Cloud Computing - CLD

Leonard Cseres | June 3, 2025

Google App Engine

Scaling Types

- **Manual Scaling**: Manually specify number of instances, stateful allowed, 24h timeout
- **Basic Scaling**: Auto scale to/from zero, stateless, 24h timeout
- **Automatic Scaling**: Predictive algorithms, stateless, 10min timeout

Automatic Scaling Metrics The automatic scaling algorithm uses these metrics to make scaling decisions:

- **CPU utilization** of instances
- **Request queue latency** - time HTTP requests wait before being processed
- **Concurrent requests** being handled simultaneously by instances
- *Note: Memory consumption is NOT used for scaling decisions*

```
automatic_scaling:
  target_cpu_utilization: 0.65 # Default 0.6
  min_instances: 5 # Avoid cold start
  max_instances: 100 # Control cost
  min_pending_latency: 30ms # Default 500ms
  max_pending_latency: automatic # Default 10s
  max_concurrent_requests: 50 # Default 10
```

- **CPU utilization**: 70% -> start new instance, 0% for 15min -> shutdown
- **Queue latency**: 500ms wait time -> start new instance
- **Concurrent requests**: 6 parallel requests -> start new instance

Cold Start Delay when a new application instance must be started to handle a request

- Occurs when scaling up from zero instances or when traffic exceeds current capacity
- Can be minimized by setting `min_instances > 0` in automatic scaling

Scale to Zero Reducing instances to zero when not in use to reduce costs

- Automatic scaling can scale down to 0 instances when idle
- Optional feature - disable by setting `min_instances > 0`
- Saves costs but introduces cold start delay for first request

Java Application Deployment

- Use Maven with `pom.xml` to declare dependencies
- Google App Engine provides and updates the JDK
- Developer must provide and update the web application server
- Two build processes: local (testing) + cloud (production binaries)
- No need to create VM images manually for deployments

Request Handler Lifecycle

1. Request arrives
2. Handler created, receives request
3. Handler creates response (stateless!)
4. Response sent, handler removed from memory

Pricing (us-central1) Free limits per day: 28 instance hours, 1 GB data transfer out, 1 GiB datastore storage

---

Google Datastore

Data Model

- **Entity**: Collection of key-value pairs where each key is unique within the entity
  - Identified by a unique key in the database
  - Like a row in a relational database but schema-less
- **Kind**: Type of entity (like table name)
- **Property**: Key-value data within entity
- **Key**: Application ID + Kind + Entity ID + (optional) Ancestor path

Low-level API

```
KeyFactory keyFactory = datastore.newKeyFactory().setKind("book");
Key key = datastore.allocateId(keyFactory.newKey());
Entity entity = Entity.newBuilder(key)
    .set("title", "The grapes of wrath")
    .build();
datastore.put(entity);
```

JPA High-level API

```
@Entity(name = "Book")
public class Book {...}
EntityManager em = emf.createEntityManager();
em.persist(new Book());
```

Vertical Database Scaling More powerful hardware, easy to implement in cloud, costs increase more than linearly

Horizontal Database Scaling Distribute across machines, replication, partitioning/sharding

Single-Leader Replication

- One leader accepts writes, followers accept reads
- **Synchronous**: Leader waits for follower acknowledgment (slower, safer)
- **Asynchronous**: Leader doesn't wait (faster, potential data loss)
- **Key characteristics**:
  - All write operations go to leader, then replicated to followers
  - System continues working even if a follower fails
  - Leader and followers can be in different data centers
  - Reads can be distributed across followers (not just leader)

Partitioning/Sharding

Sharding Calculation Example For NoSQL databases with horizontal scaling:

- **Scenario**: Need 2.5 TB storage, each server stores 1 TB
- **Solution**: 3 servers with sharding (3 TB total capacity > 2.5 TB needed)
- **Mechanism**: Hash function distributes data across servers
- **Performance**:
  - Best case: Requests distributed evenly → 3000 reads/sec (3×1000)
  - Worst case: All requests to same server → 1000 reads/sec

Consistent Hashing

- Keys and machines mapped to circle using hash function
- Object assigned to next machine clockwise
- Adding/removing machines only affects adjacent objects
- Minimizes data movement during scaling

```
position = hash(key) mod number_of_machines
```

Problems with simple hashing: Adding machine changes almost all object positions

---

NoSQL Data Models

Key-Value

- Simple hashmap: key -> value
- **Data opacity**: Record information is opaque to the database
- Operations: get(key), put(key, value), delete(key)

Wide Column/Column-Family

- Row key + Column families
- **Structured data**: Record information structured as key-value pairs
- Each column family contains key-value pairs

```
Row key: 071943
+-- profile: name="Martin", age=30
+-- billing: address="...", payment="..."
+-- orders: OR1001="data", OR1002="data"
```

Document

- **Hierarchical structure**: Record information organized hierarchically
- JSON-like nested documents
- Query over nested data

```
{
  "id": "1001",
  "customer_id": "7231",
  "line_items": [
    { "product_id": "4555", "quantity": 8 },
    { "product_id": "7655", "quantity": 4 }
  ]
}
```

Graph

- **Explicit relationships**: Model includes both records AND relationships between them
- Vertices and edges
- Query language for relationships (e.g., Cypher)

```
START barbara = node:nodeIndex(name = "Barbara")
MATCH (barbara)-[:FRIEND]->(friend_node)
RETURN friend_node.name, friend_node.location
```

Cloud Databases

Single-Tenant

- Dedicated VM per client
- Usually SQL
- Scheduled maintenance
- Instance time + storage capacity
- Examples: AWS RDS, Azure SQL

Multi-Tenant

- Shared cluster
- Usually NoSQL
- No downtime
- Data volume stored
- Examples: DynamoDB, CosmosDB

Container Orchestration - Kubernetes

- **Cluster**: Set of machines running Kubernetes
- **Node**: Machine in cluster (master/worker)
- **Pod**: Smallest deployable unit, 1+ containers
- **Service**: Stable network endpoint for pods
- **Deployment**: Manages pod replicas and updates

Pod Lifecycle

```
Pending -> Running -> Succeeded/Failed/Unknown
```

- Pods are cattle, not pets (disposable)
- New pod gets new ID and IP when replaced

Pod Characteristics

- **Co-location**: All containers in a Pod are always placed on the same node
- **Networking**: Pod can communicate with any other Pod in cluster (not limited to same node)
- **IP addresses**: Pods get new IP when recreated (no persistence without special configuration)
- **Container relationship**: One Pod can contain multiple containers (not the reverse)

YAML Structure

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
  labels:
    component: redis
    app: todo
spec:
  containers:
    - name: redis
      image: redis
      ports:
        - containerPort: 6379
      resources:
        limits:
          cpu: 100m
      env:
        - name: REDIS_ENDPOINT
          value: redis-svc
```

Service Types

- **ClusterIP**: Internal cluster IP only (default)
- **NodePort**: Expose on each node's IP at static port
- **LoadBalancer**: Cloud provider load balancer

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - port: 80
      targetPort: 8080
  type: LoadBalancer
```

Deployments

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
```

Deployment Capabilities

- **Zero-downtime updates**: Update application code without service interruption
- **Template specification**: `template:` section contains Pod specification
- **Health monitoring**: Continuously ensures specified number of healthy Pods
- **Rolling updates**: Gradually replace old pods with new ones
- **Independent of Services**: Service creation doesn't require existing Deployment

Rolling Updates

- Update deployment -> new ReplicaSet created
- Gradually replace old pods with new ones
- Zero downtime deployment

```
kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1
kubectl rollout status deployment/nginx-deployment
```

Persistent Volumes

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 8Gi
  storageClassName: default
```

Volume Types

- **emptyDir**: Temporary, deleted with pod
- **hostPath**: Host machine path
- **secret**: Sensitive data (passwords, keys)
- **persistentVolumeClaim**: Persistent storage
- **nfs**: Network file system
- **awsElasticBlockStore**: AWS EBS
- **gcePersistentDisk**: Google Persistent Disk

Cluster Architecture Master Node (Control Plane):

- **etcd**: Key-value store for cluster state
- **API Server**: REST API for cluster management
- **Scheduler**: Assigns pods to nodes
- **Controller Manager**: Runs control loops

Worker Nodes:

- **kubelet**: Node agent managing containers
- **kube-proxy**: Network proxy and load balancer
- **Container Runtime**: Docker/containerd

Common Commands

```
# Scaling
kubectl scale deployment <name> --replicas=5
# Updates
kubectl set image deployment/<name> <container>=<image>
kubectl rollout undo deployment/<name>
```

Networking

- **Flat network**: All pods can communicate
- **Overlay network**: Software-defined networking
- **Service discovery**: DNS names for services
- **Load balancing**: Services distribute traffic

Popular overlay networks: Flannel, Calico, Weave

Infrastructure as Code

Imperative Approach (AWS CLI) Specify step-by-step commands to achieve desired state

Declarative Approach (Terraform) Define desired end state, tool figures out how to achieve it

Terraform

Capabilities

- **Infrastructure management**: Create, modify, destroy cloud resources
- **State tracking**: Maintains state of infrastructure
- **Cross-platform**: Works with multiple cloud providers

What Terraform Can Do

- Change EC2 instance types
- Attach EBS volumes to EC2 instances
- Manage VPCs, subnets, security groups
- Configure load balancers

What Terraform Cannot Do

- **OS-level operations**: Cannot create user accounts on instances
- **Application management**: Cannot start/stop applications within instances
- **Runtime configuration**: Cannot manage running services inside VMs

Ansible

Architecture

- **Agentless**: No software installation on managed machines
- **Push-based**: Control machine pushes configurations
- **Idempotent**: Safe to run multiple times

Execution Model Tasks execute in parallel across hosts, but sequentially per host:

```
Parallel: host1.task1, host2.task1, host3.task1
Then: host1.task2, host2.task2, host3.task2
Then: host1.task3, host2.task3, host3.task3
```

Core Modules

- **apt**: Install software packages on managed machines
- **service**: Start/stop background services
- **copy**: Transfer files from control to managed machines
- **template**: Transfer files with variable substitution

Example Playbook

```
- name: webserver setup
  hosts: webservers
  become: True
  tasks:
    - name: Install nginx
      apt: name=nginx
    - name: Install mysql
      apt: name=mysql-server
    - name: Install apache
      apt: name=apache2
```