

# Cloud Architecture Best practices

Omid Vahdaty, Cloud Architect




# Generic Architecture questions:

- **When would you cloudify?**
  - opex
  - capacity unclear
  - Elasticity
  - Agility in development



# Generic Architecture questions:

- **When not to cloudify?**

- Network latency
  - Long terms costs
  - Building datacenter is your business
  - Where vendor lockin is important. (offloading terabytes from Amazon??)
  - The architecture of your app is not a match to the clouds best practices.
  - Regulation?
  - Performance considerations. e.g SQream.
- 


# Generic Architecture questions:

- **When would you normalize tables?**
  - eliminate redundant data, utilize space efficiently and reduce update errors.
  - in situations where we are storing immutable data such as financial transactions or a particular day's price list.




# Generic Architecture questions:

- **When NOT to NORMALIZE ?**


- Could you denormalize your schema to create flatter data structures that are easier to scale?
  - When Multiple Joins are Needed to Produce a Commonly Accessed View
  - decided to go with database partitioning/sharding then you will likely end up resorting to denormalization
  - Normalization saves space, but space is cheap!
  - Normalization simplifies updates, but reads are more common!
  - Performance, performance, Performance
- 

# Generic Architecture questions:


- **NoSql advantages?**

- Non-relational and schema-less data model
  - Low latency and high performance
  - Highly scalable
  - SQL queries are not well suited for the object oriented data structures that are used in most applications now.
  - Some application operations require multiple and/or very complex queries. In that case, data mapping and query generation complexity raises too much and becomes difficult to maintain
- 

# Generally speaking - a good architecture?


- stable / HA / DR / graceful failure
  - Structure - start with skeleton, and be able to add future components.
  - Loosely coupled
  - Low complexity on components
  - Passes all basic tests
  - Use Cases matches Tradeoff of architecture (e.g scale out VS. scale up) - i.e each architecture has advantage and disadvantage
  - Performance aspects.
  - Maintenance aspects
- 

# Generally speaking - a good architecture?

- Documented
  - Decoupling storage and compute
  - API, Services - not Servers.(PaaS VS. IaaS. )
  - TCO
  - Hardware considerations
  - Network considerations- AWS VS IBM - no money on private network.
  - Security considerations
- 



# Cloud Architecture Best Practices

- **Scale out, Stateless, Need a dispatcher !**
    - **Pull** - Asynchronous event-driven workloads do not require a load balancing solution because you can implement a pull model instead. In a pull model, tasks that need to be performed or data that need to be processed could be stored as messages in a queue using Amazon Simple Queue Service (Amazon SQS) or as a streaming data solution like Amazon Kinesis.
    - **Push** ( like elastic load balancer or route 53 -)
- 


# Cloud Architecture Best Practices

- **Scale out, Making it stateless**

- web applications can use HTTP **cookies** to store information about a session at the client's browser (e.g., items in the shopping cart). The browser passes that information back to the server at each subsequent request so that the application does not need to store it. However, there are two **drawbacks** with this approach.
  - First, the content of the HTTP **cookies can be tampered** with at the client side, so you should always treat them as untrusted data that needs to be validated.
  - Second, HTTP **cookies are transmitted with every request**, which means that you should keep their size to a minimum (to avoid unnecessary latency).

# Cloud Architecture Best Practices

- **Scale out, Making it stateless**

- Consider storing a unique session identifier in a HTTP cookie and storing more detailed user session information server-side. Most programming platforms provide a native session management mechanism that works this way, however this is often stored on the local file system by default. This would result in a stateful architecture. A common solution to this problem is to store user session information in a database. Amazon DynamoDB is a great choice due to its scalability, high availability, and durability characteristics.
- 

# Cloud Architecture Best Practices

- **Scale out, Making it stateless**

- Other scenarios require storage of larger files (e.g., user uploads, interim results of batch processes, etc.). By placing those files in a shared storage layer like Amazon S3 or Amazon Elastic File System (Amazon EFS) you can avoid the introduction of stateful components. Another example is that of a complex multistep workflow where you need to track the current state of each execution.



# Cloud Architecture Best Practices


- **Scaleout, Stateful**

- by definition, databases are stateful
- “Sticky sessions for HTTP/s” - You might still be able to scale those components horizontally by distributing load to multiple nodes with “session affinity.” In this model, you bind all the transactions of a session to a specific compute resource. You should be aware of the limitations of this model. Existing sessions do not directly benefit from the introduction of newly launched compute nodes. More importantly, session affinity cannot be guaranteed. For example, when a node is terminated or becomes unavailable, users bound to it will be disconnected and experience a loss of session-specific data (e.g., anything that is not stored in a shared resource like S3, EFS, a database, etc.).



# Cloud Architecture Best Practices

- **Elastic resources**

- **Kill unused** resources
  - **Don't use fixed IP** unless a must
  - **Immutable infrastructure pattern.** With this approach a server, once launched, is never updated throughout its lifetime. Instead, when there is a problem or a need for an update the **server is replaced with a new one that has the latest configuration.** In this way, resources are always in a consistent (and tested) state and rollbacks become easier to perform.
  - **Bootstrapping**
- 



# Cloud Architecture Best Practices

- **Elastic resources**

- When you launch an AWS resource like an Amazon EC2 instance or Amazon Relational Database (Amazon RDS) DB instance, you start with a default configuration. You can then execute automated bootstrapping actions. That is, scripts that install software or copy data to bring that resource to a particular state. You can parameterize configuration details that vary between different environments (e.g., production, test, etc.) so that the same scripts can be reused without modifications.



# Cloud Architecture Best Practices

- **Elastic resources**

- AMI Images - faster start times and removes dependencies to configuration services or third-party repositories. This is important in auto-scaled environments
- Convert vmware to ami
- AWS Elastic Beanstalk and the Amazon EC2 Container Service (Amazon ECS) support Docker and enable you to deploy and manage multiple Docker containers across a cluster of Amazon EC2 instances.





# Cloud Architecture Best Practices

- **Elastic resources**

- **AWS Elastic Beanstalk** follows the hybrid model. It provides preconfigured run time environments (each initiated from its own AMI) but allows you to run bootstrap actions (through configuration files called .ebextensions 11) and configure environment variables to parameterize the environment differences.
- **Infrastructure as code (cloud Formations):** Write code to set up new environments - and keep it versioned.



# Cloud Architecture Best Practices

- **Automation**

- **BeansTalk** - just add code, scaling, resource are taken care of automatically
- **EC2 auto recovery** - coupled with cloud watch - restore to specific state
- **AutoScaling**
- **Amazon CloudWatch Alarms with SNS**, Those Amazon SNS messages can automatically kick off the **execution** of a subscribed **AWS Lambda function**, enqueue a notification message to an Amazon SQS queue, or perform a POST request to an HTTP/S endpoint



# Cloud Architecture Best Practices

- Automation
  - **AWS OpsWorks Lifecycle events:** AWS OpsWorks supports **continuous configuration** through lifecycle events that automatically update your instances configuration to adapt to environment changes. These events can be used to **trigger Chef recipes** on each instance to perform specific configuration task
  - **AWS Lambda Scheduled events:** These events allow you to create a **Lambda function** and **direct AWS Lambda to execute it on a regular schedule.**



# Cloud Architecture Best Practices

- API :
  - support backwards compatibility for rollback scenarios.
  - Remember: services - not servers!
  - Auto scaling included
  - Multi AZ redundancy included




# Cloud Architecture Best Practices

- **Loose coupling** - As application complexity increases, a desirable attribute of an IT system is that it can be broken into smaller, loosely coupled component. **services are meant to be loosely coupled**, they should be able to be consumed without prior knowledge of their network topology details. Loose coupling is a crucial element if you want to take advantage of the elasticity of cloud computing where new resources can be launched or terminated at any point in time. **Service discovery while maintaining Loose coupling:**



# Cloud Architecture Best Practices

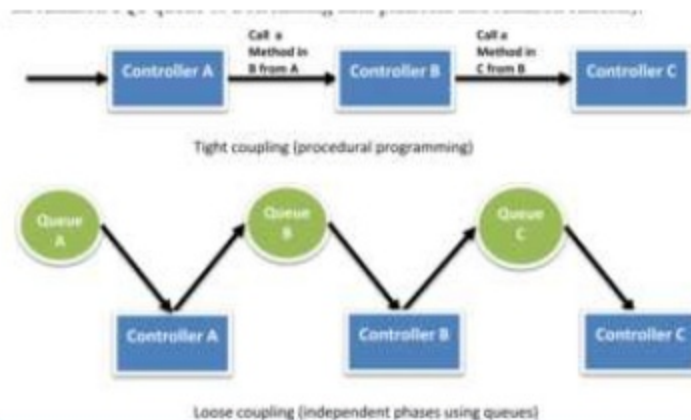
- **Loose coupling Service discovery while maintaining Loose coupling:**

- **service discovery via Elastic Load Balancing** service. Because each load balancer gets its own hostname you now have the ability to consume a service through a stable endpoint. This can be combined with DNS and private Amazon Route53 zones, so that even the particular load balancer's endpoint can be abstracted and modified at any point in time.
  - use a **service registration and discovery** method to **allow retrieval of the endpoint IP addresses and port number of any given service**. Because service discovery becomes the glue between the components, it is important that it is highly available and reliable
- 

# Cloud Architecture Best Practices


- **Loose coupling Service discovery while maintaining Loose coupling:**

- **Asynchronous integrations** - This model is suitable for any interaction that does **not need an immediate response and where an acknowledgement that a request has been registered will suffice**. It involves one component that generates events and another that consumes them. The two components do not integrate through direct point-to-point interaction but usually through an intermediate durable storage layer (e.g., an Amazon SQS queue or a streaming data platform like Amazon Kinesis)





# Cloud Architecture Best Practices

- **Loose coupling Service discovery while maintaining Loose coupling:**
    - **Asynchronous integrations** - This approach decouples the two components and introduces additional resiliency. So, for example, if a process that is reading messages from the queue fails, messages can still be added to the queue to be processed when the system recovers. It also allows you to protect a less scalable backend service from front end spikes and find the right tradeoff between cost and processing lag. For example, you can decide that you don't need to scale your database to accommodate for an occasional peak of write queries as long as you eventually process those queries asynchronously with some delay. Finally, by moving slow operations off of interactive request paths you can also improve the end-user experience.
- 




# Cloud Architecture Best Practices

- **Loose coupling Service discovery while maintaining Loose coupling:**
  - **Graceful failure** - Another way to increase loose coupling is to build applications in such a way that they handle component failure in a graceful manner. You can identify ways to reduce the impact to your end users and increase your ability to make progress on your offline procedures, even in the event of some component failure. in practice A request that fails can be retried with an exponential backoff and Jitter strategy or it could be stored in a queue for later processing. For front-end interfaces, it might be possible to provide alternative or cached content instead of failing completely when, for example, your database server becomes unavailable. The Amazon Route 53 DNS failover feature also gives you the ability to monitor your website and automatically route your visitors to a backup site if your primary site becomes unavailable. You can host your backup site as a static website on Amazon S3 or as a separate dynamic environment.




# Cloud Architecture Best Practices: Serverless

- **Serverless architecture (AWS API gateway + lambda)**

- AWS Lambda allows you to write code functions, called handlers, which will execute when triggered by an event.
  - Immediate benefits: easy to maintain, decoupled, and scalable
    - No operating systems to choose, secure, patch, or manage.
    - No servers to right size, monitor, or scale out.
    - No risk to your cost by over-provisioning.
    - No performance impact by under-provisioning.
- 

# Cloud Architecture Best Practices: Serverless

- **Serverless architecture (AWS API gateway + lambda)**
    - Optimizations
      - Includes by design, cloudFront in the BE to cache static requests
      - Lower latency
      - Protection against DDoS
    - You can improve the performance of specific API requests by using Amazon API Gateway to store responses in an optional in-memory cache. This not only provides performance benefits for repeated API requests, but it also reduces backend executions, which can reduce your overall cost.
- 

# Cloud Architecture Best Practices: Databases

- If your application primarily indexes and queries data with no need for joins or complex transactions (especially if you expect a write throughput beyond the constraints of a single instance) consider a NoSQL database instead.
- If you have large binary files (audio, video, and image), it will be more efficient to store the actual files in the Amazon Simple Storage Service (Amazon S3) and only hold the metadata for the files in your database.




# Cloud Architecture Best Practices: Databases

- Use read replicas to increase read performance
- Use aurora instead of mysql
- If your schema cannot be denormalized consider a relational database instead of noSQL



# Cloud Architecture Best Practices: HA

- You should aim to build as much automation as possible in both detecting and reacting to failure. You can use services like ELB and Amazon Route53 to configure health checks and mask failure by routing traffic to healthy endpoints. In addition, Auto Scaling can be configured to automatically replace unhealthy nodes. You can also replace unhealthy nodes using the Amazon EC2 auto recovery feature or services such as AWS OpsWorks and AWS Elastic Beanstalk
  - Asynchronous replication decouples the primary node from its replicas at the expense of introducing replication lag. This means that changes performed on the primary node are not immediately reflected on its replicas. Asynchronous replicas are used to horizontally scale the system's read capacity for queries that can tolerate that replication lag. It can also be used to increase data durability when some loss of recent transactions can be tolerated during a failover.
- 


# Cloud Architecture Best Practices: HA

- Quorum-based replication combines synchronous and asynchronous replication to overcome the challenges of large-scale distributed database systems. Replication to multiple nodes can be managed by defining a minimum number of nodes that must participate in a successful write operation






# Cloud Architecture Best Practices: DR


- Because of the long distance between the two data centers, latency makes it impractical to maintain synchronous cross-datacenter copies of the data. As a result, a failover will most certainly lead to data loss or a very costly data recovery process.
  - Each Availability Zone is engineered to be isolated from failures in other Availability Zones. An Availability Zone is a data center. Availability Zones within a region provide inexpensive, low-latency network connectivity to other zones in the same region
  - In fact, many of the higher level services on AWS are inherently designed according to the Multi-AZ principle
- 




# Cloud Architecture Best Practices: DR

- Because of the long distance between the two data centers, latency makes it impractical to maintain synchronous cross-datacenter copies of the data. As a result, a failover will most certainly lead to data loss or a very costly data recovery process.
  - Each Availability Zone is engineered to be isolated from failures in other Availability Zones. An Availability Zone is a data center. Availability Zones within a region provide inexpensive, low-latency network connectivity to other zones in the same region
  - In fact, many of the higher level services on AWS are inherently designed according to the Multi-AZ principle
- 

# Cloud Architecture Best Practices: cost reduction

- User serverless when possible - don't pay for idle.
  - Monitoring size of compute
  - Turn off unused workloads
  - Auto scaling
  - Purchasing options - reserved etc.
  - Applications can be designed so that they store and retrieve information from fast, managed, in-memory caches
  - Edge caching - cloudFront (CDN), S3
- 

# Stay in touch...

- [Omid Vahdaty](#) 
- +972-54-2384178



 **HALO**  
ANALYTICS



The logo for Jajah Telefonica. It features the word "jajah" in a stylized, pink, sans-serif font, with a small, curved line above the "j". Below "jajah" is the word "Telefonica" in a black, cursive script font. The entire logo is set against a white background.