Architecting for the Cloud

Tejas Parikh (t.parikh@northeastern.edu)
CSYE 6225
Northeastern University

Building & Deploying Applications on Cloud

Lift & Shift

- Migrating application to cloud without significant changes.
- Gain benefits of a secure and cost-efficient infrastructure.

Maximize Cloud Investments

- Make the most of the elasticity and agility that are possible with cloud computing, engineers have to evolve their architectures to take advantage of cloud capabilities.
- For new applications, cloud-specific IT architecture patterns can help drive efficiency and scalability.
- Whether you are rearchitecturing the applications that currently run in your on-premises environment to run on AWS, or designing cloud-native applications, you must consider the differences between traditional environments and cloud computing environments.

Differences Between Traditional and Cloud Computing Environments

On-Premises vs. Cloud

Cloud computing differs from a traditional, on-premises environment in many ways, including flexible, global, and scalable capacity, managed services, built-in security, options for cost optimization, and various operating models.

IT Assets as Provisioned Resources

- In a traditional computing environment, you provision capacity based on an estimate of a theoretical maximum peak. This can result in periods where expensive resources are sitting idle or occasions of insufficient capacity.
- With cloud computing, you can access as much or as little capacity as you need and dynamically scale to meet actual demand, while only paying for what you use.
- On cloud, servers, databases, storage, and higher-level application components can be instantiated within seconds.
- You can treat these as temporary resources, free from the inflexibility and constraints of a fixed and finite IT infrastructure.
- This resets the way you approach change management, testing, reliability, and capacity planning.
- This change in approach encourages experimentation by introducing the ability in processes to fail fast and iterate quickly.

Global, Available, and Scalable Capacity

- Using the global infrastructure of cloud platform provider, you can deploy your application to the Region that best meets your requirements (e.g., proximity to your end users, compliance, data residency constraints, and cost).
- For global applications, you can reduce latency to end users around the world by using content delivery network (CDN).
- This also makes it much easier to operate production applications and databases across multiple data centers to achieve high availability and fault tolerance.
- The global infrastructure of cloud platform provider and the ability to provision capacity as needed let you think differently about your infrastructure as the demands on your applications and the breadth of your services expand.

COMPANIES > APPLE

Apple to Spend $10B on US Data Center Construction Over Five Years

Built-in Security

- In traditional IT environments, infrastructure security auditing can be a periodic and manual process.
- In contrast cloud provides governance capabilities that enable continuous monitoring of configuration changes to your IT resources.
- Since cloud resources are programmable using tools and APIs, you can formalize and embed your security policy within the design of your infrastructure.
- With the ability to spin up temporary environments, security testing can now become part of your continuous delivery pipeline.
- Finally, you can leverage a variety of native security and encryption features that can help you achieve higher levels of data protection and compliance.

Architecting for Cost

- Traditional cost management of on-premises solutions is not typically tightly coupled to the provision of services.
- When you provision a cloud computing environment, optimizing for cost is a fundamental design tenant for architects.
- When selecting a solution, you should not only focus on the functional architecture and feature set but on the cost profile of the solutions you select.

Design Principles

Scalability

- Systems that are expected to grow over time need to be built on top of a scalable architecture.
- Such an architecture can support growth in users, traffic, or data size with no drop-in performance.
- It should provide that scale in a linear manner where adding extra resources results in at least a proportional increase in ability to serve additional load.
- Growth should introduce economies of scale, and cost should follow the same dimension that generates business value out of that system.
- While cloud computing provides virtually unlimited on-demand capacity, your design needs to be able to take advantage of those resources seamlessly.
- There are generally two ways to scale an IT architecture: vertically and horizontally.

Scaling Vertically

- Scaling vertically takes place through an increase in the specifications of an individual resource, such as upgrading a server with a larger hard drive or a faster CPU.
- With Amazon EC2, you can stop an instance and resize it to an instance type that has more RAM, CPU, I/O, or networking capabilities.
- This way of scaling can eventually reach a limit, and it is not always a cost-efficient or highly available approach.
- However, it is very easy to implement and can be sufficient for many use cases especially in the short term.

Scaling Horizontally

- Scaling horizontally takes place through an increase in the number of resources, such as adding more hard drives to a storage array or adding more servers to support an application.
- This is a great way to build internet-scale applications that leverage the elasticity of cloud computing.
- Not all architectures are designed to distribute their workload to multiple resources.

Stateless Applications

- When users or services interact with an application, they will often perform a series of interactions that form a session.
- A session is unique data for users that persists between requests while they use the application.
- A stateless application is an application that does not need knowledge of previous interactions and does not store session information.
- Stateless applications can scale horizontally because any of the available compute resources can service any request.
- Without stored session data, you can simply add more compute resources as needed.
- When that capacity is no longer required, you can safely terminate those individual resources, after running tasks have been drained.
- Those resources do not need to be aware of the presence of their peers—all that is required is a way to distribute the workload to them.

Distribute Load to Multiple Nodes

- To distribute the workload to multiple nodes in your environment, you can choose either a push or a pull model.
- With a push model, you can use Elastic Load Balancing (ELB) to distribute a workload. ELB routes incoming application requests across multiple EC2 instances. When routing traffic, a Network Load Balancer operates at layer 4 of the Open Systems Interconnection (OSI) model to handle millions of requests per second. With the adoption of container-based services, you can also use an Application Load Balancer. An Application Load Balancer provides Layer 7 of the OSI model and supports content- based routing of requests based on application traffic.
- Alternatively, you can use Amazon Route 53 to implement a DNS round robin. In this case, DNS responses return an IP address from a list of valid hosts in a round-robin fashion. While easy to implement, this approach does not always work well with the elasticity of cloud computing. This is because even if you can set low time to live (TTL) values for your DNS records, caching DNS resolvers are outside the control of Amazon Route 53 and might not always respect your settings.
- Instead of a load balancing solution, you can implement a pull model for asynchronous, event-driven workloads. In a pull model, tasks that need to be performed or data that needs to be processed can be stored as messages in a queue or as a streaming data solution.

Stateless Components

- In practice, most applications maintain some kind of state information.
- Store session information in database or external cache.
- Store files in Object Storage (S3) or Network File Share (EFS).

Stateful Components

- Inevitably, there will be layers of your architecture that you won't turn into stateless components. By definition, databases are stateful. In addition, many legacy applications were designed to run on a single server by relying on local compute resources. Other use cases might require client devices to maintain a connection to a specific server for prolonged periods.
- For example, real-time multiplayer gaming must offer multiple players a consistent view of the game world with very low latency. This is much simpler to achieve in a non-distributed implementation where participants are connected to the same server.
- You might still be able to scale those components horizontally by distributing the load to multiple nodes with session affinity. In this model, you bind all the transactions of a session to a specific compute resource. But this model does have some limitations. 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, which is anything that is not stored in a shared resource such as Amazon S3, Amazon EFS, or a database.

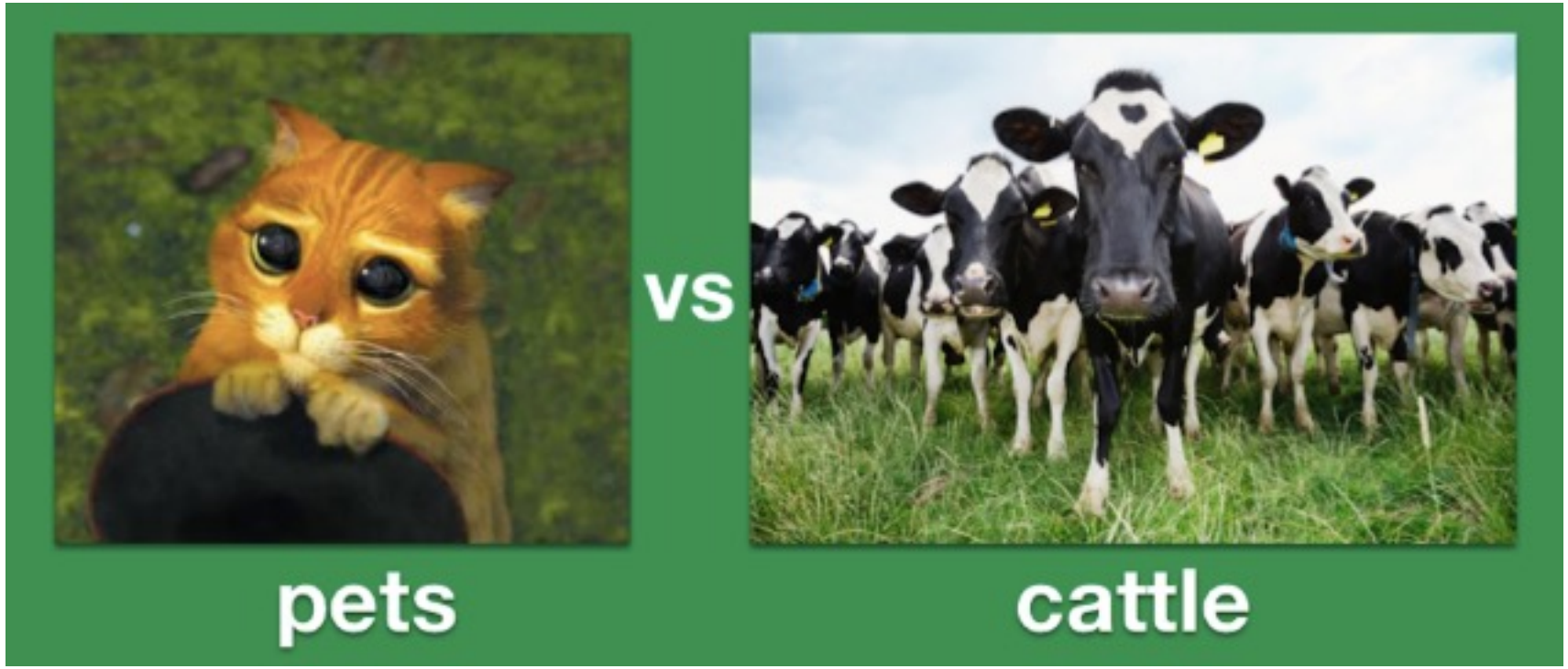
Implement Session Affinity

- For HTTP and HTTPS traffic, you can use the sticky sessions feature of an Application Load Balancer to bind a user's session to a specific instance.
- With this feature, an Application Load Balancer will try to use the same server for that user for the duration of the session.

Distributed Processing

- Use cases that involve the processing of very large amounts of data—anything that can't be handled by a single compute resource in a timely manner—require a distributed processing approach.
- By dividing a task and its data into many small fragments of work, you can execute them in parallel across a set of compute resources.

Pets vs. Cattle



Disposable Resources Instead of Fixed Servers

- In a traditional infrastructure environment, you have to work with fixed resources because of the upfront cost and lead time of introducing new hardware. This drives practices such as manually logging in to servers to configure software or fix issues, hardcoding IP addresses, and running tests or processing jobs sequentially.
- When designing for cloud, you can take advantage of the dynamically provisioned nature of cloud computing. You can think of servers and other components as temporary resources. You can launch as many as you need, and use them only for as long as you need them.
- Another issue with fixed, long-running servers is configuration drift. Changes and software patches applied through time can result in untested and heterogeneous configurations across different environments. You can solve this problem with an immutable infrastructure pattern. With this approach, a server—once launched—is never updated. Instead, when there is a problem or need for an update, the problem server is replaced with a new server that has the latest configuration. This enables resources to always be in a consistent (and tested) state, and makes rollbacks easier to perform. This is more easily supported with stateless architectures.

Bootstrapping

- When you launch an AWS resource such as an EC2 instance or Amazon Relational Database Service (Amazon RDS) DB instance, you start with a default configuration.
- You can then execute automated bootstrapping actions, which are 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 (such as production or test) so that you can reuse the same scripts without modifications.
- You can set up new EC2 instances with user data scripts and cloud-init directives.

Golden Images

Golden Images

- Certain AWS resource types, such as EC2 instances, Amazon RDS DB instances, and Amazon Elastic Block Store (Amazon EBS) volumes, can be launched from a golden image, which is a snapshot of a particular state of that resource.
- When compared to the bootstrapping approach, a golden image results in faster start times and removes dependencies to configuration services or third-party repositories. This is important in auto-scaled environments where you want to be able to quickly and reliably launch additional resources as a response to demand changes.
- You can customize an EC2 instance and then save its configuration by creating an Amazon Machine Image (AMI).
- You can launch as many instances from the AMI as you need, and they will all include those customizations.
- Each time you want to change your configuration you must create a new golden image, so you must have a versioning convention to manage your golden images over time.

Hybrid

- You can also use a combination of the two approaches: some parts of the configuration are captured in a golden image, while others are configured dynamically through a bootstrapping action.
- Items that do not change often or that introduce external dependencies will typically be part of your golden image.
- An example of a good candidate is your web server software that would otherwise have to be downloaded by a third-party repository each time you launch an instance.

Infrastructure as Code

- Application of the principles we have discussed does not have to be limited to the individual resource level. Because assets are programmable, you can apply techniques, practices, and tools from software development to make your whole infrastructure reusable, maintainable, extensible, and testable.
- AWS CloudFormation templates give you an easy way to create and manage a collection of related AWS resources, and provision and update them in an orderly and predictable fashion.
- You can describe the AWS resources and any associated dependencies or runtime parameters required to run your application.
- Your CloudFormation templates can live with your application in your version control repository, which allows you to reuse architectures and reliably clone production environments for testing.

Loose Coupling

- As application complexity increases, a desirable attribute of an IT system is that it can be broken into smaller, loosely coupled components.
- This means that IT systems should be designed in a way that reduces interdependencies—a change or a failure in one component should not cascade to other components.

Service Discovery

- Applications that are deployed as a set of smaller services depend on the ability of those services to interact with each other.
- Because each of those services can be running across multiple compute resources, there needs to be a way for each service to be addressed.
- For example, in a traditional infrastructure, if your front-end web service needs to connect with your back-end web service, you could hardcode the IP address of the compute resource where this service was running.
- Although this approach can still work in cloud computing, if those services are meant to be loosely coupled, they should be able to be consumed without prior knowledge of their network topology details.
- Apart from hiding complexity, this also allows infrastructure details to change at any time.
- 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.
- In order to achieve that you will need some way of implementing service discovery.

Implement Service Discovery

- For an Amazon EC2-hosted service, a simple way to achieve service discovery is through Elastic Load Balancing (ELB). Because each load balancer gets its own hostname, you can consume a service through a stable endpoint. This can be combined with DNS and private Amazon Route 53 zones, so that the particular load balancer's endpoint can be abstracted and modified at any time.
- Another option is to use a service registration and discovery method to allow retrieval of the endpoint IP addresses and port number of any service. Because service discovery becomes the glue between the components, it is important that it is highly available and reliable.
- If load balancers are not used, service discovery should also allow options such as health checks. Amazon Route 53 supports auto naming to make it easier to provision instances for microservices. Auto naming lets you automatically create DNS records based on a configuration you define.
- Other example implementations include custom solutions using a combination of tags, a highly available database, custom scripts that call the AWS APIs, or open-source tools such as Netflix Eureka, Airbnb Synapse, or HashiCorp Consul.

Asynchronous Integration

- Asynchronous integration is another form of loose coupling between services. 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, such as an queue or a streaming data platform.
- This approach decouples the two components and introduces additional resiliency.

Services, Not Servers

- Developing, managing, and operating applications, especially at scale, requires a wide variety of underlying technology components. With traditional IT infrastructure, companies would have to build and operate all those components.
- Cloud offers a broad set of compute, storage, database, analytics, application, and deployment services that help organizations move faster and lower IT costs.
- Architectures that do not leverage that breadth might not be making the most of cloud computing and might be missing an opportunity to increase developer productivity and operational efficiency.

Databases

- With traditional IT infrastructure, organizations are often limited to the database and storage technologies they can use.
- There can be constraints based on licensing costs and the ability to support diverse database engines.
- On cloud, these constraints are removed by managed database services that offer enterprise performance at open- source cost.
- As a result, it is not uncommon for applications to run on top of a polyglot data layer choosing the right technology for each workload.

Choose the Right Database Technology for Each Workload

These questions can help you make decisions about which solutions to include in your architecture:

- Is this a read-heavy, write-heavy, or balanced workload? How many reads and writes per second are you going to need? How will those values change if the number of users increases?
- How much data will you need to store and for how long? How quickly will this grow? Is there an upper limit in the near future? What is the size of each object (average, min, max)? How will these objects be accessed?
- What are the requirements in terms of durability of data? Is this data store going to be your “source of truth?”
- What are your latency requirements? How many concurrent users do you need to support?
- What is your data model and how are you going to query the data? Are your queries relational in nature (e.g., JOINs between multiple tables)? Could you denormalize your schema to create flatter data structures that are easier to scale?

Relational Databases

- Relational databases (also known as RDBS or SQL databases) normalize data into well- defined tabular structures known as tables, which consist of rows and columns.
- They provide a powerful query language, flexible indexing capabilities, strong integrity controls, and the ability to combine data from multiple tables in a fast and efficient manner.
- Managed database makes it easy to set up, operate, and scale a relational database in the cloud with support for many familiar database engines.

Database Scalability

- Relational databases can scale vertically by upgrading to a larger instance or adding more and faster storage.
- For read-heavy applications, you can also horizontally scale beyond the capacity constraints of a single DB instance by creating one or more read replicas.
- Read replicas are separate database instances that are replicated asynchronously. As a result, they are subject to replication lag and might be missing some of the latest transactions.
- Application designers need to consider which queries have tolerance to slightly stale data. Those queries can be executed on a read replica, while the remainder should run on the primary node.
- Read replicas can not accept any write queries.

High Availability

- For any production relational database use the Amazon RDS Multi-AZ deployment feature, which creates a synchronously replicated standby instance in a different Availability Zone.
- In case of failure of the primary node, Amazon RDS performs an automatic failover to the standby without the need for manual administrative intervention.
- When a failover is performed, there is a short period during which the primary node is not accessible.
- Resilient applications can be designed for Graceful Failure by offering reduced functionality, such as read-only mode by using read replicas.
- Amazon Aurora provides multi-master capability to enable reads and writes to be scaled across Availability Zones and also supports cross-Region replication.

Anti-Patterns

- 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 Amazon S3 and only hold the metadata for the files in your database.

NoSQL Databases

- NoSQL databases trade some of the query and transaction capabilities of relational databases for a more flexible data model that seamlessly scales horizontally.
- NoSQL databases use a variety of data models, including graphs, key-value pairs, and JSON documents, and are widely recognized for ease of development, scalable performance, high availability, and resilience.

Removing Single Points of Failure

- Production systems typically come with defined or implicit objectives for uptime.
- A system is highly available when it can withstand the failure of an individual component or multiple components, such as hard disks, servers, and network links.
- To help you create a system with high availability, you can think about ways to automate recovery and reduce disruption at every layer of your architecture.

Introducing Redundancy

- Single points of failure can be removed by introducing redundancy, which means you have multiple resources for the same task. Redundancy can be implemented in either standby or active mode.
- In standby redundancy, when a resource fails, functionality is recovered on a secondary resource with the failover process.
- The failover typically requires some time before it completes, and during this period the resource remains unavailable.
- The secondary resource can either be launched automatically only when needed (to reduce cost), or it can already be running idle (to accelerate failover and minimize disruption).
- Standby redundancy is often used for stateful components such as relational databases.
- In active redundancy, requests are distributed to multiple redundant compute resources. When one of them fails, the rest can simply absorb a larger share of the workload.
- Compared to standby redundancy, active redundancy can achieve better usage and affect a smaller population when there is a failure.

Detect Failure

- You should aim to build as much automation as possible in both detecting and reacting to failure.
- You can use services such as ELB and Amazon Route 53 to configure health checks and mask failure by routing traffic to healthy endpoints.
- You can replace unhealthy nodes automatically using Auto Scaling or by using the Amazon EC2 auto-recovery.
- It won’t be possible to predict every possible failure scenario on day one.
- Make sure you collect enough logs and metrics to understand normal system behavior. After you understand that, you will be able to set up alarms for manual intervention or automated response.

Optimize for Cost

- When you move your existing architectures into the cloud, you can reduce capital expenses and drive savings as a result of the AWS economies of scale.
- By iterating and using more cloud capabilities, you can realize further opportunity to create cost- optimized cloud architectures.

Right Sizing

- AWS offers a broad range of resource types and configurations for many use cases. For example, services such as Amazon EC2, Amazon RDS, Amazon Redshift, and Amazon ES offer many instance types. In some cases, you should select the cheapest type that suits your workload's requirements. In other cases, using fewer instances of a larger instance type might result in lower total cost or better performance. You should benchmark your application environment and select the right instance type depending on how your workload uses CPU, RAM, network, storage size, and I/O.
- Similarly, you can reduce cost by selecting the right storage solution for your needs. For example, Amazon S3 offers a variety of storage classes, including Standard, Reduced Redundancy, and Standard-Infrequent Access. Other services, such as Amazon EC2, Amazon RDS, and Amazon ES, support different EBS volume types (magnetic, general purpose SSD, provisioned IOPS SSD) that you should evaluate.
- Over time, you can continue to reduce cost with continuous monitoring and tagging. Just like application development, cost optimization is an iterative process. Because, your application and its usage will evolve over time, and because AWS iterates frequently and regularly releases new options, it is important to continuously evaluate your solution.

Caching

- Caching is a technique that stores previously calculated data for future use.
- This technique is used to improve application performance and increase the cost efficiency of an implementation.
- It can be applied at multiple layers of an IT architecture.

Application Data Caching

- Applications can be designed so that they store and retrieve information from fast, managed, in-memory caches.
- Cached information might include the results of I/O- intensive database queries, or the outcome of computationally intensive processing.
- When the result set is not found in the cache, the application can calculate it, or retrieve it from a database or expensive, slowly mutating third-party content, and store it in the cache for subsequent requests.
- However, when a result set is found in the cache, the application can use that result directly, which improves latency for end users and reduces load on back-end systems.
- Your application can control how long each cached item remains valid.
- In some cases, even a few seconds of caching for very popular objects can result in a dramatic decrease on the load for your database.

Conclusion

- When you design your architecture in the Cloud, it is important to consider the important principles and design patterns, including how to select the right database for your application, and how to architect applications that can scale horizontally and with high availability.
- Because each implementation is unique, you must evaluate how to apply this guidance to your implementation.
- The topic of cloud computing architectures is broad and continuously evolving.

Additional Resources

See Lecture Page