Cloud computing - ch - 5

Architecting on AWS

TOPICS

Introduction to System Design

AWS Essentials Review

System Design for High Availability

Automation and Serverless

Architectures: Event-Driven Scaling

Well-Architected Best Practices: Security, Reliability, Performance Efficiency, Cost, Optimization

Deployment and Implementation: Design Patterns and Sample Architectures

Introduction to System Design

System Design involves identifying sources of data, It is an intuition towards characterizing, creating, and planning a framework to fulfill the necessity and indeed the prerequisites of particular businesses.

Why learn System Design?

In any development process, be it Software or any other tech, the most important stage is Design. Without the designing phase, you cannot jump to the implementation or the testing part.

The same is the case with the System as well.

Objectives of System Design

1. **Practicality:** We need a system that should be targetting the set of audiences(users) corresponding to which they are designing.
2. **Accuracy:** Above system design should be designed in such a way it fulfills nearly all requirements around which it is designed be it functional o non-functional requirements.
3. **Completeness:** System design should meet all user requirements
4. **Efficient:** The system design should be such that it should not overuse surpassing the cost of resources nor under use as it will by now we know will result in low thorough put (output) and less response time(latency).
5. **Reliability:** The system designed should be in proximity to a failure-free environment for a certain period of time.

System Development Life Cycle

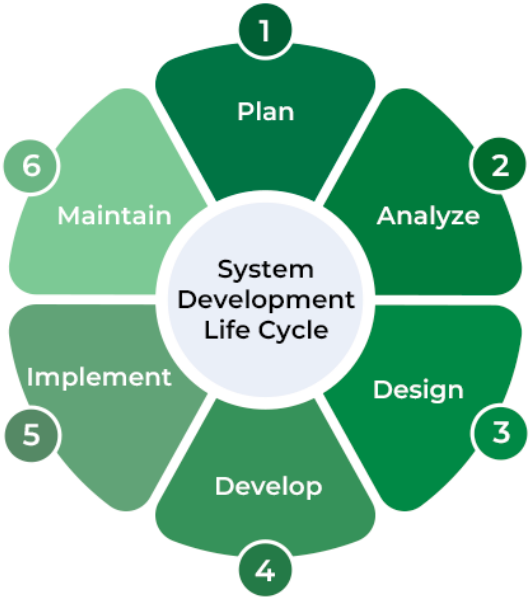
Any system doesn't get designed in a single day or in a single step. For every system design, there are a series of steps/phases/stages that takes place to get a strong system, this series is defined as **System Development Life Cycle** (SDLC).

We need a strong understanding of understanding life-cycle of a system just likely to define the scope for any variable in code chunk because then only we will be able to grasp deep down how humongous systems are interacting within machines in the real world.

Stages of SDLC

The stages in System Development Life Cycle are as follows:

1. Plan
2. Analyse
3. Design
4. Develop
5. Implement
6. Maintain



Components of System Design

1. **Load balancers:** Most crucial component for scalability, availability, and performance measures for systems.
2. **Key Value Stores:** It is a storage system similar to hashtables where key-value stores are distributed hash tables.
3. **Blob Storage:** Blob stands for binary large objects, as the name suggests is storage for unstructured data such as YouTube, and Netflix.
4. **Databases:** It is an organized collection of data so that they can be easily accessed and modified.
5. **Rate Limiters:** These sets the maximum number of requests a service can fulfill.
6. **Monitoring System:** These are basically software where system administrator monitor infrastructures such as bandwidth, CPU, routers, switches, etc.
7. **Distributed System Messaging Queue:** Transaction medium between producers and consumers.
8. **Distributed Unique ID generator:** In the case of large distributed systems, every moment multiple tasks are occurring so in order to distinguish it assign a tag corresponding to every event.

AWS Essentials Review

AWS ESSENTIALS

AWS Technical Essentials introduces to essential AWS services and common solutions.

The course covers the fundamental AWS concepts related to compute, database, storage, networking, monitoring, and security.

It will showcase working on AWS through hands-on course experiences.

The course covers the concepts necessary to increase understanding of AWS services, so that we can make informed decisions about solutions that meet business requirements.

Throughout the course, we will gain information on how to build, compare, and apply highly available, fault tolerant, scalable, and cost-effective cloud solutions.

System Design for High Availability

System Design for High Availability

High-availability systems are designed to minimize downtime and maximize performance, reliability, and user satisfaction.

They are essential for critical applications and services that cannot afford to fail or slow down

Define your availability goals

- Before you start designing your system, you need to define your availability goals and metrics.
- Availability is usually measured by the percentage of time that your system is operational and functional.
- For example, a 99.9% availability means that your system is down for less than nine hours per year.
- However, availability also depends on other factors, such as latency, throughput, scalability, and fault tolerance.
- You need to specify your target values and thresholds for each of these factors and how you will monitor and evaluate them.

Choose the right architecture

- The architecture of your system determines how it is structured, organized, and distributed.
- It affects how your system handles load balancing, redundancy, replication, backup, recovery, and failover.
- There are different types of architectures for high-availability systems, such as monolithic, microservices, serverless, or hybrid.
- Each type has its advantages and disadvantages, depending on your use case, resources, and preferences.
- You need to choose the right architecture that suits your needs and supports your availability goals.

Implement best coding practices

- The quality of your code also affects the availability of your system.
- You need to implement best coding practices that ensure your code is readable, maintainable, testable, and secure.
- You need to follow coding standards, conventions, and guidelines that make your code consistent and easy to understand.

Test your system thoroughly

- Testing your system is crucial to verify its functionality, performance, and reliability.
- You need to test your system thoroughly at different levels, such as unit, integration, system, and acceptance.
- You need to use different testing methods, such as functional, non-functional, load, stress, and security testing.
- You need to simulate different scenarios, such as normal, peak, and failure conditions. You need to identify and fix any issues or defects that can affect your system's availability.

Automation and Serverless

Automation and Serverless

Aws has number of service providing automation while using serverless architecture for automating deployment process.

Few of the serverless services by AWS:

1. AWS Lambda
2. AWS Fargate
3. AWS Step Functions
4. Amazon EventBridge

Serverless on AWS

- AWS offers technologies for running code, managing data, and integrating applications, all without managing servers.
- Serverless technologies feature automatic scaling, built-in high availability, and a pay-for-use billing model to increase agility and optimize costs.
- These technologies also eliminate infrastructure management tasks like capacity provisioning and patching, so you can focus on writing code that serves your customers.
- Serverless applications start with AWS Lambda, an event-driven compute service natively integrated with over 200 AWS services and software as a service (SaaS) applications.

Architectures: Event-Driven Scaling

Architectures: Event-Driven Scaling

An event-driven architecture uses events to trigger and communicate between decoupled services and is common in modern applications built with microservices.

An event is a change in state, or an update, like an item being placed in a shopping cart on an e-commerce website.

Events can either carry the state (the item purchased, its price, and a delivery address) or events can be identifiers (a notification that an order was shipped).

Architectures: Event-Driven Scaling

Event-driven architectures have three key components: event producers, event routers, and event consumers.

A producer publishes an event to the router, which filters and pushes the events to consumers.

Producer services and consumer services are decoupled, which allows them to be scaled, updated, and deployed independently.

Benefits of an event-driven architecture

Scale and fail independently

By decoupling your services, they are only aware of the event router, not each other. This means that your services are interoperable, but if one service has a failure, the rest will keep running.

Develop with agility

You no longer need to write custom code to poll, filter, and route events; the event router will automatically filter and push events to consumers. The router also removes the need for heavy coordination between producer and consumer services, speeding up your development process.

Audit with ease

An event router acts as a centralized location to audit your application and define policies. These policies can restrict who can publish and subscribe to a router and control which users and resources have permission to access your data. You can also encrypt your events both in transit and at rest.

Cut costs

Event-driven architectures are push-based, so everything happens on-demand as the event presents itself in the router. This way, you're not paying for continuous polling to check for an event. This means less network bandwidth consumption, less CPU utilization, less idle fleet capacity, and less SSL/TLS handshakes.

Well-Architected Best Practices

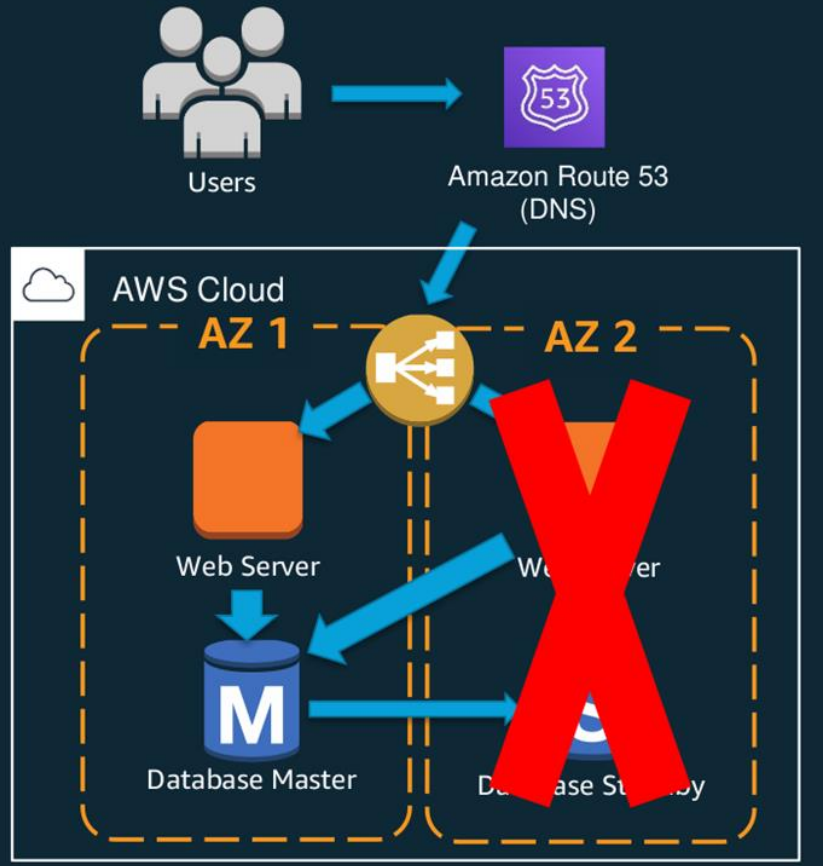
Cloud Architecture Best Practices

1. Design for failure and nothing fails
2. Build security in every layer
3. Leverage different storage options
4. Implement elasticity
5. Think parallel
6. Loose coupling sets you free
7. Don’t fear constraints
8. Use Caching

Design for Failure: Best Practices

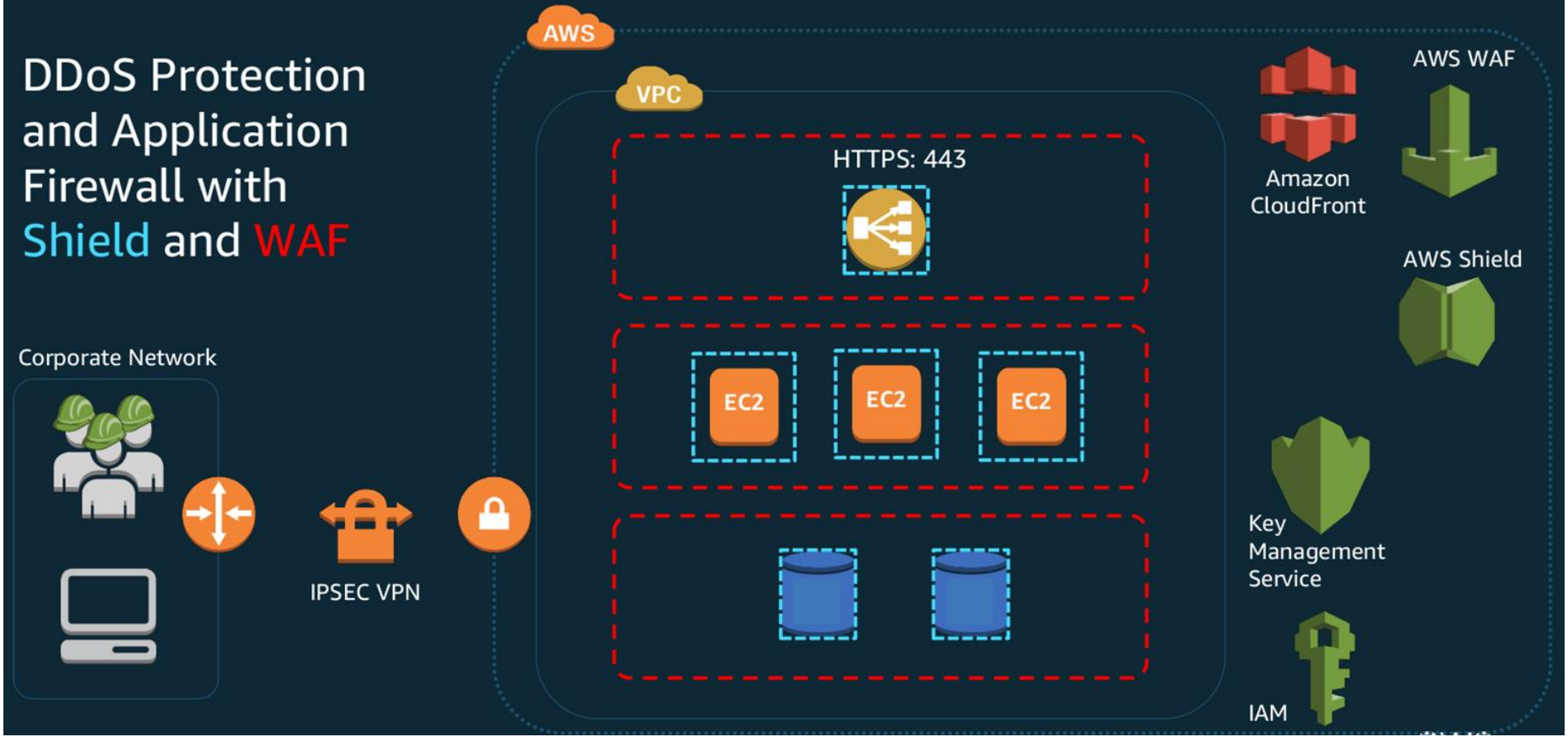
Best Practices:

- Eliminate single points of failure
- Use multiple Availability Zones
- Use Elastic Load Balancing
- Do real-time monitoring with CloudWatch
- Create a database standby across Availability Zones



Build Security in Every Layer

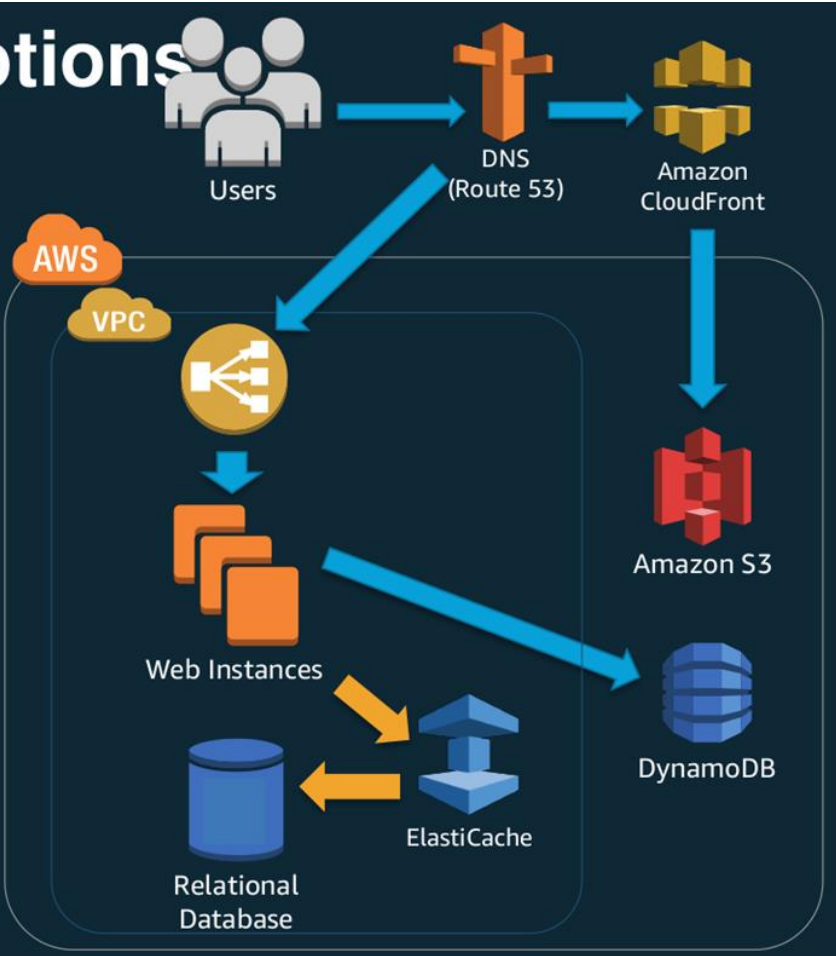
DDoS Protection and Application Firewall with Shield and WAF



Leverage Many Storage Options

Cache frequent queries to shift the load off of your database:

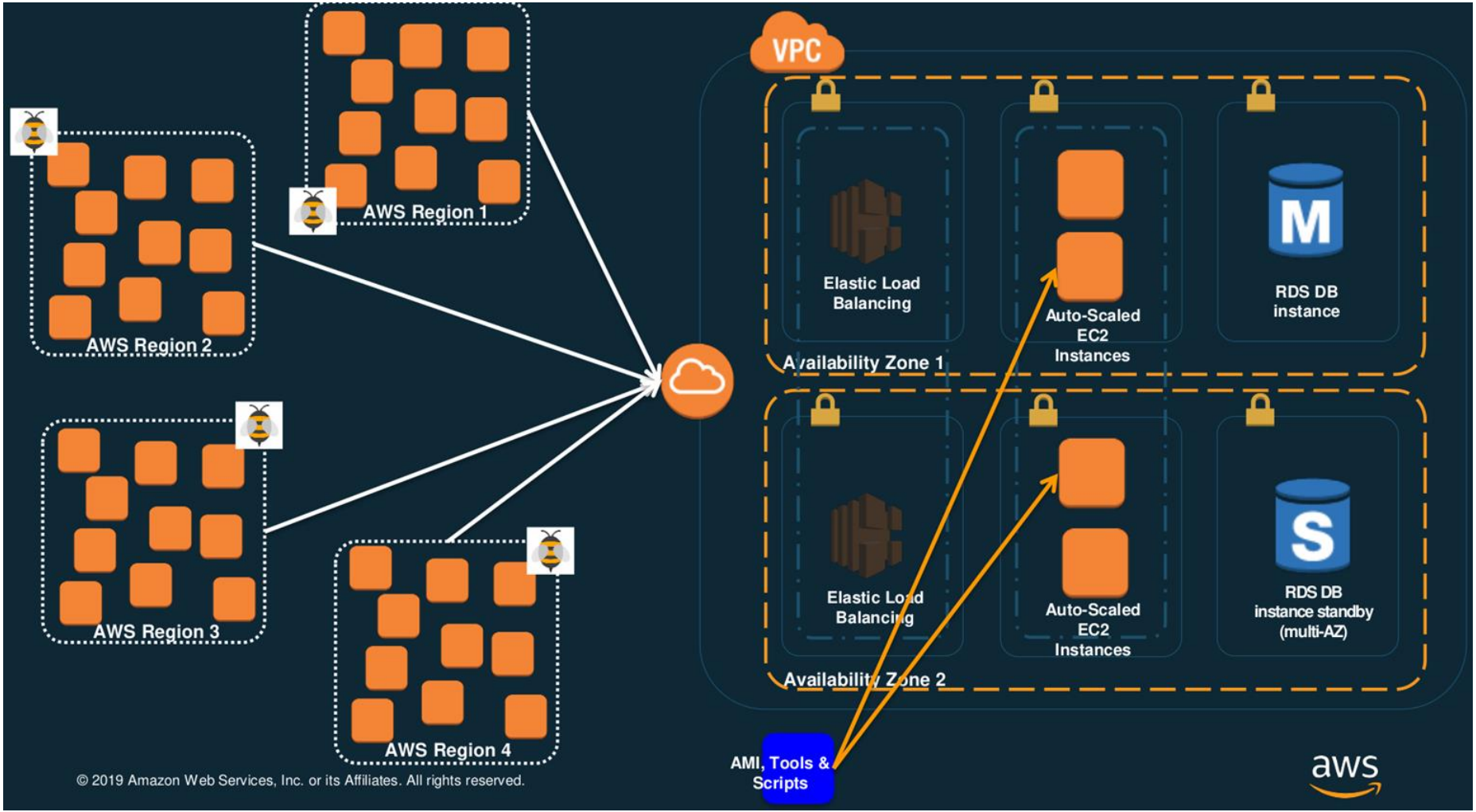
- Put ElastiCache as a caching layer between the web hosts and the database



Implement Elasticity

How To Guide:

- Write Auto Scaling policies with your specific application access patterns in mind
- Prepare your application to be flexible: don't assume the health, availability, or fixed location of components
- Architect resiliency to reboot and relaunch
    - When an instance launches, it should ask "*Who am I and what is my role?*"
- Leverage highly scalable, managed services such as S3 and DynamoDB

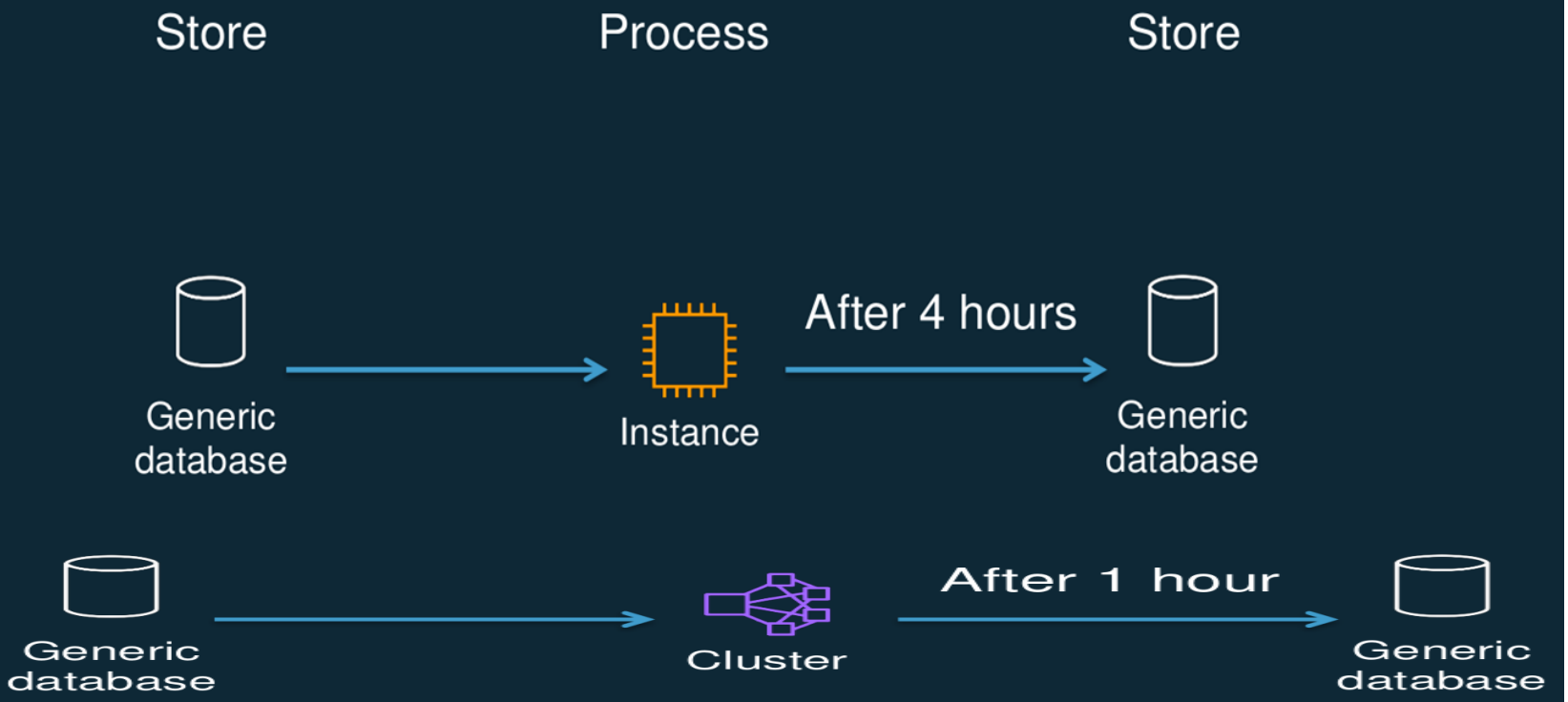


Think Parallel

Scale Horizontally, Not Vertically

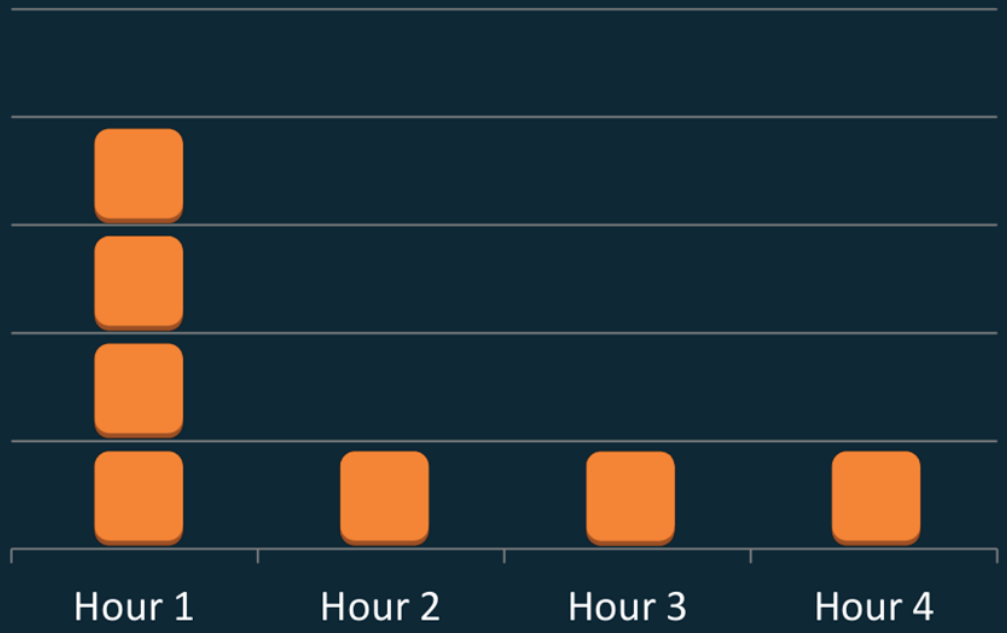
- Decouple compute from state/session data
- Use ELBs to distribute load
- Break up big data into pieces for distributed processing
    - AWS Elastic Map Reduce (EMR) – managed Hadoop

Example – Data Processing



Think Parallel

Faster doesn't need to mean more expensive!



- One Server working for Four hours costs the same as Four servers working for One hour
- Combine with elasticity to increase capacity when you need it most
- The beauty of the cloud shines when you combine elasticity and parallelization

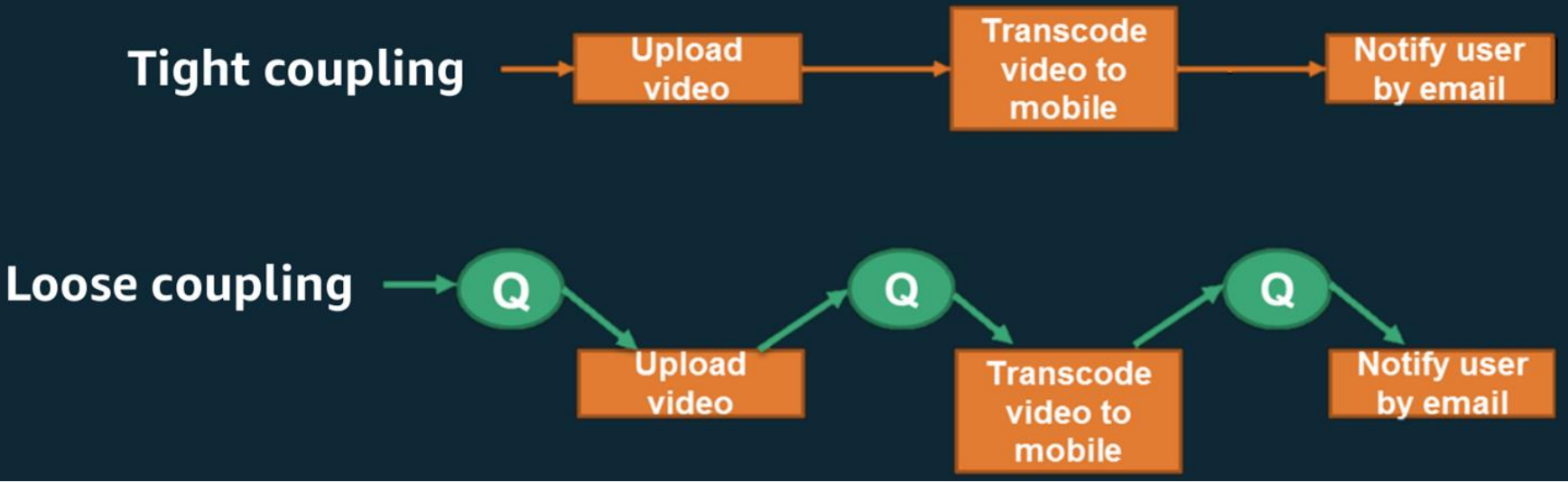
Think Parallel

Parallelize using native managed services

- Get the best performance out of S3 with parallelized reads/writes
  - Multi-part uploads (API) and byte-range GETs (HTTP)
- Take on high concurrency with Lambda
  - Initial soft limit: 1000 concurrent requests per region

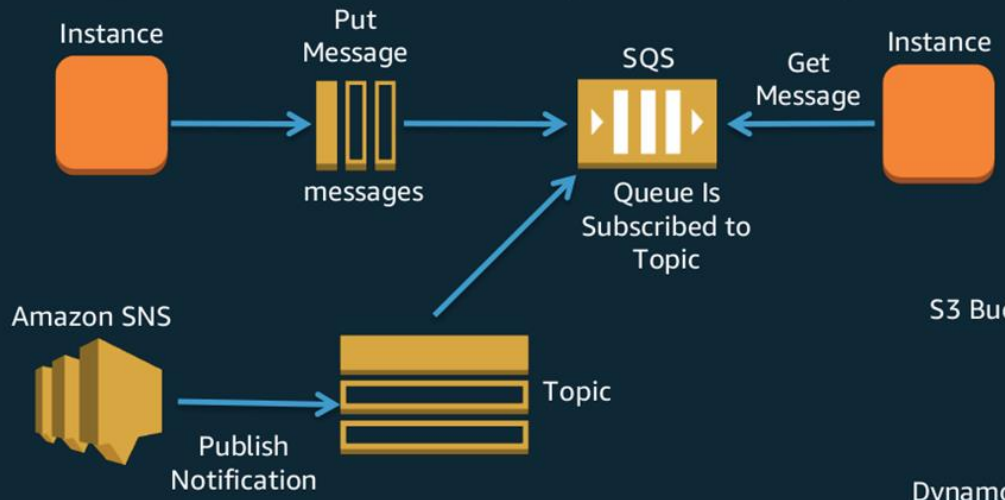
Loose Coupling Sets You Free: Queueing

Use Amazon Simple Queue Service (SQS) to pass messages between loosely coupled components

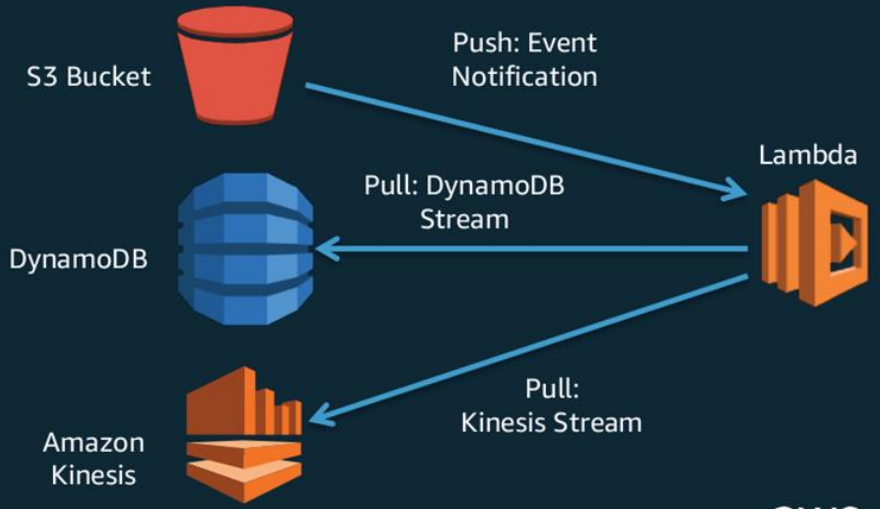


Loose Coupling Sets You Free

Using SNS and SQS to asynchronously scale:



Using Lambda triggers to decouple actions:



Don’t Fear Constraints

Rethink traditional architectural constraints

Need more RAM?

- Don’t: vertically scale
- Do: distribute load across machines or a shared cache

Need better IOPS for database?

- Don’t: rework schema/indexes or vertically scale
- Do: create read replicas, implement sharding, add a caching layer

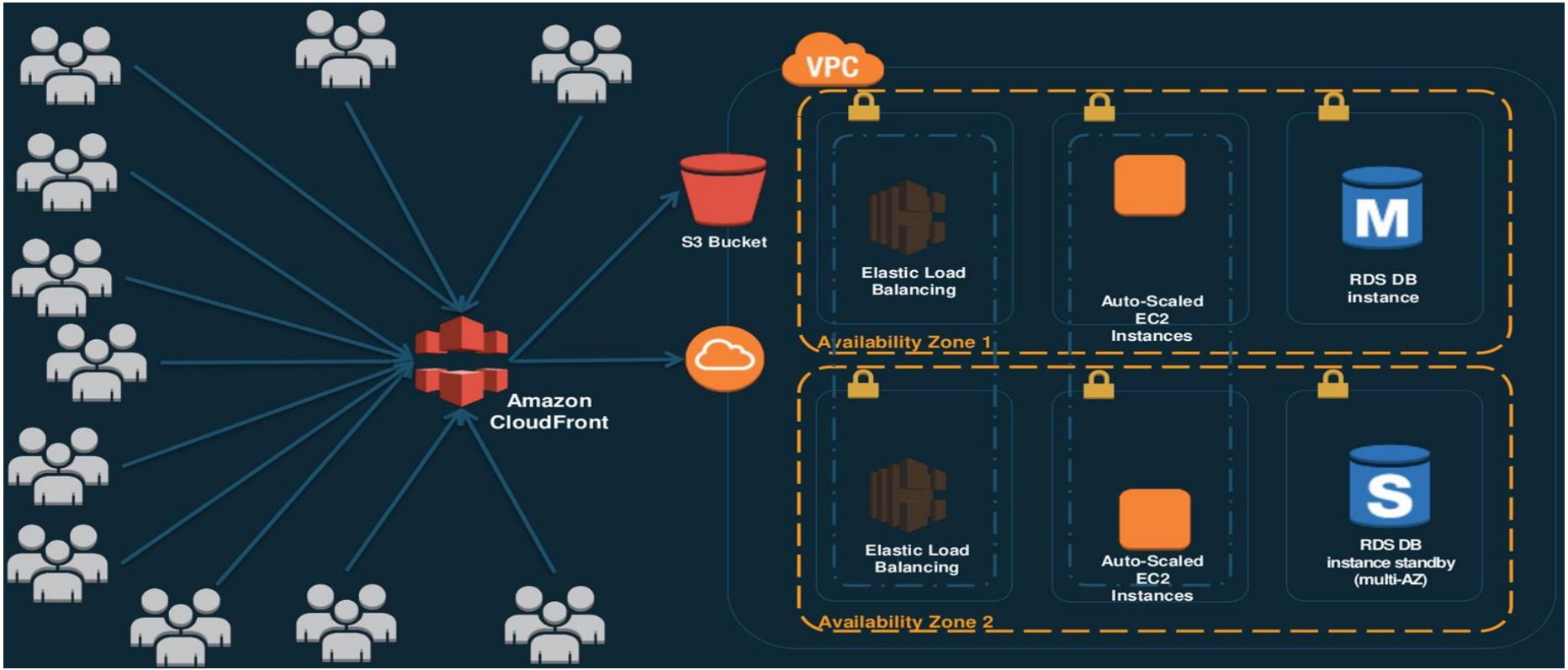
Hardware failed or config got corrupted?

- Don’t: waste production time diagnosing the problem
- Do: “Rip and replace” – stop/terminate old instance and relaunch

Need a Cost Effective Disaster Recovery (DR) strategy?

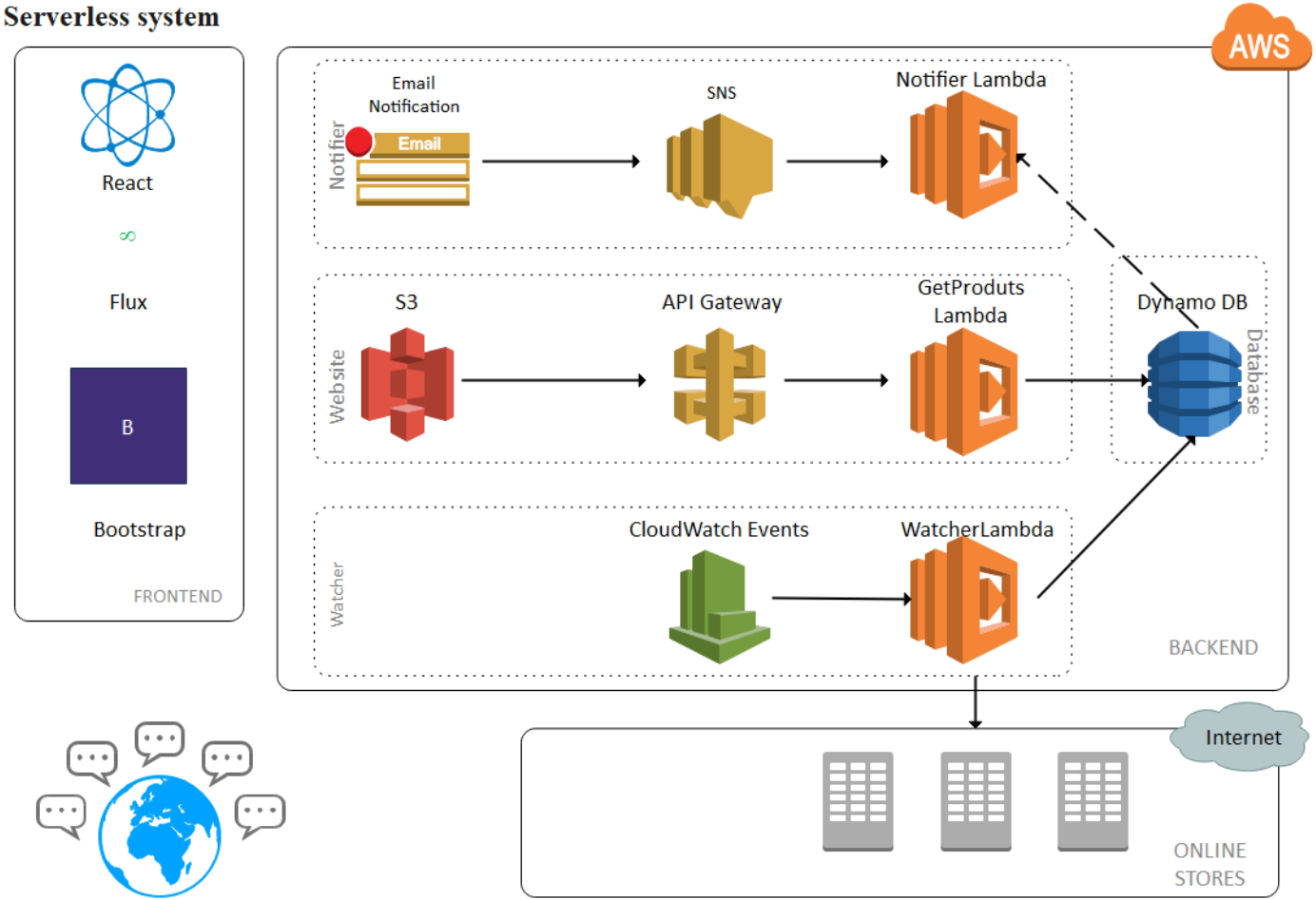
- Don’t: double your infrastructure costs when you don’t need to
- Do: implement Pilot Light or Warm Standby DR stacks

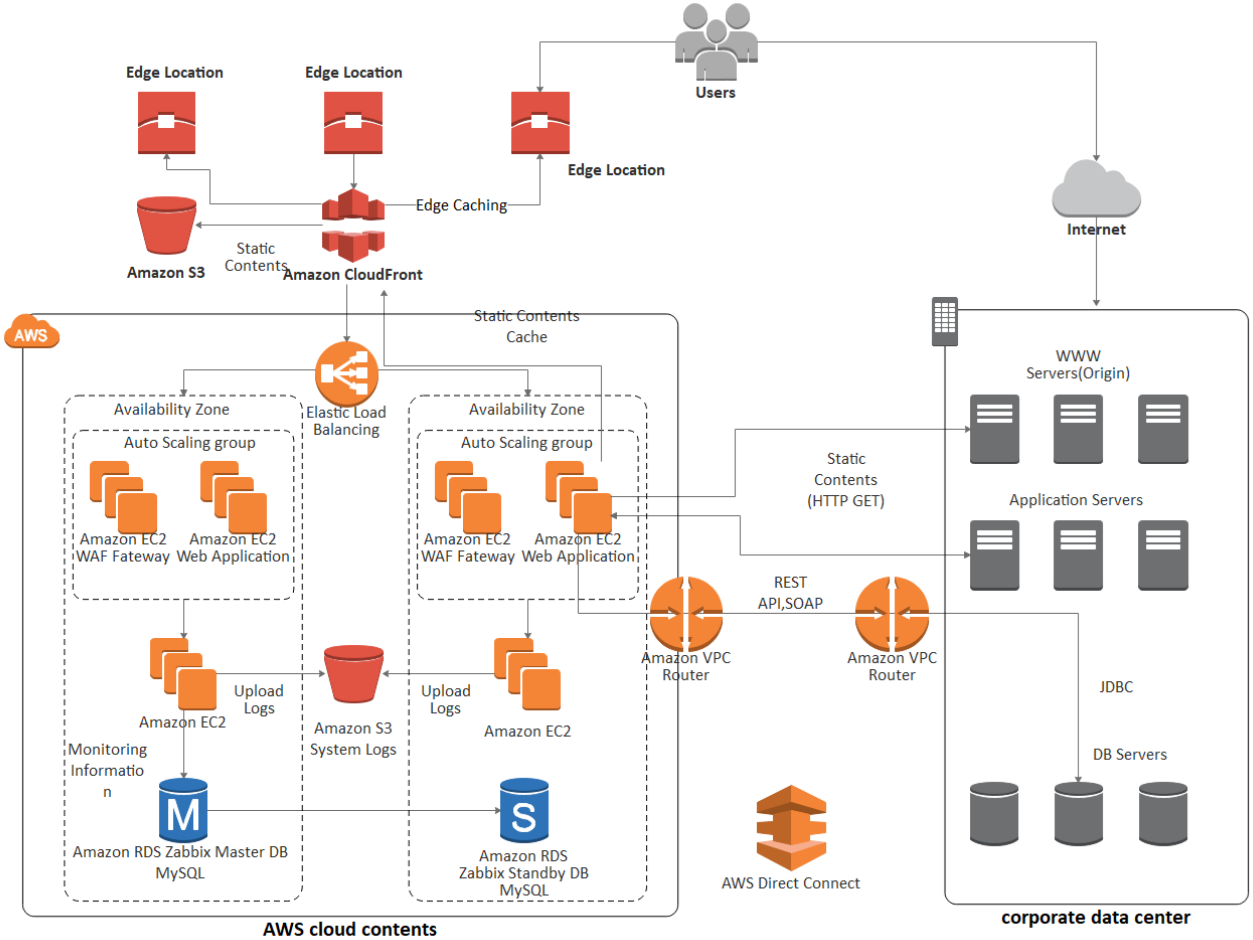
Use Caching

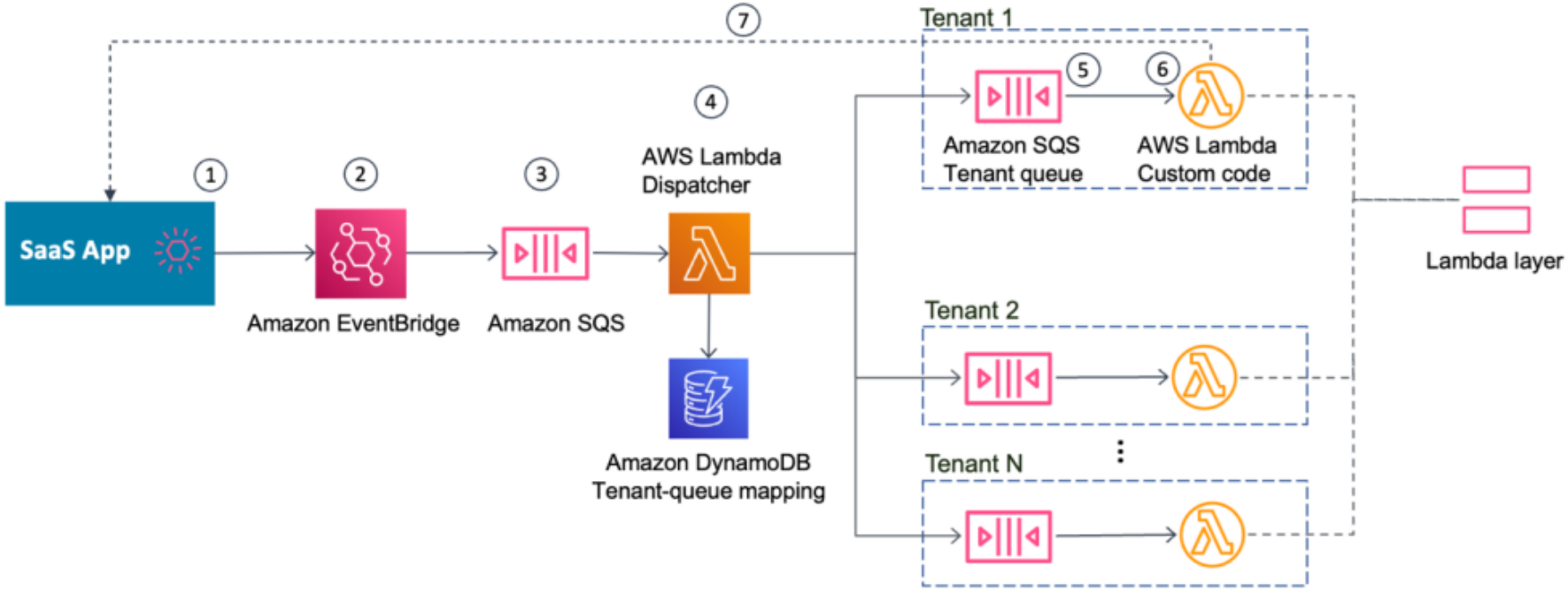

VPC
S3 Bucket
Amazon
CloudFront
Elastic Load
Balancing
Auto-Scaled
EC2
Instances
RDS DB
instance
Availability Zone 1
Elastic Load
Balancing
Auto-Scaled
EC2
Instances
RDS DB
instance standby
(multi-AZ)
Availability Zone 2

Sample Architectures

Serverless system




Users
Edge Location
Edge Location
Edge Location
Edge Caching
Amazon S3
Static Contents
Amazon CloudFront
Internet
AWS
Static Contents Cache
Elastic Load Balancing
Availability Zone
Auto Scaling group
Amazon EC2 WAF Fateway
Amazon EC2 Web Application
Availability Zone
Auto Scaling group
Amazon EC2 WAF Fateway
Amazon EC2 Web Application
Static Contents (HTTP GET)
WWW Servers(Origin)
Application Servers
REST API,SOAP
Amazon VPC Router
Amazon VPC Router
JDBC
DB Servers
Upload Logs
Amazon EC2
Amazon S3 System Logs
Upload Logs
Amazon EC2
Monitoring Informatio n
M
S
Amazon RDS Zabbix Master DB MySQL
Amazon RDS Zabbix Standby DB MySQL
AWS Direct Connect
AWS cloud contents
corporate data center


7
Tenant 1
5
6
4
AWS Lambda
Dispatcher
Amazon SQS
Tenant queue
AWS Lambda
Custom code
1
2
3
SaaS App
Amazon EventBridge
Amazon SQS
Lambda layer
Tenant 2
Tenant N
Amazon DynamoDB
Tenant-queue mapping

Thats all folks for this chapter !!!!