# Amazon Builder Library Notes

Diego Pacheco

## About me...

- ❑ Cat's Father
- ❑ Principal Software Architect
- ❑ Agile Coach
- ❑ SOA/Microservices Expert
- ❑ DevOps Practitioner
- ❑ Speaker
- ❑ Author

diegopacheco

@diego_pacheco

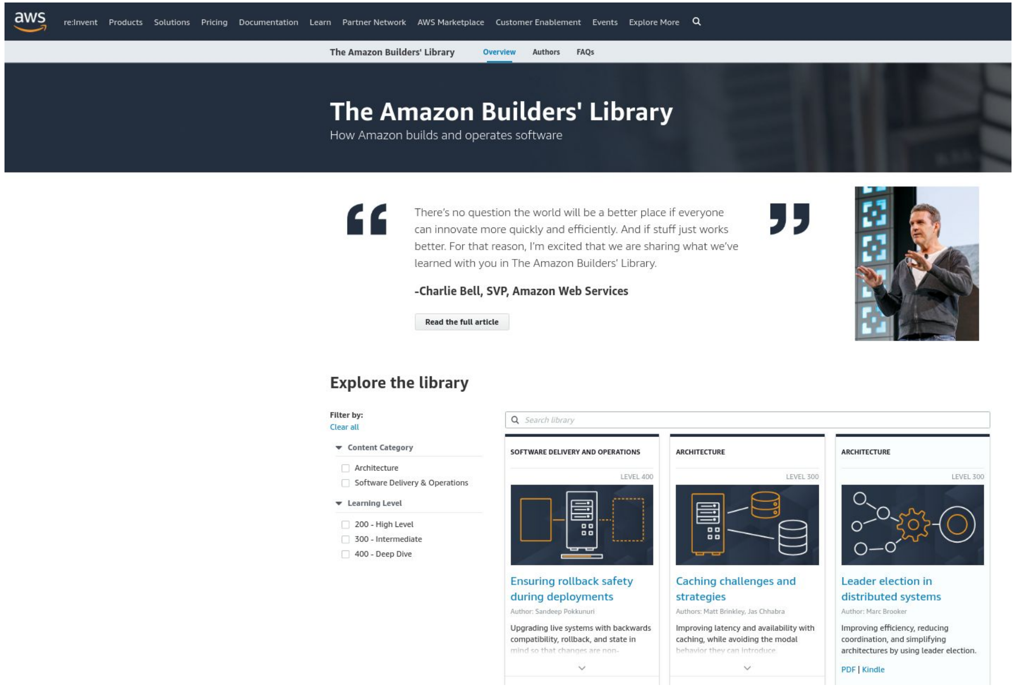
http://diego-pacheco.blogspot.com.br/

Diego Pacheco
Building Applications with Scala
Write modern, scalable, and reactive applications with the power of Scala
Packt

Diego Pacheco
Building Effective Microservices
Explore microservices and their implementation hands-on
Packt

https://diegopacheco.github.io/

# Amazon Builders Library


re:Invent Products Solutions Pricing Documentation Learn Partner Network AWS Marketplace Customer Enablement Events Explore More

The Amazon Builders' Library Overview Authors FAQs

**The Amazon Builders' Library**

How Amazon builds and operates software

There's no question the world will be a better place if everyone can innovate more quickly and efficiently. And if stuff just works better. For that reason, I'm excited that we are sharing what we've learned with you in The Amazon Builders' Library.

**-Charlie Bell, SVP, Amazon Web Services**

Read the full article

**Explore the library**

Filter by:
Clear all

Content Category
- Architecture
- Software Delivery & Operations

Learning Level
- 200 - High Level
- 300 - Intermediate
- 400 - Deep Dive

Search library

SOFTWARE DELIVERY AND OPERATIONS — LEVEL 400
Ensuring rollback safety during deployments
Author: Sandeep Pokkunuri
Upgrading live systems with backwards compatibility, rollback, and state in mind so that changes are non-

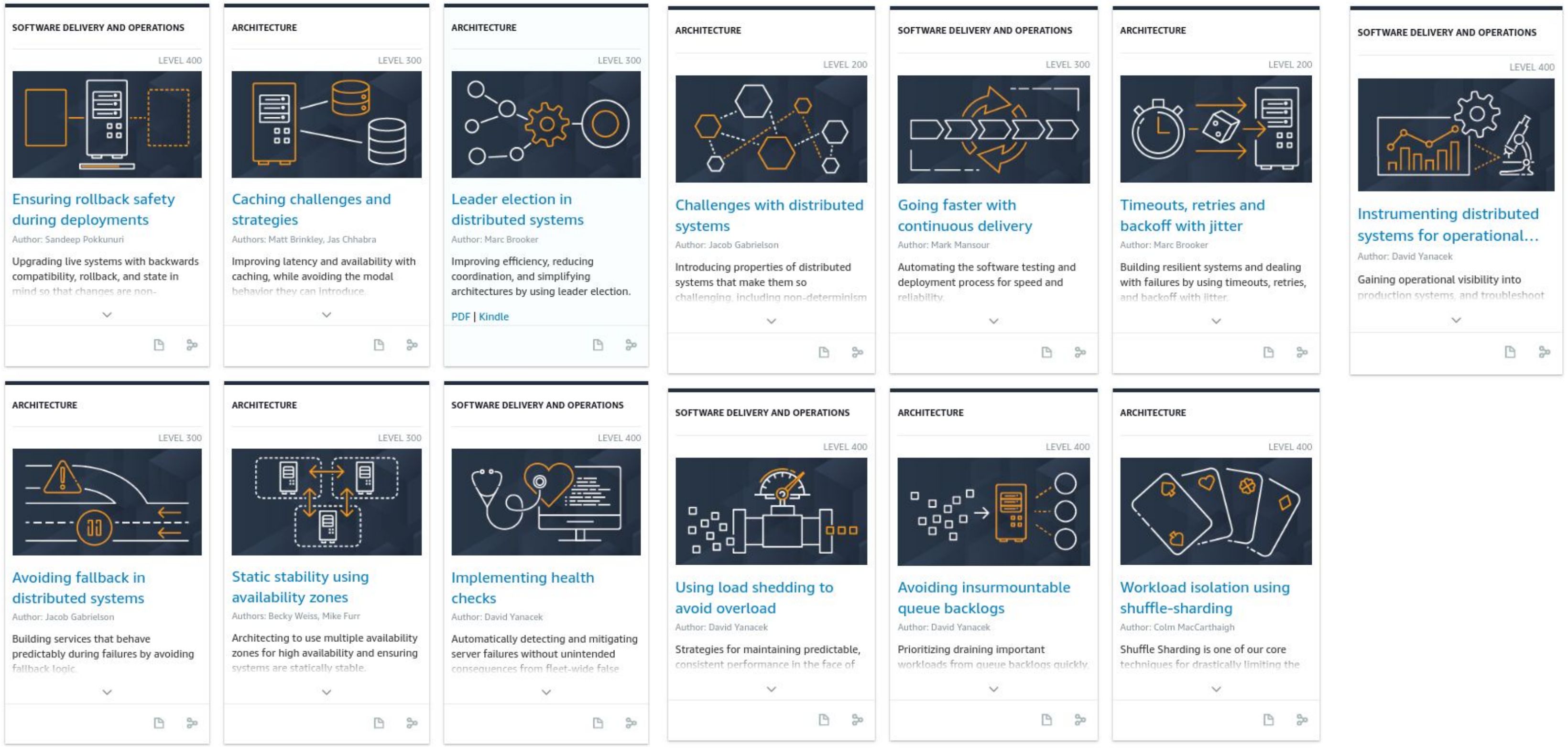
ARCHITECTURE — LEVEL 300
Caching challenges and strategies
Authors: Matt Brinkley, Jas Chhabra
Improving latency and availability with caching, while avoiding the modal behavior they can introduce

ARCHITECTURE — LEVEL 300
Leader election in distributed systems
Author: Marc Brooker
Improving efficiency, reducing coordination, and simplifying architectures by using leader election.
PDF | Kindle

- ☐ How the Build AWS
- ☐ Amazon Experience
- ☐ Theory
- ☐ Practice
- ☐ Real Cases
- ☐ Techniques and products
- ☐ Super interesting
- ☐ 13 Articles so far

https://aws.amazon.com/builders-library/

# The Library



SOFTWARE DELIVERY AND OPERATIONS
LEVEL 400
Ensuring rollback safety during deployments
Author: Sandeep Pokkunuri
Upgrading live systems with backwards compatibility, rollback, and state in mind so that changes are non-

ARCHITECTURE
LEVEL 300
Caching challenges and strategies
Authors: Matt Brinkley, Jas Chhabra
Improving latency and availability with caching, while avoiding the modal behavior they can introduce

ARCHITECTURE
LEVEL 300
Leader election in distributed systems
Author: Marc Brooker
Improving efficiency, reducing coordination, and simplifying architectures by using leader election.
PDF | Kindle

ARCHITECTURE
LEVEL 200
Challenges with distributed systems
Author: Jacob Gabrielson
Introducing properties of distributed systems that make them so challenging, including non-determinism

SOFTWARE DELIVERY AND OPERATIONS
LEVEL 300
Going faster with continuous delivery
Author: Mark Mansour
Automating the software testing and deployment process for speed and reliability

ARCHITECTURE
LEVEL 200
Timeouts, retries and backoff with jitter
Author: Marc Brooker
Building resilient systems and dealing with failures by using timeouts, retries, and backoff with jitter

SOFTWARE DELIVERY AND OPERATIONS
LEVEL 400
Instrumenting distributed systems for operational...
Author: David Yanacek
Gaining operational visibility into production systems, and troubleshoot

ARCHITECTURE
LEVEL 300
Avoiding fallback in distributed systems
Author: Jacob Gabrielson
Building services that behave predictably during failures by avoiding fallback logic

ARCHITECTURE
LEVEL 300
Static stability using availability zones
Authors: Becky Weiss, Mike Furr
Architecting to use multiple availability zones for high availability and ensuring systems are statically stable

SOFTWARE DELIVERY AND OPERATIONS
LEVEL 400
Implementing health checks
Author: David Yanacek
Automatically detecting and mitigating server failures without unintended consequences from fleet-wide false

SOFTWARE DELIVERY AND OPERATIONS
LEVEL 400
Using load shedding to avoid overload
Author: David Yanacek
Strategies for maintaining predictable, consistent performance in the face of

ARCHITECTURE
LEVEL 400
Avoiding insurmountable queue backlogs
Author: David Yanacek
Prioritizing draining important workloads from queue backlogs quickly

ARCHITECTURE
LEVEL 400
Workload isolation using shuffle-sharding
Author: Colm MacCarthaigh
Shuffle Sharding is one of our core techniques for drastically limiting the

## Avoid one way doors

The Importance of **Rollback**

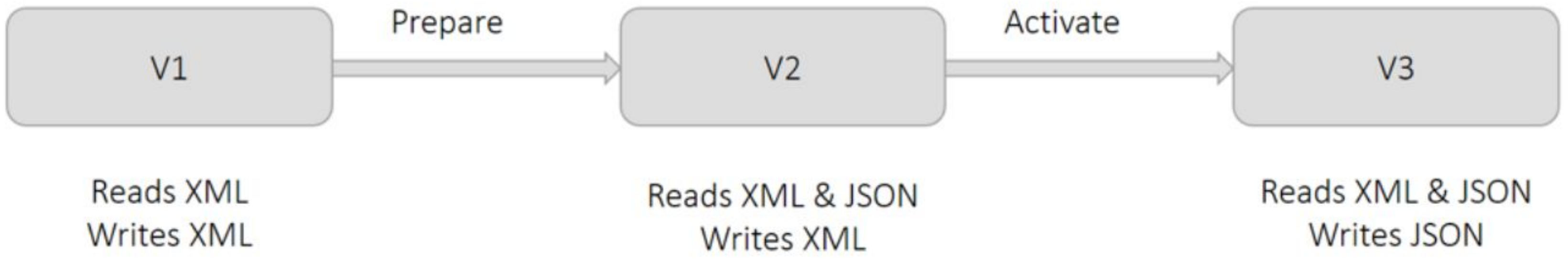
Type 1 decisions are not reversible, and you have to be very careful making them. (One way doors)

Type 2 decisions are like walking through a door — if you don't like the decision, you can always go back. (Two-Way Doors).

## Backward Compatibility

1. No Errors
2. No Service Disruption

# 2 Phase Deployment



https://aws.amazon.com/builders-library/ensuring-rollback-safety-during-deployments/?did=ba_card&trk=ba_card

# Local & External Caches

## Local cache

- ❑ Added on Demand
- ❑ No Ops Overhead
- ❑ In-memory HashTable

### Issues

- ❑ Downstream load proportional to fleet size
- ❑ Cache Coherence
- ❑ Cold Start

## External cache

- ❑ Memcached / Redis
- ❑ Reduce Cache Coherence issue
- ❑ No Cold Start issues
- ❑ Load of Downstream is reduced

### Issues

- ❑ More Complexity
- ❑ More Ops Overhead

# Inline & Side Caches

## Inline cache

- ❑ R/W Trought
- ❑ Embedded Cache mgmt
- ❑ Dax, Nginx, Varnish
- ❑ Uniform API model for clients
- ❑ Cache logic outside of the code (Eliminating potential bugs).

## Side cache

- ❑ ElastiCache(Redis/Memcached)
- ❑ Guava / EhCache
- ❑ Application controls the cache

# Cache Challenges

## Figure it out the right

- ❑ Cache Size
- ❑ Expiration Policy
- ❑ Eviction Policy

## Keep Eye on

- ❑ Cache HIT / Miss metrics

## Most Common Expiration policy

- ❑ Time-based: TTL

## Most Common Eviction policy

- ❑ LRU

## Amazon use 2 TTLS

- ❑ Soft: For updates
- ❑ Hard: For eviction

* Used in IAM

# Downstream fallback

## Be Careful

- ❑ Could spike traffic in downstream
- ❑ Could lead to:
  - ❑ Throttling
  - ❑ Burnout

## Better Options

- ❑ In case of External Cache outage:
  - ❑ Fallback to Local Cache
  - ❑ Use Load Shedding - reduce the number of requests going to downstream

# Thundering Herd Problem

ADOLPH ZUKOR AND JESSE L. LASKY PRESENT
ZANE GREY'S
THE THUNDERING HERD
WITH JACK HOLT, LOIS WILSON
NOAH BEERY, RAYMOND HATTON
SCREEN PLAY BY LUCIEN HUBBARD • DIRECTED BY WILLIAM HOWARD
A Paramount Picture
PRODUCED BY FAMOUS PLAYERS - LASKY CORP.
COUNTRY OF ORIGIN U.S.A.
THIS LOBBY DISPLAY LEASED FROM FAMOUS PLAYERS-LASKY CORPORATION

**<u>The Issue</u>**

- ❑ Many clients requesting the same key / data
- ❑ Uncached - so forces go to downstream.
- ❑ Empty Local cache (just joined the fleet)
- ❑ Situation could lead to:
  - ❑ Burnout
  - ❑ Throttling

**<u>The Solution</u>**

- ❑ Cache Coaleasing
- ❑ Varnish nginx have this feature
- ❑ Make sure just 1 request goto the downstream

# Leader Election (Single-Leader)

## Benefits

- ❑ Easier to Understand
- ❑ Works Simply
- ❑ Offers client consistency

## Downsides

- ❑ SPOF
- ❑ Single Point of Scaling
- ❑ Single point of truth (bad leader has high blast radius)
- ❑ Partial Deployments are hard to apply

# Leader Election Best Practices

Amazon does:

- ❑ Modeling systems with TLA+
- ❑ Check Remaining lease before side-effect ops outside of the leader
- ❑ Consider on the code: slow network, timeouts, retrys, gc pauses
- ❑ Avoid Heart Beating leases on background thread
- ❑ Make it easy to find the host who is current leader

# Avoinding Fallback

Issues

- ❏ Hard to Test
- ❏ Fallback could fail
- ❏ Fallback could make it worst
- ❏ Fallback could introduce latent bug

## Let it Crash

- ❑ Erland
- ❑ Akka
- ❑ ...now Amazon

# How Amazon Avoid Fallbacks

Do:

- ❏ Make non-fallback code more resilient
- ❏ Let the caller handle the failure
- ❏ Push Data Proactivity (IAM credential push data and its valid for several hours).
- ❏ Convert fallaback to failover
- ❏ Ensure retry/timeouts don't become fallback

# Static Stability

Amazon does dor Ec2:

- ❑ Control Plane vs Data Plane
- ❑ Control plane is more complex
- ❑ Data plane is more simple therefore more reliable
- ❑ AZs(Availability Zones) don't share:
  - ❑ Power
  - ❑ Infrastructure
- ❑ AZs are connected to each other fast fiber optical network

# Static Stability ~ EC2

## Control Plane

- ❑ Finds physical server
- ❑ Allocate network interface
- ❑ Generate EBS volume
- ❑ Install SG rules
- ❑ More Complex

## Data Plane

- ❑ Routes Packages to the VPC route table
- ❑ R/W from Amazon Volumes
- ❑ Much more simple than Control plane therefore more available
- ❑ Control Plane impairment:
  - ❑ Loose updates SGs
  - ❑ But machine keep working

# Static Stability Under the hood

<u>Ec2 Static Stability:</u>

- ❑ 2 Azs in same regions get deploys in different days
- ❑ Deploy first in one Box / Cell then 1/N Servers
- ❑ Align Ec2 deploy with AZ boundary ~ if deploy goes wrong affects only 1 AZ, them is rollback, fixed and deployed again.
- ❑ Packets flow stay under same AZ(avoid cross boundaries)
- ❑ Always provision capacity you don't need:
    - ❑ AZs are 50% overprovisioned
    - ❑ AZs operate at maximum 66% of the level which was load-tested

# Implementing Health Checkers

Types of Health Checkers:

- ❑ Liveness Health Checker: an I healthy?
- ❑ Local Health Checker:
  - ❑ Check disk
  - ❑ critical proxy
  - ❑ missing support process ~Observability (flying blind issue)
- ❑ Dependency Health Checkers
  - ❑ Bad Configuration or State Metadata
  - ❑ Inability to communicate with Peers Services
  - ❑ Other issues: memory leaks, deadlocks can make server show errors

# Implementing Health Checkers

## Anomaly Detection

- ❑ Compare Server with peers To realize if is behaving oddly.
- ❑ Aggregate data and compare errors rates.

## Cannot Detect

- ❑ Clock Skew
- ❑ Old Code
- ❑ Any unanticipated failure more

## React to HC Failures

- ❑ Fail Open (ELB)
  - ❑ Central authority
  - ❑ When all fail - allow traffic
- ❑ Prioritize your Health
  - ❑ Max socket connections to avoid death spiral

# Going fast with CD

Takeaways:

- ❏ Always improve release process without being a blocker to business
- ❏ Add checkers on the Pipelines/Steps rather than manual process
- ❏ Reducing risk defect affects customers:
  - ❏ Deployment hygiene (Minimum health hosts ~ CodeDeploy)
  - ❏ Test Prior Production: Unit, Integration, Browser, Inject Failure
  - ❏ Validate in Production: Don't release all at once.
  - ❏ Deploys are done in business hours

# Timeouts, Retries, Backoff + Jitter

Takeaways:

- ❑ It's impossible to avoid failure(only reduce the probability)
- ❑ Basic Constructs to make systems more reliable(Google SRE saus the same):
  - ❑ Timeouts, Retry, Exponential Backoff + Jitter
- ❑ Retries make the client survive partial failures
- ❑ Pick the right timeout is hard. Too low: Increase traffic + latency
- ❑ Latency metrics help you to pick the right value
- ❑ Amazon accept the rate of false timeouts 0.1% (p99,9)

# Timeouts, Retries, Backoff + Jitter

## When Default strategy dont work:

- ❑ Clients with substantial network latency (over the internet)
- ❑ Clients with tight latency bound p99.9 close to p50
- ❑ Impls that does not cover DNS or TLS handshake times

## Retry Issues

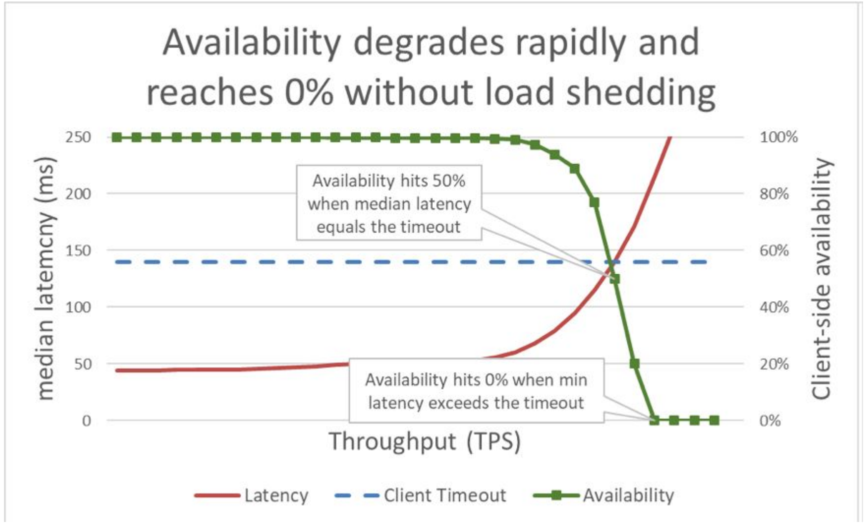
- ❑ Circuit Breakers introduce modal behavior which is difficult to test
- ❑ Local Token Bucket fix CB issues
- ❑ Local Token Bucket is on AWS SDK since 2016
- ❑ Also important to know when to retry and analyze http errors

## Using Load shedding to avoid overload

- ❑ Amazon avoid overload by design systems to scale proactively before the overload
- ❑ Protection in layers: Automatic Scaling, Shed excess load gracefully, monitoring all mechanisms and continuous testing
- ❑ University Scalability Law
  - ❑ Derivation of amdahl's law
  - ❑ Theory ~ University Scalability Law
    - ❑ While the system throughput can improve using parallelization
    - ❑ But its limited by the throughput points of serialization (what cannot be parallelized)

# Using Load shedding to avoid overload

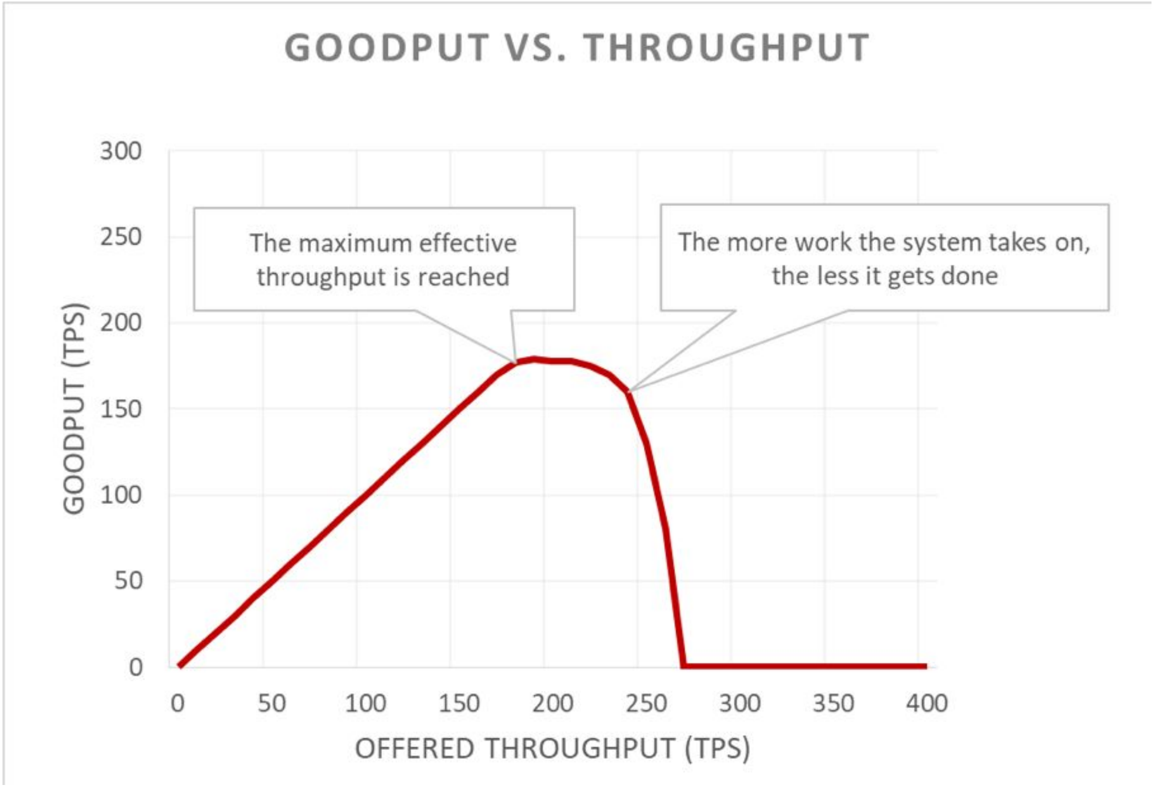
- ❏ Throughput is bounded by system resources
- ❏ Throughput also decreases with Overload



## Using Load shedding to avoid overload

- ❏ Graph is hard to read and is better distinguish good Goodput vs throughput
- ❏ Throughput = total number of requests per second (RPS)
- ❏ Goodput = subset of Throughput handle without errors and without low latency

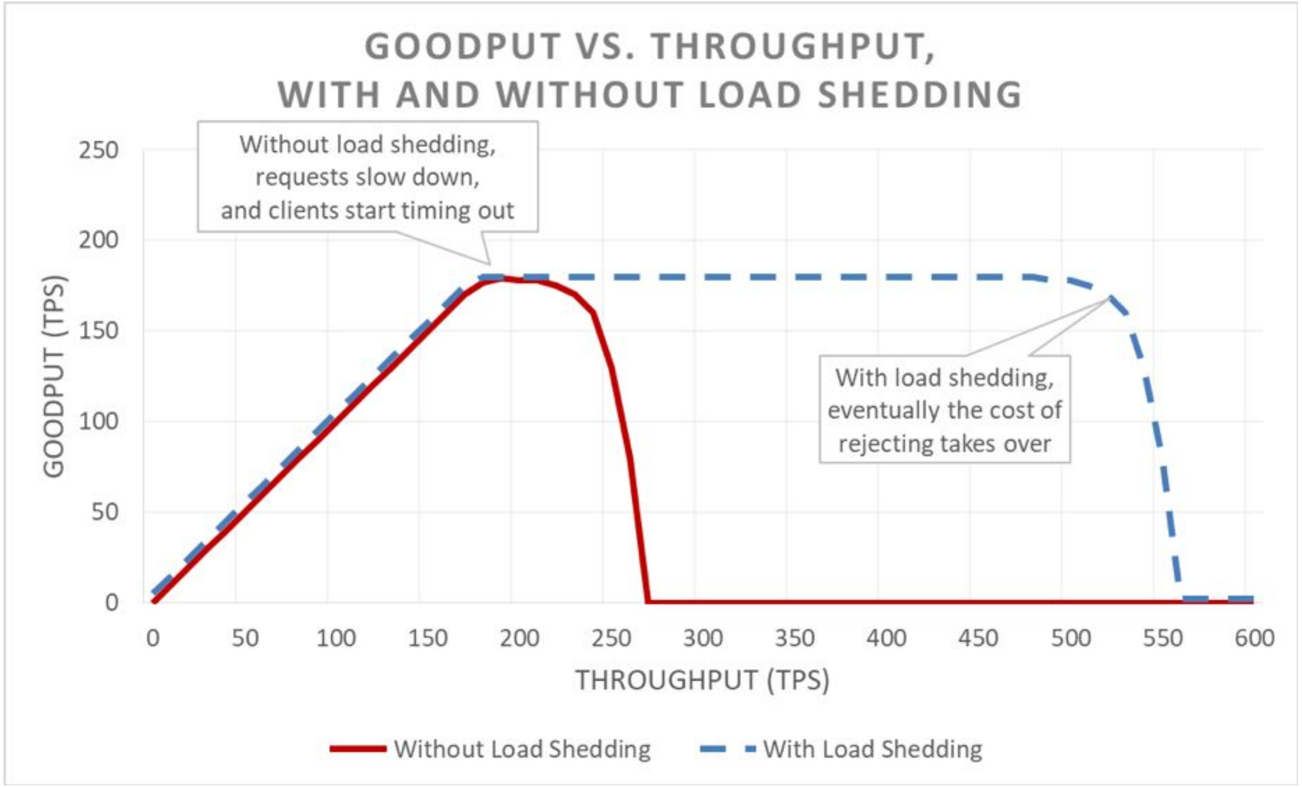
# Using Load shedding to avoid overload



# Using Load shedding to avoid overload

- Preventing work going to waste
  - Load Shedding: When server is overloaded start rejecting some requests.
  - Load Shedding: Goal - is to keep the latency low and makes the system more available
- Even with Load Shedding at some point server preys the price and amdahl's law and goodput drops.

## Using Load shedding to avoid overload



# Using Load shedding to avoid overload

- <u>Load Shedding mechanisms</u>
    - Overload might happen:
    - Unexpected traffic
    - Loss of Fleet Capacity (Bad Deployment of other reasons)
    - Client Shifting from making Cheap Requests (like cached reads) to expensive requests(cache misses or writes)
- <u>The cost of Dropping Requests</u>
    - Amazon drop requests only after the Goodput pletou
    - Amazon make sure the Cost of dropping requests is small
    - Dropping requests too early could be more expensive than it needs to be
    - In Rare cases dropping requests could be more expensive then holding the requests
    - In this cases amazon slow down rejecting requests to a minimum the latency of successful responses
- <u>Prioritize Requests</u>
    - The most important request the server will receive is the ping from load balancer
    - Prioritization and throttling can be used together
    - Amazon spend lots of time on placing algorithms but favors predictive provisioned workload over unpredictable workload

# Using Load shedding to avoid overload

- ❑ Keeping an eye on the clock
  - ❑ If the server realize the request is half-way and client timeout it could skip the rest of the work and fail the request
  - ❑ IT's important to include timeout hints on requests which tell the server how long the client will wait
  - ❑ IF an API has start() and end() operations end() should be prioritized over start().
  - ❑ Pagination can be dangerous - amazon design the services to perform bounded work and not paginate endlessly
- ❑ Watching out for queues
  - ❑ Look request duration when managing internal queue
  - ❑ Record how long the work was sitting on the queue waiting to be processed
  - ❑ Bounded Size Queues are important
  - ❑ Limit upper bound time that the work will wait on the queue and discard if pass it.
  - ❑ Sometimes use a LIFO approach which HTTP/2 supports
  - ❑ LB might queue incoming requests (Surge Queues) - these queues can lead to burnout
  - ❑ It's safair to use a spillover configuration which fails-fast instead of queueing
  - ❑ Classic ELB use surges queue but ALB reject excess traffic
- ❑ Protecting from overload in lower layers
  - ❑ MAX connection (like nginx has) is used as last resort and not as default mechanism
  - ❑ Iptables can be used to reject connection in emergencies
  - ❑ AWS WAF can shed excess traffic on a number of dimensions

# Avoid queue backlogs

- ❑ Queues suppose to increase availability could backfire make recovery time worst
- ❑ Queue-based system when system is down, message keep arriving (big backlog)
- ❑ Queue-based systems have 2 models

## Fast Mode

- ❑ When there is no backlog
- ❑ Latency is low
- ❑ System is fast

## Sinister Mode

- ❑ If load increase or failure happens
- ❑ End-2-end latency goes higher
- ❑ Sistener mode kicks in
- ❑ Takes long time to go back to fast mode.

# Avoid queue backlogs

## How to measure availability and latency?

- ❏ Producer Availability is proportional to queue availability
- ❏ IF we measure availability on consumer side it might look worse than it is.
- ❏ Availability Measures from DLQ.
- ❏ DLQ metrics are good but might detect the problem too late.
- ❏ SQS has timestamps for each message consumed from the queue : Can log produce netrics how behind it is.
- ❏ IoT Strategy: categorizing metrics of first attempts separate from metrics of the latency of retry attempts
- ❏ X-ray and Distribute tracing can help to understand/debug

# Avoid queue backlogs

<u>Backlogs in multi tenant async systems</u>

- ❑ Amazon don't expose internal queue direct to you (aws lambda)
- ❑ Throttling to guarantee fairness - per consumer rate-based limits
- ❑ Limits provide guard rails for unexpected spikes allowing aws do the provisionings need under the hood
- ❑ Design Patterns to avoid large queue backlogs
  - ❑ Protection at every layer - throttling
  - ❑ Using more than one queue helps to shape the traffic -
  - ❑ Real Time systems use FIFO but prefer LIFO behavior

# Avoid queue backlogs

## Amazon Approach: Creating Resilient multi tenant async systems

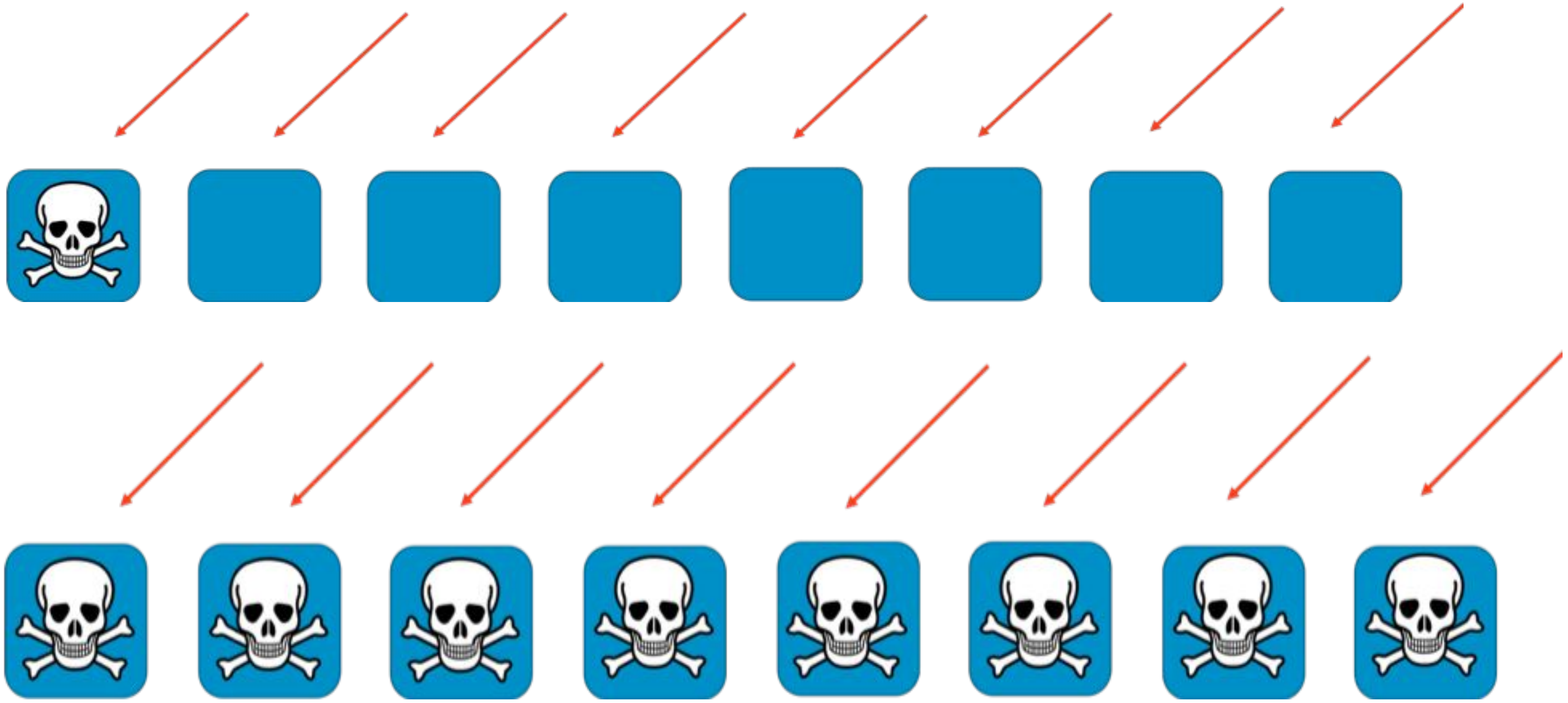
- ❑ Amazon Separate workloads in different queues
- ❑ Shuffle sharding - Aws lambda and IoT does have queues for every device/function
- ❑ Sideling excess traffic to separate queue
- ❑ Sideling old traffic to separate queue
- ❑ Dropping old messages
- ❑ Limiting Threads and other resources per queue
- ❑ Sending Back Pressure upstream - Amazon MQ
- ❑ Delay Queues
- ❑ Avoid many in-flight messages
- ❑ DLQ for messages that cannot be processed
- ❑ Ensuring additional buffer for polling threads workloads - to absorb bursts
- ❑ Heartbeating long running messages
- ❑ Plan for Cross-host debugging

# Workload isolation with shuffle sharding

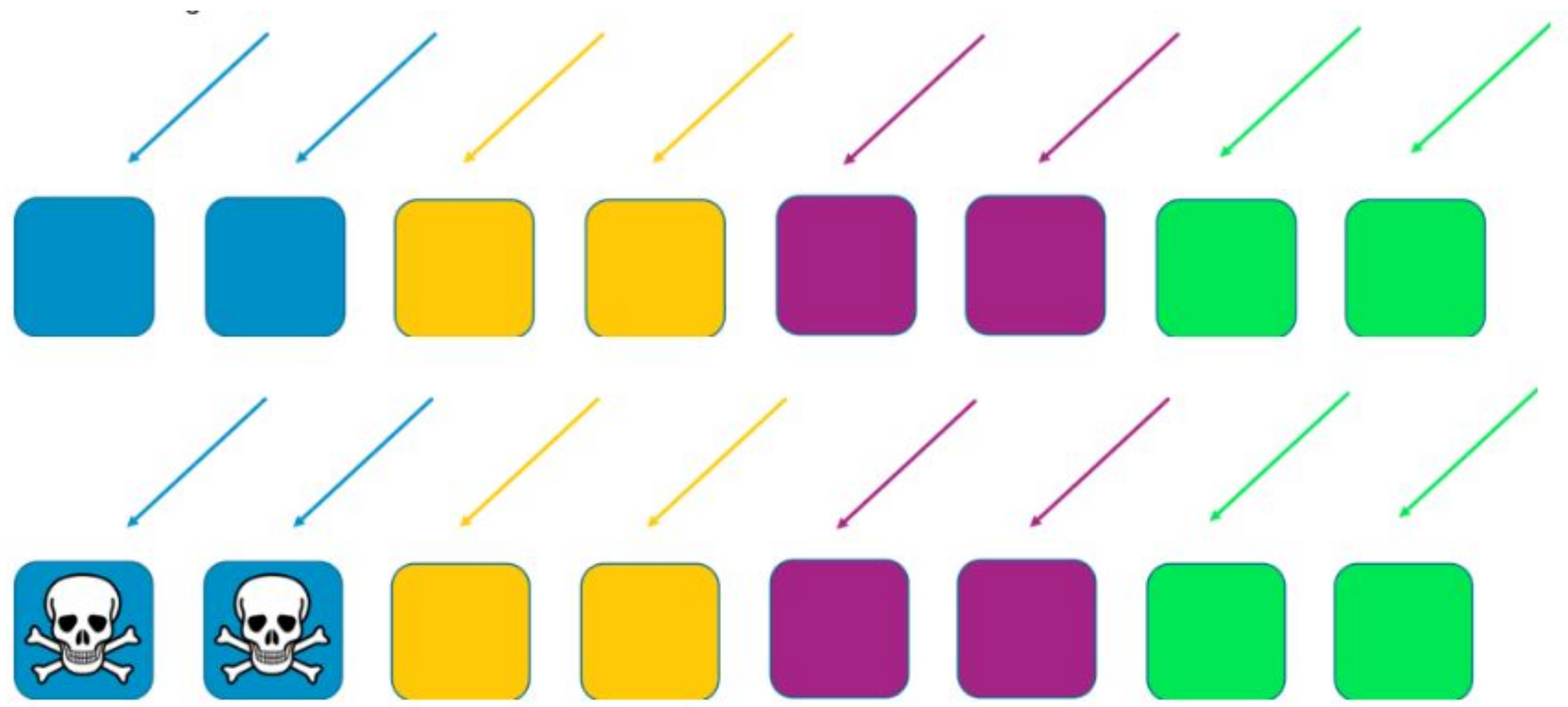
## Amazon Invented Shuffle Sharding

- ❏ Route53 serves the biggest websites in the world
- ❏ Use Amazon for Root Domain but thanks to Design decision made in DNS protocol on 1980 its not simple/easy
- ❏ CNAME offload part of the sub-domain to another provider but does not work at root top level
- ❏ To serve customer needs Amazon need to host customers domains.
- ❏ Host DNS is not small task if there is problems you can make the whole business OFFLINE
- ❏ Shuffle Sharding was invent to handle DDos attacks in Route53
- ❏ Powerful pattern to deliver cost-effective / multi-tenant services
- ❏ Regular sharding can make the whole system go down during a DDoS Attack - Scope of failure is "Everything for everyone".

# Workload isolation with shuffle sharding

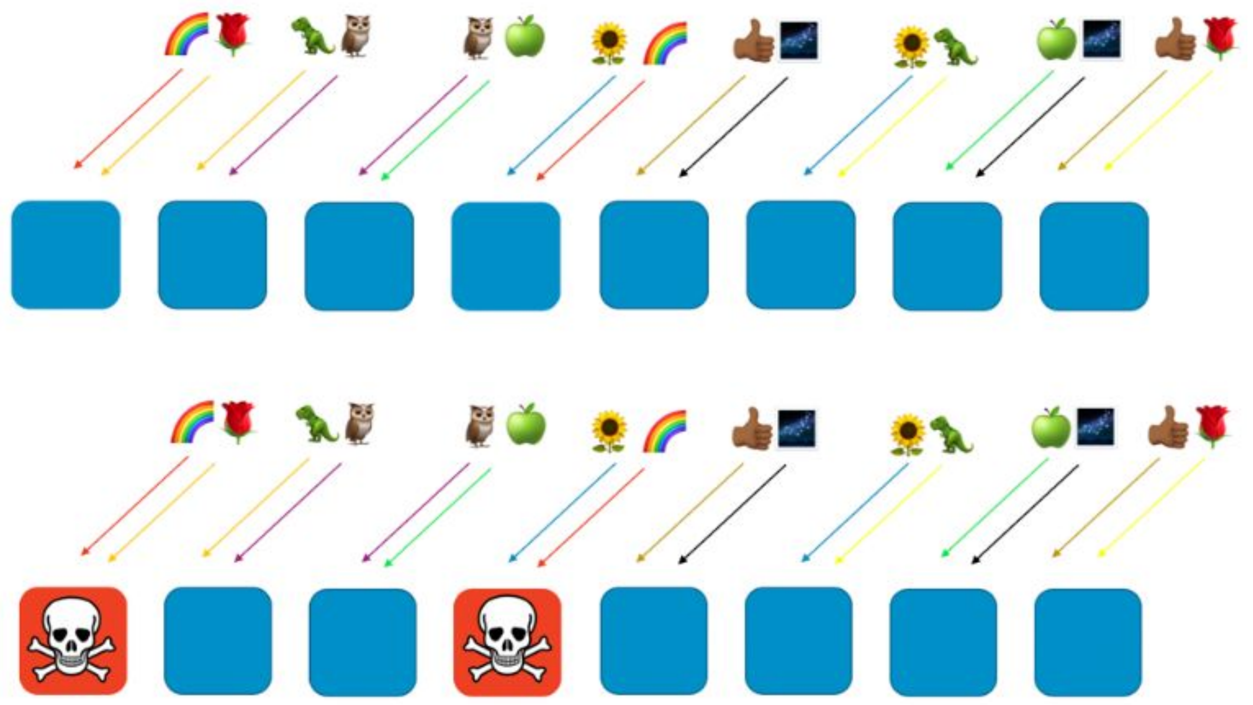


# Workload isolation with shuffle sharding



Divide the workers into 4 shards reduced the blast radius from 100% to 25%

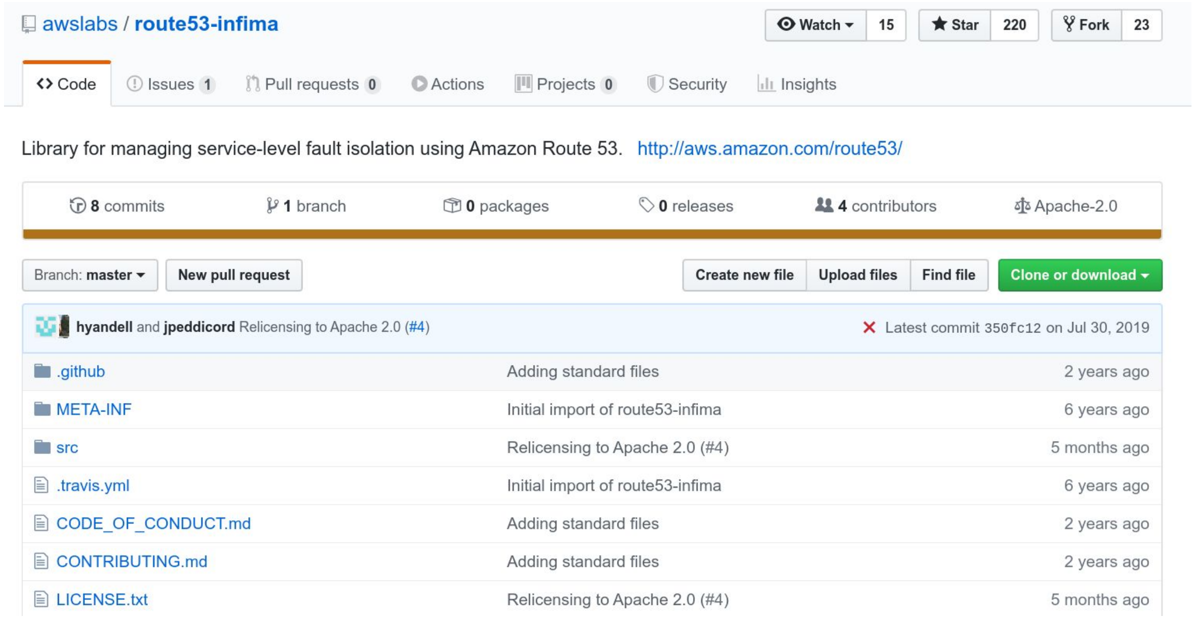
# Workload isolation with shuffle sharding



Shuffle Sharding we create virtual shards and divide even more - 8 workers = 28 unique combinations = 28 shuffle shards - Scope of the problem is 1/28 == 7 times better than regular sharding.

# Workload isolation with shuffle sharding

Route53 has 2048 virtual name servers == 730 billion shuffle shards == unique shuffle shard to every domain



Library for managing service-level fault isolation using Amazon Route 53. http://aws.amazon.com/route53/

8 commits | 1 branch | 0 packages | 0 releases | 4 contributors | Apache-2.0

hyandell and jpeddicord Relicensing to Apache 2.0 (#4) — Latest commit 350fc12 on Jul 30, 2019

| | | |
|---|---|---|
| .github | Adding standard files | 2 years ago |
| META-INF | Initial import of route53-infima | 6 years ago |
| src | Relicensing to Apache 2.0 (#4) | 5 months ago |
| .travis.yml | Initial import of route53-infima | 6 years ago |
| CODE_OF_CONDUCT.md | Adding standard files | 2 years ago |
| CONTRIBUTING.md | Adding standard files | 2 years ago |
| LICENSE.txt | Relicensing to Apache 2.0 (#4) | 5 months ago |

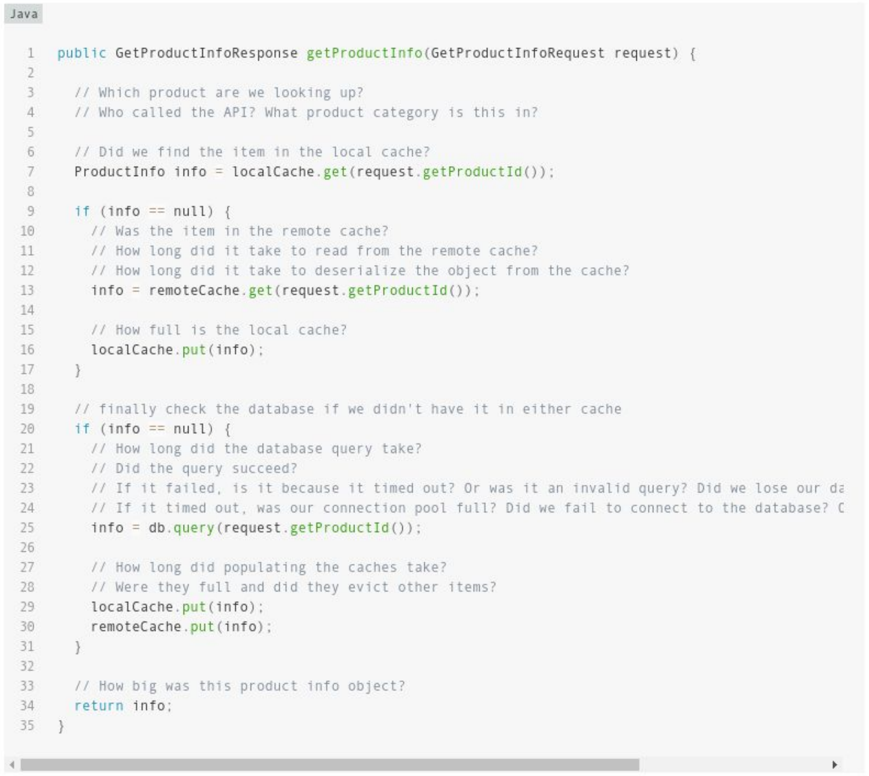
https://github.com/awslabs/route53-infima

# Instrumenting dist sys for Observability

## Amazon Learnings

- ❏ Great Instrumentation helps to see what experience we are giving to customers
- ❏ Amazon consider more than avg latency and focus on outliers p99.9 and p.99.99 - 1k in 10k request slow still poor experience.

# Instrumenting dist sys for Observability



```java
public GetProductInfoResponse getProductInfo(GetProductInfoRequest request) {

  // Which product are we looking up?
  // Who called the API? What product category is this in?

  // Did we find the item in the local cache?
  ProductInfo info = localCache.get(request.getProductId());

  if (info == null) {
    // Was the item in the remote cache?
    // How long did it take to read from the remote cache?
    // How long did it take to deserialize the object from the cache?
    info = remoteCache.get(request.getProductId());

    // How full is the local cache?
    localCache.put(info);
  }

  // finally check the database if we didn't have it in either cache
  if (info == null) {
    // How long did the database query take?
    // Did the query succeed?
    // If it failed, is it because it timed out? Or was it an invalid query? Did we lose our da
    // If it timed out, was our connection pool full? Did we fail to connect to the database? C
    info = db.query(request.getProductId());

    // How long did populating the caches take?
    // Were they full and did they evict other items?
    localCache.put(info);
    remoteCache.put(info);
  }

  // How big was this product info object?
  return info;
}
```

- ❑ Amazon has standard libraries to instrument logs and metrics.
- ❑ Amazon instrument logs with 2 kinds of data: Request data and Debug Data (different log files)

# Instrumenting dist sys for Observability

## Request Log Best Practices

- ❏ Emit 1 and 1 only log entry per request
- ❏ Record Request details before doing validations
- ❏ Sanitize request before logging (encode, escape, and truncate)
- ❏ Don't add 1MB Strings into the log just because is on the request
- ❏ Keep metric names short but not too short
- ❏ Break Long-running task (minutes / hours) in multiple logs entry
- ❏ Amazon Logs format are binary and use http://amzn.github.io/ion-docs/
- ❏ Ensure Log Volumes are big enough to handle at Max Throughput
- ❏ Consider Behavior of the system with disk full - Operate without log is risky, detect when server has a disk near to be full.

# Instrumenting dist sys for Observability

## Request Log Best Practices

- ❑ Synchronize clocks https://aws.amazon.com/blogs/aws/keeping-time-with-amazon-time-sync-service/
- ❑ Amazon also uses: https://chrony.tuxfamily.org/
- ❑ Emit zero counts for availability metrics
  - ❑ 1 Request succeeded
  - ❑ 0 Request failed

# Instrumenting dist sys for Observability

## What to Log?

- ❑ Log Availability and latency of dependencies
- ❑ Break out dependency metrics per call, per resource, per status code
- ❑ Record memory queue depth when accessing them
- ❑ Organize Errors by Category of Cause / Add Additional counter for error reason (Diego Pacheco Note: I did this in the past - called "Error Observability" - Also expose via REST)
- ❑ Log Important metadata about the unit of work
- ❑ Protect logs with access control and encryption

# Instrumenting dist sys for Observability

<u>What to Log?</u>

- ❑ Avoid putting overly sensitive information in logs
- ❑ Log Trace ID and propagate to backend calls (Diego Pacheco Note: I did this a lot also called MID(Message ID) generated at the Gateway/Edge layer and propagated to all calls via HTTP HEADERS and Message HEADERS .i.g: JMS).
- ❑ Log different latency metrics depending on status code and size
  - ❑ Categorized, like Small Request Latency and Large Request Latency

# Instrumenting dist sys for Observability

## Application Log Best Practices

- ❑ Keep the Application log free of spam - INFO / DEBUG are disabled in prod.
  - ❑ Application log is a location for trace information
- ❑ Include the corresponding request ID
- ❑ Rate-limit an application log error spam
- ❑ Prefer format strings over String#format or string concatenation. - Avoid Format String on DEBUG calls won't be called.
- ❑ Log request IDs from failed service calls

# Instrumenting dist sys for Observability

## High throughput Services Log Best Practices

- ❑ DynamoDB serves 20M RPS of amazon internal traffic
- ❑ Log Sampling - Write out every N entries not every single one. Prioritize Log slow and failure requests instead of successful ones.
- ❑ Offload serialization and log flushing to a separate thread.
- ❑ Frequent Log Rotation
- ❑ Write logs pre-compressed
- ❑ Write to a ramdisk / tmpfs
- ❑ In-memory aggregates / Monitor resource utilization

# Amazon Builder Library Notes

Diego Pacheco