the web server and from the database. The app server does

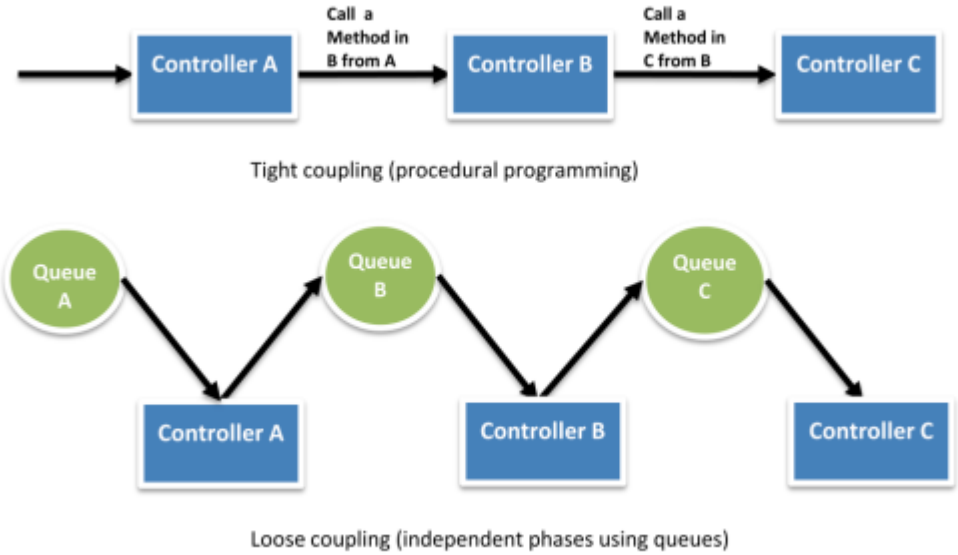
not know about your web server and vice versa, this gives decoupling between these layers and there are no dependencies code-wise or functional perspectives. In the case of batch processing architecture, you can create asynchronous components that are independent of each other.

Questions you need to ask:

Which business component or feature could be isolated from current monolithic application and can run standalone separately? And then how can I add more instances of that component without breaking my current system and at the same time serve more users? How much effort will it take to encapsulate the component so that it can interact with other components asynchronously? Decoupling your components, building asynchronous systems and scaling horizontally become very important in the context of the cloud.

It will not only allow you to scale out by adding more instances of same component but also allow you to design innovative hybrid models in which a few components continue to run in on-premise while other components can take advantage of the cloud-scale and use the cloud for additional compute-power and bandwidth. That way with minimal effort, you can “overflow” excess traffic to the cloud by implementing smart load balancing tactics.

One can build a loosely coupled system using messaging queues. If a queue/buffer is used to connect any two components together, it can support concurrency, high availability and load spikes. As a result, the overall system continues to perform even if parts of components are momentarily unavailable. If one component dies or becomes temporarily unavailable, the system will buffer the messages and get them processed when the component comes back up.




You will see heavy use of queues in in the Cloud Architectures in general. If lots of requests suddenly reach the server (an Internet-induced overload situation) or the processing of regular expressions takes a longer time than the median (slow response rate of a component), the Amazon SQS queues buffer the requests in a durable fashion so that those delays do not affect other components. AWS specific tactics for implementing this best practice:

1. Use SQS to isolate components
2. Use SQS as buffers between components
3. Design every component such that it expose a service interface and is responsible for its own scalability in all appropriate dimensions and interacts with other components asynchronously
4. Bundle the logical construct of a component into a Machine Image so that it can be deployed more often
5. Make your applications as stateless as possible. Store session state outside of component.

### Implement elasticity

The cloud brings a new concept of elasticity in your applications. Elasticity can be implemented in three ways:

1. Proactive Cyclic Scaling: Periodic scaling that occurs at fixed interval (daily, weekly, monthly, quarterly)
2. Proactive Event-based Scaling: Scaling just when you are expecting a big

surge of traffic requests due to a scheduled business event (new product launch, marketing campaigns)

3. Auto-scaling based on demand. By using a monitoring service, your system can send triggers to take appropriate actions so that it scales up or down based on metrics (utilization of the servers or network i/o, for instance) To implement “Elasticity”, one has to first automate the deployment process and streamline the configuration and build process. This will ensure that the system can scale without any human intervention.

This will result in immediate cost benefits as the overall utilization is increased by ensuring your resources are closely aligned with demand rather than potentially running servers that are under-utilized. Automate your infrastructure One of the most important benefits of using a cloud environment is the ability to use the cloud’s APIs to automate your deployment process. It is recommended that you take the time to create an automated deployment process early on during the migration process and not wait till the end. Creating an automated and repeatable deployment process will help reduce errors and facilitate an efficient and scalable update process.

To automate the deployment process:

1. Create a library of “recipes” – small frequently-used scripts (for installation and configuration)
   Manage the configuration and deployment process using agents bundled inside an AMI
2. Bootstrap your instances
3. Bootstrap your instances Let your instances ask you a question at boot “who am I and what is my role?” Every Instance should have a role (“DB server”, “app server”, “slave server” in the case of a Web application) to play in the environment.

This role may be passed in as an argument during launch that instructs the AMI when instantiated the steps to take after it has booted. On boot, instances should grab the necessary resources (code, scripts, configuration) based on the role and “attach” itself to a cluster to serve its function.

Benefits of bootstrapping your instances:

1. Recreate the (Dev, staging, Production) environment with few clicks and minimal effort

2. More control over your abstract cloud-based resources
3. Reduce human-induced deployment errors
4. Create a Self Healing and Self-discoverable environment which is more resilient to hardware failure

### AWS Specific Tactics To Automate Your Infrastructure

1. Define Auto-scaling groups for different clusters using the Amazon Auto-scaling feature in EC2.
2. Monitor your system metrics (CPU, Memory, Disk I/O, Network I/O) using Amazon CloudWatch and take appropriate actions (launching new AMIs dynamically using the Auto-scaling service) or send notifications.
3. Store and retrieve machine configuration information dynamically: Utilize Amazon SimpleDB to fetch config data during boot-time of an instance (eg. database connection strings). SimpleDB may also be used to store information about an instance such as its IP address, machine name and role.
4. Design a build process such that it dumps the latest builds to a bucket in Amazon S3; download the latest version of an application from during system startup.
5. Invest in building resource management tools (Automated scripts, pre-configured images) or Use smart open source configuration management tools like Chef, Puppet, CFEngine or Genome.
6. Bundle Just Enough Operating System (JeOS) and your software dependencies into an Amazon Machine Image so that it is easier to manage and maintain. Pass configuration files or parameters at launch time and retrieve user data and instance metadata after launch.
7. Reduce bundling and launch time by booting from Amazon EBS volumes and attaching multiple Amazon EBS volumes to an instance. Create snapshots of common volumes and share snapshots among accounts wherever appropriate.
8. Application components should not assume health or location of hardware it is running on. For example, dynamically attach the IP address of a new node to the cluster. Automatically failover and start a new clone in case of a failure.

### Think parallel

The cloud makes parallelization effortless. Whether it is requesting data from the cloud, storing data to the cloud, processing data (or executing jobs) in the cloud, you need to internalize the concept of parallelization when designing architectures in the cloud. It is advisable to not only implement parallelization wherever possible but also automate it because the cloud allows you to create a repeatable process every easily.

When it comes to accessing (retrieving and storing) data, the cloud is designed to handle massively parallel operations. In order to achieve maximum performance and throughput, you should leverage request parallelization.

Multi-threading your requests by using multiple concurrent threads will store or fetch the data faster than requesting it sequentially. Hence, wherever possible, the processes of a cloud application should be made thread-safe through a share-nothing philosophy and leverage multi-threading. When it comes to processing or executing requests in the cloud, it becomes even more important to leverage parallelization.

A general best practice, in the case of a web application, is to distribute the incoming requests across multiple asynchronous web servers using load balancer. In the case of batch processing application, you can master node can spawn up multiple slave worker nodes that processes task in parallel (as in distributed processing frameworks like Hadoop)

The beauty of the cloud shines when you combine elasticity and parallelization. Your cloud application can bring up a cluster of compute instances which are provisioned within minutes with just a few API calls, perform a job by executing tasks in parallel, store the results and terminate all the instances.

Tactics for parallelization:

1. Multi-thread your Amazon S3 requests
2. Multi-thread your Amazon SimpleDB GET and BATCHPUT requests
3. Create a JobFlow using the Amazon Elastic MapReduce Service for each of your daily batch processes (indexing, log analysis etc.) which will compute the job in parallel and save time.
4. Use the Elastic Load Balancing service and spread your load across multiple web app servers dynamically Keep dynamic data closer to the compute and static data closer to the end-user.

In general it's a good practice to keep your data as close as possible to your compute cluster or processing elements to reduce latency. In the cloud, this best practice is even more relevant and important because you often have to deal with Internet latencies.

Moreover, in the cloud, you are paying for bandwidth in and out of the cloud by the gigabyte of data transfer and the cost can add up very quickly. If a large quantity of data that needs to be processed resides outside of the cloud, it might be cheaper and faster to "ship" and transfer the data to the cloud first and then perform the computation.

For example, in the case of a data warehousing application, it is advisable to move the dataset to the cloud and then perform parallel queries against the dataset. In the case of web applications that store and retrieve data from relational databases, it is advisable to move the database as well as the app server into the cloud all at once. If the data is generated in the cloud, then the applications that consume the data should also be deployed in the cloud so that they can take advantage of in-cloud free data transfer and lower latencies.

For example, in the case of an ecommerce web application that generates logs and clickstream data, it is advisable to run the log analyzer and reporting engines in the cloud. Conversely, if the data is static and not going to change often (for example, images, video, audio, PDFs, JS, CSS files), it is advisable to take advantage of a content delivery service so that the static data is cached at an edge location closer to the end-user (requester) thereby lowering the access latency. Due to the caching, a content delivery service provides faster access to popular objects.

Tactics for implementing this best practice:

1. Ship your data drives to Amazon using the Import/Export service. It may be cheaper and faster to move large amounts of data using the sneakernet than to upload using the Internet.
2. Utilize the same Availability Zone to launch a cluster of machines.
3. Create a distribution of your Amazon S3 bucket and let Amazon CloudFront caches content in that bucket across all the several edge locations around the world

locations around the world.