RELIABILITY IN MODERN CLOUD SYSTEMS

Summer 2025

LOGISTICS

ASSIGNMENT 1

- Assignment 1 has been released
- Due: Saturday 10th May, 2025 5pm CEST.
- Each student registered on CMS has their own private fork of the assignments repo.
  - If you do not have access to your repo then contact the course staff immediately after class.
  - We will only grade the officially created forks of the main assignments repo.
  - Remember to pull test fixes from the main repository.

TAIL AT SCALE DISCUSSION

PAPER SUMMARY

PAPER SUMMARY

Cloud Computing landscape has changed

- Datacentres are heterogeneous due to ending of Moore's Law
- New workloads/applications have fundamentally different performance requirements

PAPER SUMMARY

Cloud Computing landscape has changed

- Datacentres are heterogeneous due to ending of Moore's Law
- New workloads/applications have fundamentally different performance requirements

Guaranteeing tail latency in current landscape is hard

- Different hardwares have different performance profiles
- Increase in latency variability!

DISCUSSION THEMES

- What software techniques do we need to handle performance variability?
- Why can we not eliminate all performance variability?
- What is the impact of new workloads and hardware on reliability?

DISCUSSION THEMES

- Why can we not eliminate all performance variability?

DISCUSSION THEMES

- Why can we not eliminate all performance variability?

Systems will get more complex
Complex systems have more variability

DISCUSSION THEMES

- What is the impact of new workloads and hardware on reliability?

DISCUSSION THEMES

- What is the impact of new workloads and hardware on reliability?

Harder to provide reliability guarantees

DISCUSSION THEMES

- What software techniques do we need to handle performance variability?

DISCUSSION THEMES

- What software techniques do we need to handle performance variability?

Techniques that reduce the tail latency! Eg: request hedging

RELIABILITY BASICS

THE TAIL AT SCALE

Reliability of cloud systems at scale is largely dependent on the tail performance of the system at scale

Software techniques that tolerate performance variability are vital to building responsive cloud systems at scale

THE TAIL AT SCALE

Reliability of cloud systems at scale is largely dependent on the tail performance of the system at scale

Software techniques that tolerate performance variability are vital to building responsive cloud systems at scale

WHAT IS RELIABILITY?

Reliability is the ability of cloud services to consistently perform as expected with minimal disruption

HOW TO SPECIFY RELIABILITY?

Reliability is the ability of cloud services to consistently perform as expected with minimal disruption

SERVICE LEVEL OBJECTIVES

Reliability is the ability of cloud services to consistently perform as expected with minimal disruption

Specify reliability of a system using service level objectives!

- Service Level Objectives are measurable targets that define the expected level of a service's reliability behaviour

MEASURING SLO

Reliability is the ability of cloud services to consistently perform as expected with minimal disruption

Specify reliability of a system using service level objectives!

- Service Level Objectives are measurable targets that define the expected level of a service's reliability behaviour

SLOs are measured using a Service Level Indicator (SLI)

MEASURING SLO

Reliability is the ability of cloud services to consistently perform as expected with minimal disruption

Specify reliability of a system using service level objectives!

- Service Level Objectives are measurable targets that define the expected level of a service's reliability behaviour

SLOs are measured using a Service Level Indicator (SLI)

```
func IsSLOAchieved(measurement SLI) {
  return measurement <= upper_bound && measurement >= lower_bound
}
```

MEASURING SLO

Reliability is the ability of cloud services to consistently perform as expected with minimal disruption

Specify reliability of a system using service level objectives!

The measurement is usually an aggregation of a distribution OR a percentage of a total

SLOs are measured using a Service Level Indicator (SLI)

```
func IsSLOAchieved(measurement SLI) {
  return measurement <= upper_bound && measurement >= lower_bound
}
```

RESPONSIVENESS

Service Level Indicator:

RESPONSIVENESS

Service Level Indicator: Request Latency (99th percentile)

RESPONSIVENESS

Service Level Indicator: Request Latency (99th percentile)

Service Level Objective:

RESPONSIVENESS

Service Level Indicator: Request Latency (99th percentile)

Service Level Objective:

```
func Isp99Achieved(p99latency uint64) {
  upper_bound = 1e9 // 1 second in nanoseconds
  return latency <= upper_bound
}
```

RESPONSIVENESS

Service Level Indicator: Request Latency (99th percentile)

Service Level Objective:

```
func Isp99Achieved(p99latency uint64) {
  upper_bound = 1e9 // 1 second in nanoseconds
  return latency <= upper_bound
}
```

99% of requests must have latency is lower than 1 second

AVAILABILITY - UPTIME

Service Level Indicator:

AVAILABILITY - UPTIME

Service Level Indicator: Uptime (How long has the service been running?)

AVAILABILITY - UPTIME

Service Level Indicator: Uptime (How long has the service been running?)

Service Level Objective:

AVAILABILITY - UPTIME

Service Level Indicator: Uptime (How long has the service been running?)

Availability %	Downtime per year[note 1]	Downtime per quarter	Downtime per month	Downtime per week	Downtime per day (24 hours)
99.999% ("five nines")	5.26 minutes	1.31 minutes	26.30 seconds	6.05 seconds	864.00 milliseconds
99.9999% ("six nines")	31.56 seconds	7.89 seconds	2.63 seconds	604.80 milliseconds	86.40 milliseconds
99.99999% ("seven nines")	3.16 seconds	0.79 seconds	262.98 milliseconds	60.48 milliseconds	8.64 milliseconds
99.999999% ("eight nines")	315.58 milliseconds	78.89 milliseconds	26.30 milliseconds	6.05 milliseconds	864.00 microseconds
99.9999999% ("nine nines")	31.56 milliseconds	7.89 milliseconds	2.63 milliseconds	604.80 microseconds	86.40 microseconds

AVAILABILITY - GOODPUT

Service Level Indicator: Goodput (# successful reqs / # total reqs)

AVAILABILITY - GOODPUT

Service Level Indicator: Goodput (# successful reqs / # total reqs)

Service Level Objective:

```
func IsAvailabilityAchieved(goodput float64) {
  lower_bound = 0.95
  return goodput >= lower_bound
}
```

AVAILABILITY - GOODPUT

Service Level Indicator: Goodput (# successful reqs / # total reqs)

Service Level Objective:

```
func IsAvailabilityAchieved(goodput float64) {
  lower_bound = 0.95
  return goodput >= lower_bound
}
```

95% of all requests should succeed

RESOURCE COST

Service Level Indicator: Resource Utilization

RESOURCE COST

Service Level Indicator: Resource Utilization

Service Level Objective:

```
func IsUtilizationAchieved(utilization float64) {
  lower_bound = 0.75
  return goodput >= lower_bound
}
```

At least 75% of the allocated resource must not be idle

RESILIENCE

Service Level Indicator:

RESILIENCE - FAILURES

Service Level Indicator:

- MTBF: Mean Time Between Failures
  - Goal is to maximize the MTBF
- Avg. # of incidents in a given time window
  - Goal is to minimize the number of incidents in the time window

RESILIENCE - RECOVERY

Service Level Indicator:

- MTTR: Mean Time to Recovery
  - Goal is to minimize the MTTR

RESILIENCE

Failures SLIs:

- MTBF: Mean Time Between Failures
  - Goal is to maximize the MTBF
- Avg. # of incidents in a given time window
  - Goal is to minimize the number of incidents in the time window

Recovery SLIs:

- MTTR: Mean Time to Recovery
  - Goal is to minimize the MTTR

DURABILITY

Primarily defined for stateful services like databases, caches

STATELESS VS STATEFUL

Stateless

- Does not track app state
- Operations are idempotent
- Maintains very little to no “state”

Stateful

- Tracks app state
- All Operations are not idempotent
- Maintains all the state!

DURABILITY

Primarily defined for stateful services like databases, caches

Service Level Indicator: % of written that is readable

Service Level Objective: Ideally, 100% of the written data is readable

CORRECTNESS & QUALITY

CORRECTNESS & QUALITY

Definition of quality is application specific

- Example SLO: At least 10% of the ads shown in Google Search results must result in an ad click

CORRECTNESS & QUALITY

Definition of quality is application specific

- Example SLO: At least 10% of the ads shown in Google Search results must result in an ad click

Correctness is measured using tests prior to deployment

During execution, you can track correctness violations

- Example SLO: Less than 5% read requests must violate causal consistency

THE TAIL AT SCALE

Reliability of cloud systems at scale is largely dependent on the tail performance of the system at scale

Software techniques that tolerate performance variability are vital to building responsive cloud systems at scale

THE TAIL AT SCALE

Reliability of cloud systems at scale is largely dependent on the tail performance of the system at scale

Software techniques that tolerate performance variability are vital to building responsive cloud systems at scale

VERTICAL SCALING

- Provide more computation resources to the service

VERTICAL SCALING

- Provide more computation resources to the service
  - Assign more CPUs to the container executing the service
  - Migrate the service to a more powerful machine

VERTICAL SCALING

- Provide more computation resources to the service
  - Assign more CPUs to the container executing the service
  - Migrate the service to a more powerful machine

If all fails, then just download more RAM!

VERTICAL SCALING

Provide more computation resources to the service

Reliability Analysis

- Limited improvement to throughput + goodput
- Marginal difference in latency
- No improvement in uptime!

DATA SHARDING

DATA SHARDING

Only applicable to stateful services

How to have data sharding?

DATA SHARDING

Only applicable to stateful services

How to have data sharding?

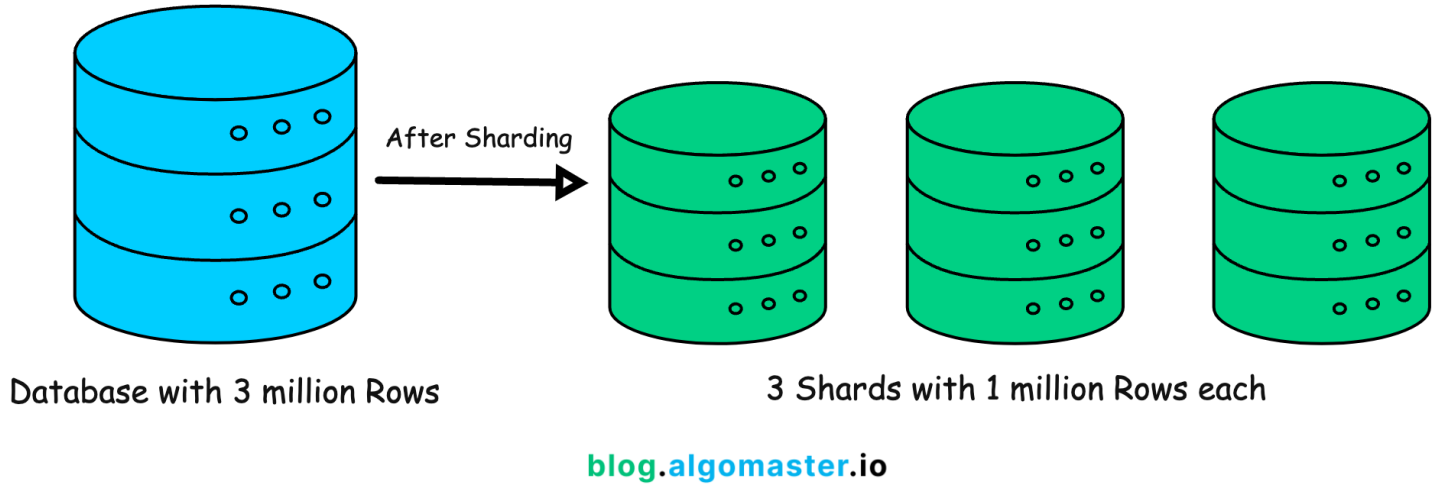
+ Better throughput

+ Slightly better latency

+ Lower utilization

+ No single pt of failure

+ No redundancy



REPLICATION (HORIZONTAL SCALING)

REPLICATION (HORIZONTAL SCALING)

Stateless Services

- Add more instances of the service
- Put a load balancer in front to direct the requests
- Improves throughput
- Worsens latency (we now have an extra hop)

REPLICATION (HORIZONTAL SCALING)

Stateless Services

- Add more instances of the service
- Put a load balancer in front to direct the requests
- Improves throughput
- Worsens latency (we now have an extra hop)

Stateful Services

- Improves read throughput; Worsens write throughput

MULTI-TENANCY

Problem: We have limited capacity, need to maximize the efficiency of the hardware resources (specifically compute)

MULTI-TENANCY

Problem: We have limited capacity, need to maximize the efficiency of the hardware resources (specifically compute)

Solution:

- Co-locate services so that we pack as many services on a machine as possible
- Improves resource efficiency and utilization
- Horrible for individual service tput + latency

REQUEST HEDGING

Problem: Lot of variability in latency, latency distribution has long tails in worst case scenarios

Solution:

- Send the same request to multiple (say 2) replicas
- Use the result from the faster replica
- Do n-times the amount of work to improve latency

TIMEOUTS

Problem: Have to wait a long time to get any response from an overloaded server

TIMEOUTS

Problem: Have to wait a long time to get any response from an overloaded server

Solution:

- Make the request and wait for a specified time period (eg: 1s)
- If response doesn't come in wait period then abort the execution

RETRIES

Problem: Did not receive a timely or successful response from the server

RETRIES

Problem: Did not receive a timely or successful response from the server

Solution

- If request fails or times out, try again
- ...but try again smartly so that we don't overload the system more

RETRIES

Problem: Did not receive a timely or successful response from the server

Solution

- If request fails or times out, try again
- …but try again smartly so that we don't overload the system more

CANCELLATIONS

Problem: Initial request may have timed out but downstream services still keep doing the work

CANCELLATIONS

Problem: Initial request may have timed out but downstream services still keep doing the work

Solution:

- If a request is timed-out or cancelled higher up in the call chain, propagate the cancellation to all downstream services
- Saves wasted work
- Cancellation propagation takes extra calls

DEADLINES

Problem: Cancellation propagation does not prevent the problem of wasted effort

DEADLINES

Problem: Cancellation propagation does not prevent the problem of wasted effort

Solution:

- For each request, set a deadline that it must finish by
- Propagate deadline with request to each server
- Cancel requests whenever deadline is set

CANARY REQUESTS

Problem: New never-seen-before incoming requests could cause issues with the downstream services

CANARY REQUESTS

Problem: New never-seen-before incoming requests could cause issues with the downstream services

Solution:

- Send the request to a subset of servers
- Wait for successful response from the subset
- Abort if not reached the minimum threshold of servers

CANARY RELEASES

Problem: New releases might be buggy; can bring the system down

CANARY RELEASES

Problem: New releases might be buggy; can bring the system down

Solution:

Roll-out new releases slowly

Instead of updating all instances of a service at once, update incrementally

May have stale instances and new instances operating together

DISCUSSION THEMES

- When to retry and how to retry?
- How should a load balancer balance requests across the replica group?
- Cancellations and Deadlines try to reduce wasted work. Which technique should a system employ?
- Are retries good?