# Cloud Computing

ECPE 276

# AWS Hosted Services

# AWS Hosted Services

- What can a cloud service do for me beyond providing a raw virtual machine and raw disks attached to the same system?
- Using Amazon as model
  - Similar services exist from Google, Microsoft, …

# AWS SDK

**AWS SDK for Java**

**AWS SDK for .NET**

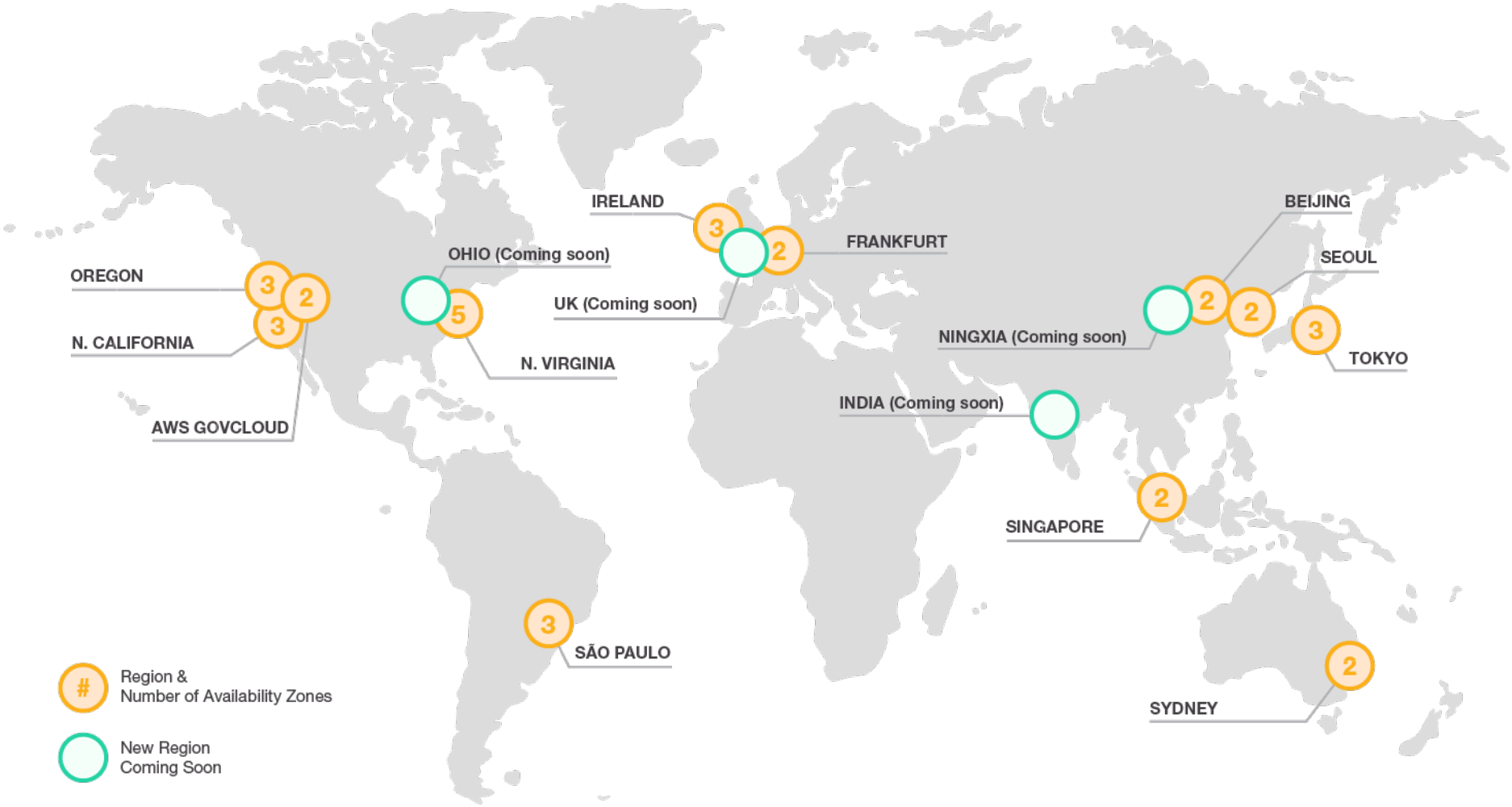
**AWS SDK for Python**

**AWS SDK for PHP**

**AWS SDK for Node.js**

**AWS SDK for Ruby**

*(Or the CLI… Or the web interface…. Or 3rd-party clients/SDKs)*


IRELAND
3
2
FRANKFURT
OHIO (Coming soon)
OREGON
3
2
5
UK (Coming soon)
3
N. CALIFORNIA
N. VIRGINIA
AWS GOVCLOUD
BEIJING
SEOUL
2
2
NINGXIA (Coming soon)
3
TOKYO
INDIA (Coming soon)
2
SINGAPORE
3
SÃO PAULO
2
SYDNEY
# Region & Number of Availability Zones
New Region Coming Soon

# Regions

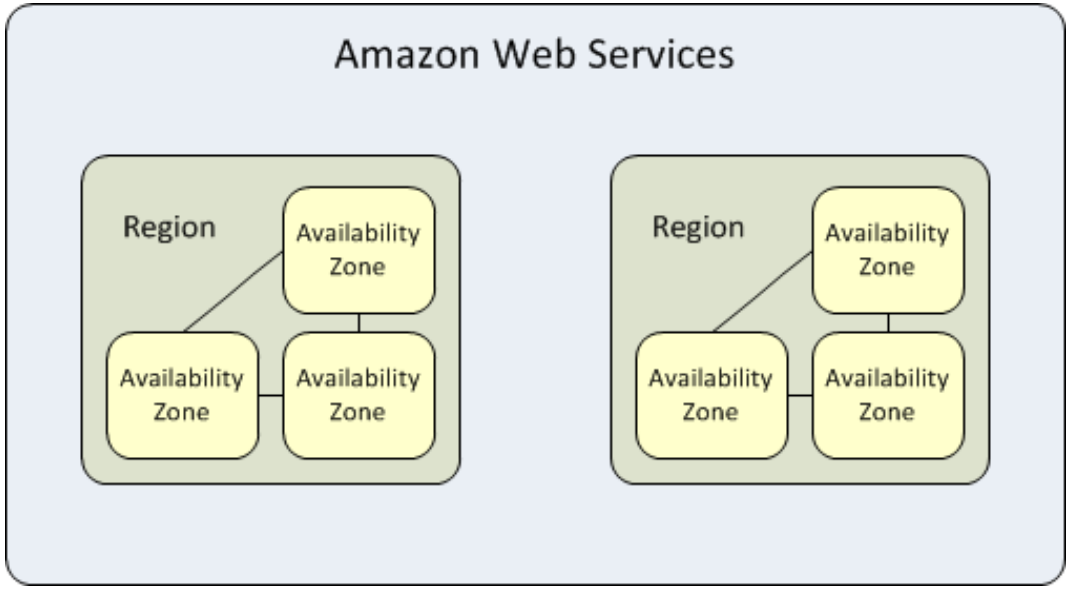
- **Americas**
  - Northern Virginia
  - Oregon
  - Northern California
  - Sao Paulo
  - GovCloud
- **Europe**
  - Ireland
  - Frankfurt
- **Asia**
  - Singapore
  - Tokyo
  - Sydney
  - Seoul
  - Beijing (restricted/isolated)

# Regions - Selection Criteria

- Proximity to customers?
- Proximity to your existing data centers / equipment?
- Remote from your operations for redundancy / disaster recovery?
- Legal / regulatory requirements?
- Cost? (differs by region)

# Availability Zones

- Independent facilities in same geographic area
  - Different power
  - Different network
  - Different building
- Low latency networking between zones



# Storage

# Storage Options

1. Amazon **Simple Storage Service** (S3)
2. Amazon **Elastic Block Storage** (EBS)
3. Amazon **Elastic File System** (EFS)

# Amazon S3 - Overview

- Marketing: “Highly scalable, reliable, and low-latency data storage infrastructure at very low cost”
- **Object storage**
  - Operations: PUT, POST, COPY, DELETE
- Features
  - Automatic versioning (restore old versions)
  - Automatic replication
  - Encryption (AWS keys or your keys)
  - Pay per usage (GB per month + # of requests + bandwidth)

# Amazon S3 – Storage Classes

- Storage classes
  - S3 Standard ($0.0300 per GB)
    - Milliseconds to access
  - S3 Infrequent Access ($0.0125 per GB)
    - Milliseconds to access
  - Amazon Glacier ($0.007 per GB)
    - 3-5 hours to access (higher access costs)
- Lifecycle policies (migrate older data automatically)

# Amazon S3 - Reliability

- Reliability of Standard storage class:
  - Service Level Agreement
  - Design: 99.999999999% durability and 99.99% availability of objects over a given year
  - Design: Sustain the concurrent loss of data in two facilities
  - Periodic consistency checks
- Data stored on multiple devices and in multiple facilities in the **same region**
  - Extra $$ option: Automatic Cross-Region Replication

# Amazon S3 - Buckets

- No such thing as “folders” in S3, only **buckets**
- Buckets
  - Name must be globally unique across all users
    - Your code should be intelligent to respect name conflicts
  - Cannot be renamed, only deleted/created anew
  - Cannot be nested inside another bucket
    - “Fake folders” with a prefix (e.g. “folder1/”), but the structure is flat to Amazon internally
  - Buckets are created inside a specific region (should be consistent with your computation)
  - Names should be in “DNS Format”
    - “my.aws.bucket” is OK, but not “mybucket.” or “.mybucket”

# Amazon S3 - Objects

- Objects cannot be modified once created
  - (Cannot modify bytes 100-245, but you can upload an entirely new file)
- Limits
  - Maximum object size: 5TB
  - Maximum objects in bucket: unlimited
- Data consistency
  - Read-after-write
    - PUT for new objects
  - Eventual consistency
    - DELETE, PUT for overwritten (modified) objects

# Other Storage Options

- **What about legacy applications?**
- **EC2 local storage**
  - Most nodes *(aside from cheapest/smallest)* have either local disk(s) or local SSD(s)
  - Raw disk – you format with your filesystem
  - Private disk (unless you choose to export it via the network)
  - Free (you already pay EC2 for the virtual machine)
  - Warning: Local storage is lost if you stop paying for the node!

# Other Storage Options

- **What about legacy applications?**
- Amazon **Elastic Block Storage (EBS)**
  - Virtual network disk instead of physical disk in EC2 node
    - Pro: Will persist after node is shut down!
    - Con: Slower? (must traverse network)
  - Raw disk – you format with your filesystem
  - Pay per GB (must "provision" in advance) + IOPS
  - SSD or Hard drive
  - **Not sharable!** (unless you use a fancy filesystem that allows one disk to be concurrently accessed by multiple computers)

# Other Storage Options

- **What about legacy applications?**
- Amazon **Elastic File System** (EFS)
  - Looks like convention enterprise NFS storage
  - SSD based, "petabyte scale"
  - **Sharable** - Multiple EC2 nodes can access same EFS drive
    - NFS server (from Amazon) software coordinates/ synchronizes between multiple clients

# Databases

# Database Options

1. Amazon **DynamoDB**
2. Amazon **ElastiCache**
3. Amazon **Relational Database Service** (RDS)

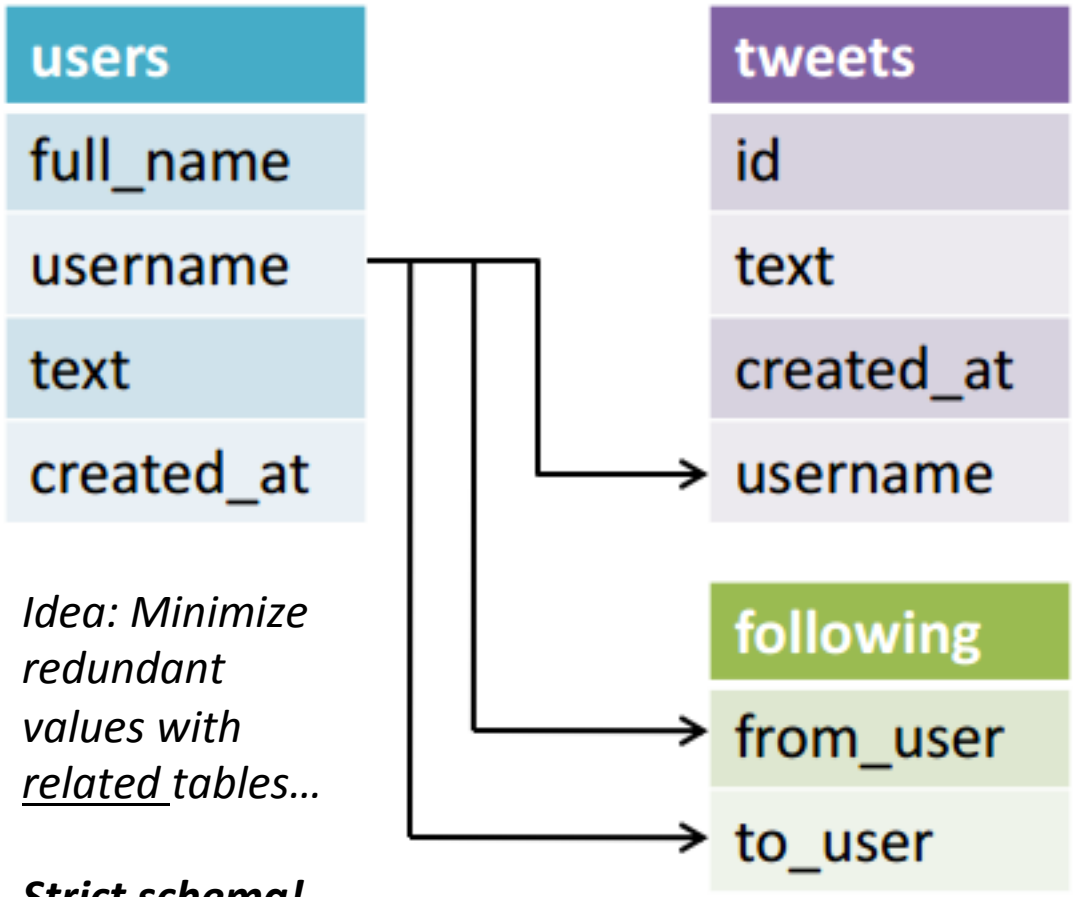
# Amazon DynamoDB

- Marketing: “a fast and flexible **NoSQL** database service for all applications that need consistent, single-digit millisecond latency at any scale.”
- Fully managed cloud database + All SSDs
- Automatic replication across 3 availability zones
- Data models: document and key-value store
- Pricing model: Throughput, not raw capacity
  - Write operations per hour
  - Read operations per hour / per consistency level

# DynamoDB – NoSQL?

- What is a **NoSQL** database?
- What is a **SQL** ("relational") database?

# Relational Database




*Idea: Minimize redundant values with related tables…*

***Strict schema!***

```
INSERT INTO users
(full_name, username)
VALUES ("Jeff Shafer",
"shafer");
```

```
UPDATE users SET
full_name="J Shafer"
WHERE
username="shafer";
```

```
SELECT full_name,
text, created_at FROM
users, tweets WHERE
users.username =
tweets.username AND
username="shafer";
```

# Relational Database Challenges

- Relational databases **scale up**
  - Buy a bigger, more expensive server
  - Lots of CPU cores
  - Lots of RAM
  - Lots of SSDs
- But you still have **one server**
  - Hot backup? (only one used at a time)
  - Read replicas? (helpful if writes are infrequent)
- Internet systems **scale out** to multiple servers

# NoSQL Inspiration

- It would be easier to scale-out a database if SQL wasn't so **complicated**
- Do we really need all these features?
- Do we really need all these data consistency guarantees?
- What set of features do web apps really need?

| | Relational Database | NoSQL Database |
|---|---|---|
| **Data model** | Strict table schema (rows, columns) | No schema ("hash key" / index accesses semi-structured data |
| **ACID** | **A**tomicity (all or nothing)<br>**C**onsistency (meets schema)<br>**I**solation (separate transactions)<br>**D**urability (recover to last known state) | Relaxed ACID compliance (tradeoffs!) |
| **Performance** | Disk / SSD dependent | Cluster size / network speed dependent |
| **Scale** | Scale "up" with faster hardware | Scale "out" with distributed cluster / low-cost hardware |
| **API** | Structured Query Language (SQL) | Object-based API |

# DynamoDB is NoSQL

- No joins of data between tables
  - Very difficult to implement joins in a scalable manner
- Programmers must use multiple queries
- Programmer can arrange data differently so joins are never needed
  - E.g *the record for your blog post has all the comments embedded in it, so only one query is needed*

# DynamoDB is NoSQL

- No strict schema for a table
  - Each record must have a "primary key" (which is indexed)
  - Different records can have different *attributes*
  - Flexibility!
- Tradeoff: You can only search on **indexed** attributes
- What if I want to search on something other than the primary key?
  - "Secondary index" feature
  - Limit of 5 indexes per table
  - Indexes cost IOPS to maintain / update ($$)
  - Designers must plan ahead!

# DynamoDB Operations

- GET
  - Retrieve data by primary key
- PUT
  - Insert new data by primary key
- UPDATE
  - Modify existing data by primary key
- DELETE
  - Remove existing data by primary key

# DynamoDB Search

- Query
  - Search table using only primary key attribute values
  - Very fast!
- Scan
  - Search table by examining every item in table
  - Slower to very slow! (depends on table size)
  - Might consume most of your provisioned read IOPS, starving application

# DynamoDB Consistency

- Application programmer decides (**per-request**)
- **Eventually Consistent Reads** (Default)
  - An eventually consistent read might not reflect the results of a recently completed write. Consistency across all copies of data is usually reached within a second. Repeating a read after a short time should return the updated data
  - **Faster!**
- **Strongly Consistent Reads**
  - A strongly consistent read returns a result that reflects all writes *that received a successful response* prior to the read
  - **Slower!**

# Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati,
Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall
and Werner Vogels

Amazon.com

## ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

"Dynamo: Amazon's Highly Available Key-value Store"
by G. DeCandia et. al. (SOSP 2007)

# Amazon ElastiCache

- In-memory (**RAM**) key-value store for small chunks of arbitrary data (strings, objects) from results of database calls, API calls, page rendering, …, …, …
- Choice of two popular open-source implementations *(not proprietary, for a change!)*
  - **Memcached** - http://www.memcached.org/
  - **Redis** - http://redis.io/
- These are **clustered** caching systems and provide automatic detection and recovery from node failures (plus scalability!)

# In-Memory Cache

- Goal: Reduce customer latency (lag)
  - Perhaps the database is a bottleneck?
  - Do intensive queries once and save the result for reuse (helps with read-heavy workloads)
- Caching is effective if
  - Data is slow or expensive to acquire when compared to cache retrieval
  - Data is accessed with sufficient frequency
  - Data is relatively static (or if rapidly changing, staleness is not a significant issue)

# Not "Just a Cache"

- Common web feature: Box on home page:
  - "Most recent 10 posts from users"
  - "Show All" link to see all posts from newest to oldest (paginated, 20 per page)

```
SELECT * FROM foo WHERE ...
ORDER BY time DESC LIMIT 10
```

- Slow! Have to query database for every single page load

# In-Memory Cache Example

1. User submits post
    1. Full data is sent to database, **and**
    2. Snippet of data is sent to in-memory cache (post ID #, text snippet?)
2. In-memory cache configured to only hold *n* most recent entries
3. Homepage loaded
    1. In-memory cache queried (faster!)
    2. Database *only* queried if cache is empty or user selects "View All" and exceeds data stored in cache

# Other Database Options

- **What about legacy applications?**
- Amazon **Relational Database Service** (RDS)
  - Choice of database engine
    - Amazon Aurora
    - Microsoft SQL Server
    - MySQL
    - Oracle
    - PostgreSQL
    - MariaDB
  - Database engine updated by Amazon
  - Multi-availability zone instances
    - Synchronous replication with hot standby in different zone
  - Scalability? ("classic", not "cloud")
    - Might need a very large ($$$) node
    - Might need read replicas
      - Hope your workload isn't write-heavy