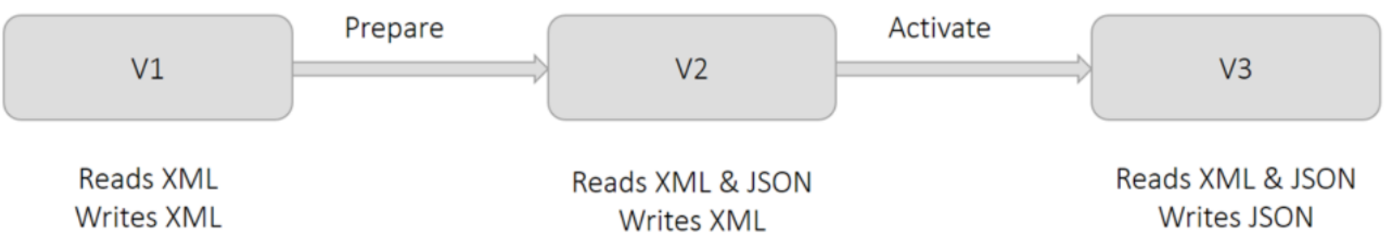
The Amazon Builders' Library

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

Notes by Diego Pacheco

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

- Avoid One Way Doors
- All code should be forward and backward compatible
- Backward compatible == (A) No Errors (B) No Service Disruption
- Stand Alone Software == Deploys are Atomic. Distributed: Rolling Update(Availability) more Complex
- Protocol Changes are hard - number 1 source of outages and breaks
- Examples of protocol Changes: Add Compression or Increase Heart beat frequency from 5 to 10s
- Explicit Check for backward and forward compatibility
- 2 Phase Deployment Pattern
- AWS recommends check phase 1 is successfully in all machines being deployed before move to phase 2
- Serialization Best Practices
  - Avoid Custom Format
  - Version Serializers
  - Avoid Serialize DS we dont control i.g: java collection objects via reflection
  - Design Serializers to allow unknown fields
- Check if is safe to go forward of backward: upgrade-downgrade test
- Use same Deploy config for Test and Prod envs
- Same order of deployments for Test and Prod



❎ 2 - Caching challenges and strategies
https://aws.amazon.com/builders-library/caching-challenges-and-strategies/?did=ba_card&trk=ba_card

- When To Cache?
  - Dependency Latency
  - Efficiency at a given Request Rate
- Start caching when find uneven request pattern: Hot Key, Hot Partition Throttling
- Caching: Trade Off: How Consistent or Tolerant to Eventual Consistency the system is. More tolerant longer cache.

- Local Caches: Added on Demand, No Operational Overhead, In-memory HashTable
- Local Caches: Has issues. Caches Coherence, Downstream load proportional to the fleet size, Metrics Hit/miss on Downstream.
- Local Caches: Has the COLD START issue
- External Caches: Like Memcached or Redis - Reduced Coherence Issues, Load on Downstream services is reduced, No Cold Start Issues.
- External Caches: Issues - More Complexity and Ops load
- External Caches: When EC downtime call downstream service could be dangerous(increase burnout and throttling) So is best use Local Caches or Load Shedding for fallback.
- Inline vs side caches: Inline cache(Read tought or Write Through) - Embedded Cache mgmt on API i.g: DAX, Nginx, Varnish.
- Inline vs side caches: Side Caches like: Elasticache(memcached / Redis) or libs like Guava or EHCache.
- Inline vs side caches: Inline cache, benefits: Standard API, cache mgmt out of app code, less bugs.
- Cache Expiration: TTL most common, Most common eviction policy LRU
- Cache Expiration: TTL Soft(Refresh) + TTL Hard(Invalidation case of downtime)
- Thundering Herd Problem(Cache Coalescing solution): Many clients requesting same key(uncached downstream) on empty local cache on instance that just joined the fleet at same time - Cache Coalescing: make sure just 1 request goes to the downstream deep. This situation could lead to burnout or throttling. Varnish and Nginx have this feature.

❎ 3 - Leader Election in Distributed Systems
https://aws.amazon.com/builders-library/leader-election-in-distributed-systems/?did=ba_card&trk=ba_card

- Leader Election is a tool for: Improve Efficiency, reduce coordination, simplify architectures and reduce operation
- Leader Election: Can introduce new Failure modes and scale bottlenecks and more difficult to evaluate if the system is correct.
- Leader Election: Single Leader Advantages:
  - Easier to understand - All concurrency in a single place - Reduce partial failure modes - single place to look logs/metrics
  - Works simply: inform systems about changes rather than build consensus.
  - Offers clients consistency - Improve performance, reduce costs
- Leader Election: Single Leader Downsides:
  - SPOFiggered tasks, it's a good idea to add jitteriggered tasks, it's a good idea to add jitter
  - Single Point of Scaling
  - Single point of Truth - Bad leader has high blast radius
  - Partial Deployments are hard to apply
- Common Pattern is Sharding. Multiple leader Avoid one way doors but each data belongs to a single leader. Usage (DynamoDB, EBS, EFS)
- Avoid Depending on time in Distributed Systems Change into a success the next moment as state propagates.
- Amazon uses leases depending on local time(elapsed duration) no dependency on wall-clock(no synchronization with dependencies).
- Large issue for Leases and distributed locks is make sure the leader just do work while has the lock.
- DynamoDB / Zookeeper offers lease-based locking clients that provide fault-tolerant leader election.

- Systems using leader election: All RDBMS, EBS, DynamoDB, QLDB, Kinises, KCL.
- Modeling systems withs TLA+
- Leader Election Best Practices:
    - Check remaining lease time frequently and before any ops with side-effect beyond the leader.
    - Consider: Slow network, timeouts, retrys, GC pauses can make remaining time of lease to expire
    - Avoid heart beating leases in background thread.
    - Have a reliable metric on how much work a leader can do VS how much work is doing. Make sure plans to scale to avoid running out of capacity.
    - Make it easy to find the host who is current leader, Audit trail of leadership changes
    - Model correctness with TLA+ to catch hard to observe and hard bugs.

iggered tasks, it's a good idea to add jitter

❎ 4 - Avoiding fallback in distributed systems

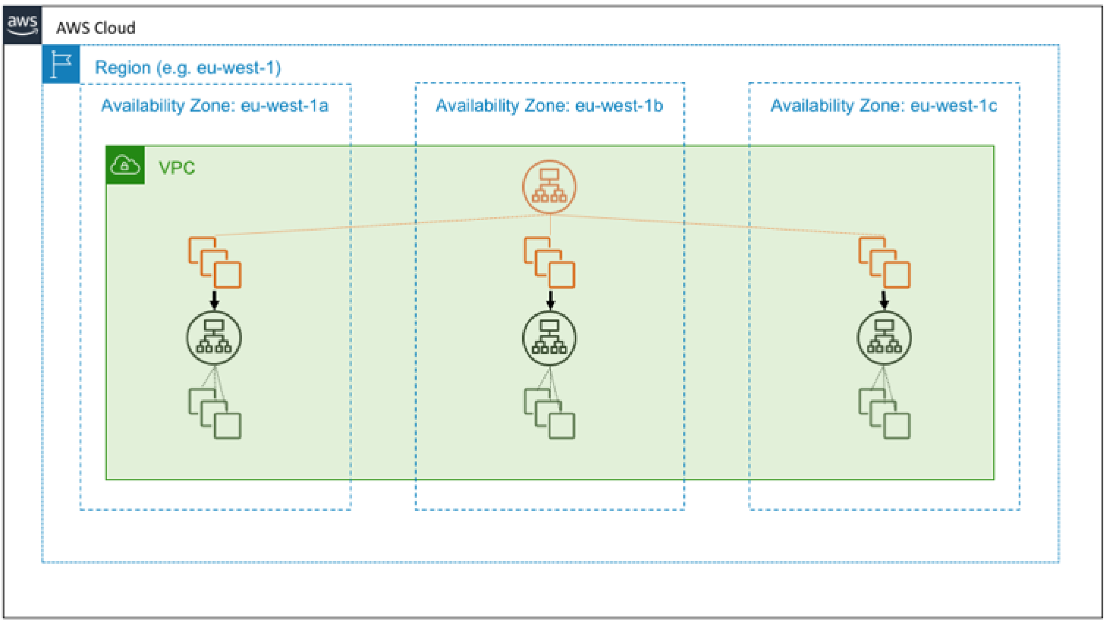
https://aws.amazon.com/builders-library/avoiding-fallback-in-distributed-systems/?did=ba_card&trk=ba_card

- 4 strategies to handle failures:
    - Retry: Try again after some delay
    - Proactive Retry: Perform activity multiple times in parallel and use first one to finish
    - Failover: Perform activity against different copy of the server or multiple in parallel and at least 1 will succeed
    - Fallback: Use Different mechanism to get same result
- Almost never use fallback strategies at Amazon. Bad Fallback strategy is hard to distinguish from good and creates lots of issues.
- Single Machine Fallback - IF memory alloc fails you cant do much often the machine will fail too.
- Single Machine Fallback - Fallback logic is hard to test - Amazon focus on making primary (non-fallback) more reliable.
- Single Machine Fallback - Fallback it self could fail - Fallback (lack of memory use SSD) might create other issues and make it worst.
- Single Machine Fallback - Fallback logic could place unpredictable load on the system (just logging msg could spike the CPU)
- Single Machine Fallback - Fallback not only could make it worst bu create a latency bug.
- Single Machine Fallback - The most common solution - Let it Crash the Application (Fail fast is a good strategy)
- Single Machine Fallback - For single machine applications is safaer to pre-allocate all heap in the startup.
- Single Machine Fallback - Amazon using this strategy(pre-allocate memory) for demons on EC2 that monitor CPU burst.
- Distributed Fallback: Has same issues that single fallback and more:
    - Distributed Fallback strategies are hard to test
    - Distributed Fallback strategies call fail as well
    - Distributed Fallback strategies often make the outage worst
    - Distributed Fallback strategies often not worth the risk
    - Distributed Fallback strategies often have latency bugs
- How Amazon Avoids Fallbacks:
    - Improve Reliability of non-fallback code
    - Let the Caller handle errors (by retry for instance)
    - Push Data Proactively (AIM credential pushed to several instances and valid for hours)
    - Convert Fallback into Failover

    - Ensure Retry / Timeouts dont become fallback
- Fallback code needs to be continually tested (often does not happen) better to stick the code you always rung and test.

✅ 5 - Static stability using Availability Zones
https://aws.amazon.com/builders-library/static-stability-using-availability-zones/?did=ba_card&trk=ba_card
iggered tasks, it's a good idea to add jitter

- Static Stability applied in Availability Zones
- EC2 was built to be Statically Stable
- AZs dont share power nor infrastructure, connected to each other with fast-fiber-optical network
- Static Stability
    - EC2 has control and data plane
    - EC2 Control Plane: finds physical server, allocate network interface, generate EBS volume, install SG rules, etc..
    - EC2 Data Plane: Routes packages to the VPC route table, R/W from amazon EBS volumes
    - Data plane is simpler therefore needs to have more availability than the control plane
    - EC2 Data plane is static available (control plane impairment: dont update machine on VPC or lose SG update) but traffic keeps working
- Control Plane, Data Plane, Static Stable are concepts to build HA because:
    - Data plane availability is more critical than control plane availability for customers.
    - EC2 Keep running is more important than ability to create new machines.
    - Data plane operates in orders of magnitude higher than control plane. Thus better to keep separate and scale separately.
    - Control plane happen to have more moving parts compared with data plane.
    - Data plane keeps working even if data plane fails.
- Static Stability Patterns
    - Active-Active on AZ: A LB Service
        - AZ is Over provisioned by 50%
        - AZ operates only at 66% of the level which was load tested.
        - In case of failure, health checkers will fail than the system will failover to other AZ which has capacity.
    - Active-Standby on AZ: Relational DB Sample
        - In case of failure RDS will replace DNS name of the master
    - Both Patterns provision capacity they dont need.
- Under the Hood Static Stability on EC2
    - EC2 has a ZONAL deployment calendar: 2 AZs in same regions will get deploys in different days.
    - Ec2 Deployment pattern: deploy 1 box then 1/N Servers
    - Ec2 does one step further and align with AZ boundary. So a deploy that affects one AZ and rollbacked and fixed.
    - Ec2 All packets flow stay under the AZ instead of crossing boundaries



❎ 6 - Implementing health checks
https://aws.amazon.com/builders-library/implementing-health-checks/?did=ba_card&trk=ba_card

- Small Failures with outsized impact
  - Amazon had a bug that generated blank pages in errors
  - Load Balancing strategy favor faster than slow servers
  - However the server did not know it was not healthy so the LB strategy make the problem worst
  - After lots of the servers on the fleet and had to reset them
- How to Measure Health Checkers
  - Liveness Check
  - Local Health Check: Check if the application is functioning
    - Checks like: If can write to the disk
    - Check if critical proxy is working like nginx
    - Missing support process(missing monitoring daemon) - Flying blind
  - Dependency Health Checks
    - Bad Configuration or State of metadata
    - Inability to communicate with peer services and dependencies
    - Other software issues like: memory leaks, deadlocks can make server spew errors
- Anomaly Detection
  - Compare if the server is behaving oddly compared with peers
  - Aggregating data and comparing error rates, latency data to find anomalies and remove the server.
  - Cannot detect:
    - Clock Skew
    - Old Code
    - Any unanticipated failure mode
  - In order to anomaly detection works:
    -

Servers should do approximately the same things
- Fleet should be relativity homogeneous(Instance Types)

- Reacting to Health check failures
  - Locally you could decide you are not healthy and remove your self to work queue or have a central authority that decides that.
  - Fail Open
    - Some LBs can act as smart central authority for severs failures
    - However when all servers fails LB fail open allowing traffic to all servers
  - Health Checks without a circuit breaker
    - Configure the work producer (load balancer, queue polling thread) to check liveness and local health checks - Servers are take out of the LB is a local problem like bad disk.
    - Configure external monitoring system to perform dependency health check and anomaly detection
- Prioritize your Health
    - Max socket connections to avoid death spiral
- Real things that gone wrong with Health Checks
  - Deployments
  - Async process (SQS / Kinesis)
  - Disks Filling up
  - Zombies

❎ 7 - Challenges with distributed systems
https://aws.amazon.com/builders-library/challenges-with-distributed-systems/?did=ba_card&trk=ba_card

- Distributed Bugs are often latent
- Distributed bugs spread epidemically
- Engineers need to consider many permutations of failures
- Network call result could be unknown
- Distributed problems can occur on logical levels not only on low level machines
- Distributed problems get worse at higher levels of the system due recursion

❎ 8 - Going faster with continuous delivery
https://aws.amazon.com/builders-library/going-faster-with-continuous-delivery/?did=ba_card&trk=ba_card

- Amazon HOST Build system is called Brazil
- Amazon Deployment system called Apollo
- Pipelines and Steps
- Always improve release process without being a blocker to the business
- Engineers make a list of top best practices that become input for the checkers.
- Amazon added checks in tools for build/deploy looking for best practices
- Reducing the risk a defect will reach customer
  - Deployment Hygiene - Make sure deployment works - Minimum health hosts on Codedeploy - avoid disrupt customers
  - Test prior to production - Unit, Integration, Pre-production testing, failure injection, automated browser testing
  - Validation in Production - dont release code all at once, deploy in cell to a customer, very cauting release only to small number of customers, monitor amount of errors on a canary deployment, if error rate goes up we automatically rollback the deploy. i.g: they might wait

for 3k positive data points before continuing a deployment. Some teams wait for hours and other per minutes do decide if the deploy was successful or not.
  - Keep in mind the higher the impact - longer is to remediate.
  - Deploys are done in business hours and when there is low customer traffic - when there is lots of people watching and could help if something goes wrong

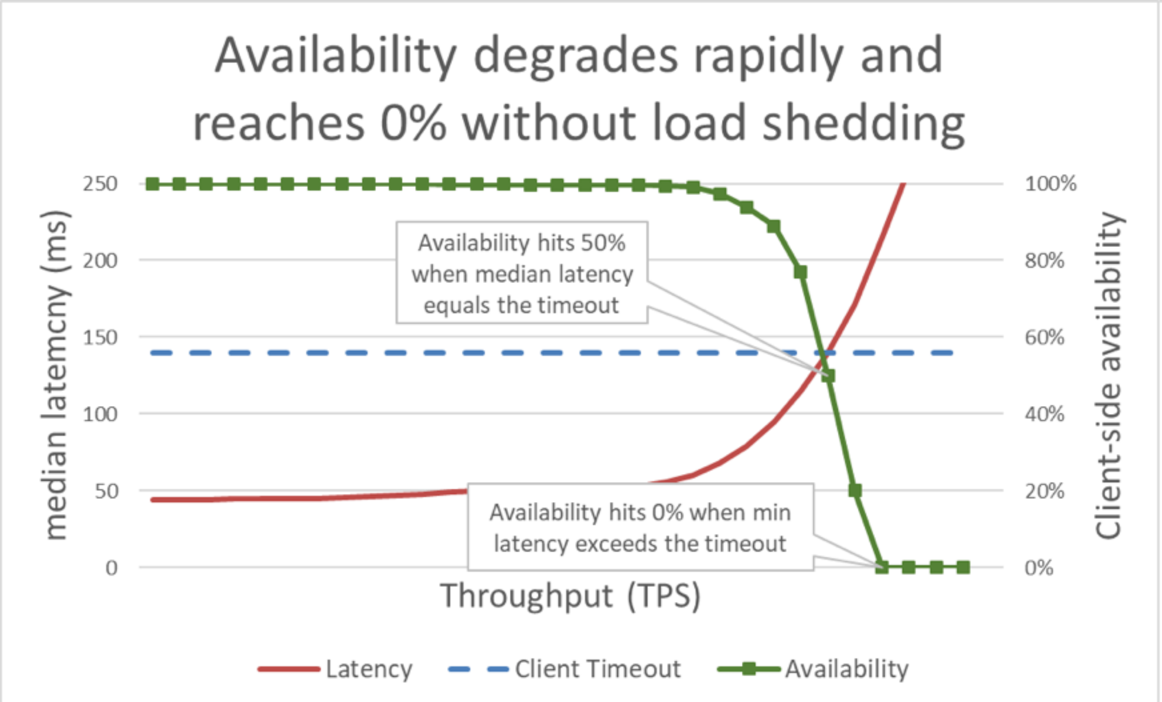
❎ 9 - Timeouts, retries, and backoff with jitter
https://aws.amazon.com/builders-library/timeouts-retries-and-backoff-with-jitter/?did=ba_card&trk=ba_card

- Reduce the probability of failure but is impossible to build systems that never fail.
- Basic Constructs to build resilient systems: Timeouts, Retrys, backoff.
- Why we use timeouts: Avoid client holding resources: memory, cpu, thread for unlimited amount of time
- Often just trying the request again works.
- Retries make the client survive partial random failures.
- Not always is safe todo retries - Retry can increase the load on the system who is calling.
- To fix this problem we need to use Backoff, increasing time of way after all retries.
- Jitter: Some random time between retries to avoid all clients request at same time.
- Any remote call should have timeouts: Connection timeouts, requests timeout.
- Pick the right timeout is hard
  - Too big makes losing it utility
  - Too low: Increases traffic, Increase latency and might leading to a complete outage.
- Latency metrics help you to choose the right time.
- Amazon accepts rate of false timeouts = 0.1% and look p99.9 works well in several cases but:
  - Clients with substantial network latencies(over the internet)
  - Clients with tight latency bounds where p99.9 is close to p50
  - Implementations where does not cover: DNS or TLS Handshakes
- Retries issues
  - Circuit Breakers - introduce modal behavior which can be difficult to test - also adds significant time to recover
  - The circuit breaker issue could be mitigated with a Local Token Bucket - Allows calls to retries as long as has tokens.
  - Local Token Bucket is present in AWS SDK since 2016
  - Is also important know when to retry, analyze HTTP error before retry.
- Jitter allows reduce load on the servers

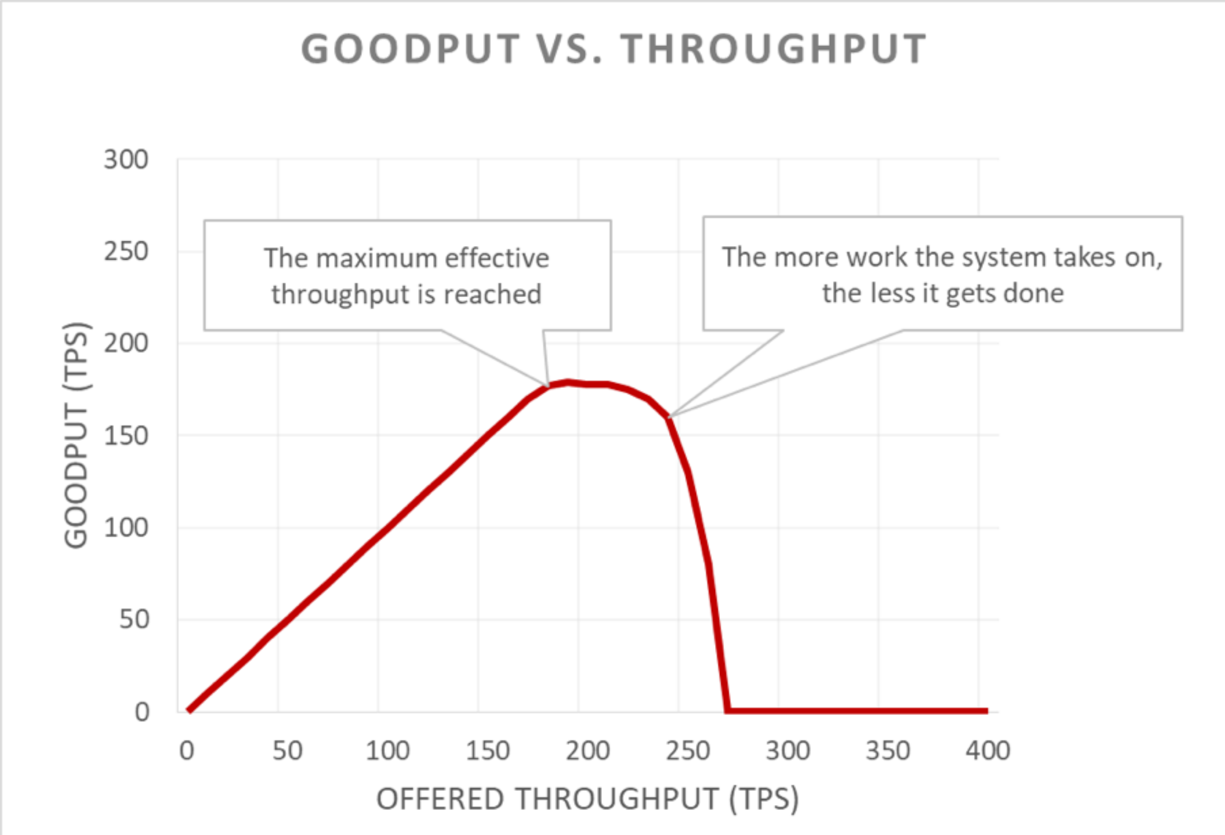
❎ 10 - Using load shedding to avoid overload
https://aws.amazon.com/builders-library/using-load-shedding-to-avoid-overload/?did=ba_card&trk=ba_card

- Services framework team
- Building tools for Route53 and ELB teams for productivity and reliability
- One challenge is how to have sensible defaults for performance/availability features(i.g: client-side timeout)
- Determine the max number of connections was challenging - too low would result in under performing and resources not being used, too high might result in overloads and outages.
- Amazon discovered max number of connection was a imprecise concept and focus then in load shedding
- Anatomy of an Overload

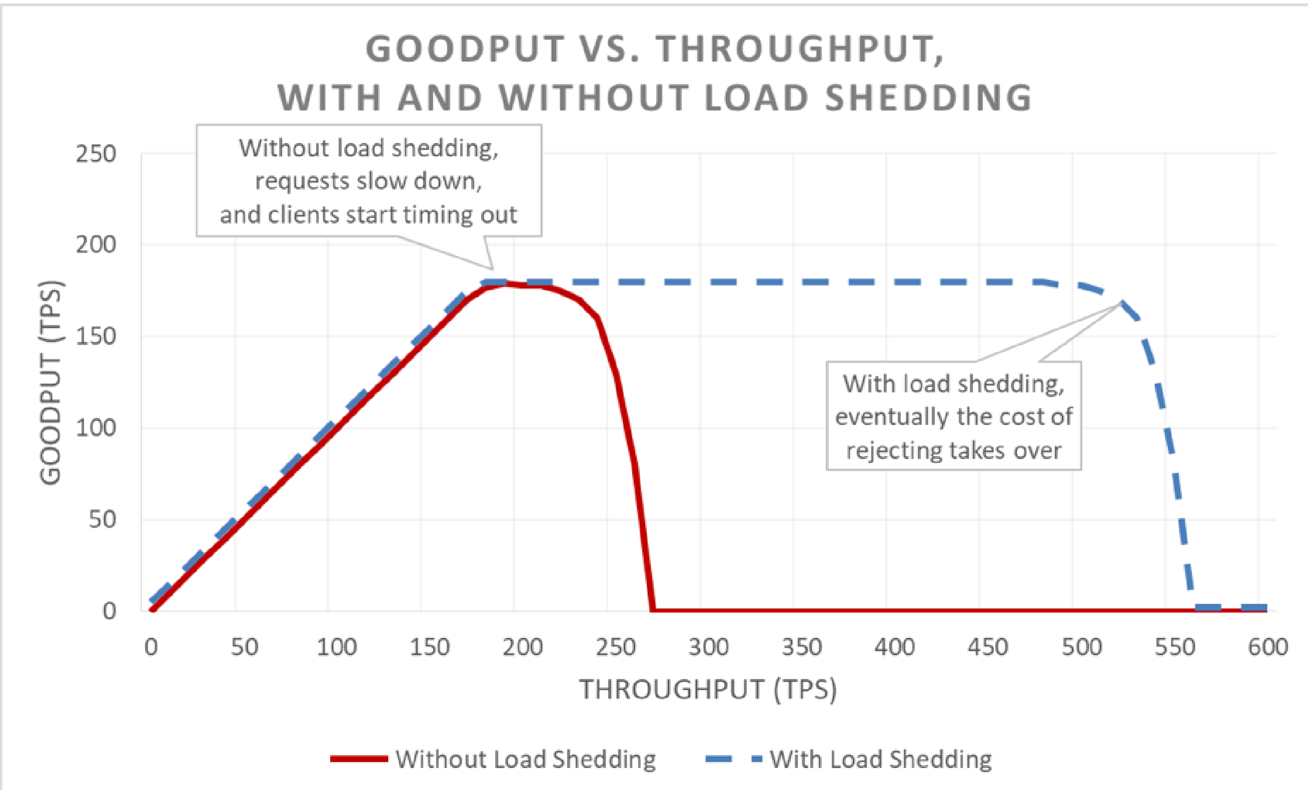
- Amazon avoid overload by design systems to scale proactively before the overload
- Protection in layers: Automatic Scaling, Shed excess load gracefully, monitoring all mechanisms and continuous testing
- University Scalability Law
  - Derivation of amdahl's law
  - Theory:
    - While the system throughput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



- Positive Feedback Loops
  - Worst thing on overload is all progress server so far made goes WASTE
  - Several layers(SOA) doing retries might make even worst and create they own feedback loop creating overload as steady state.
- 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.



- Load Testing
  -

Amazon spend lots of time doing Load Testing
  - Some load testes make sure the service automatically scales
  - IF during load test throughput increase but availability decreases is a sign you need load shedding mechanisms
  - During tests need to measure client's perceived availability
- Visibility
  - Proper instrumentation when load shedding happens to know - the client, operation and other info. to Tune up protection.
  - When Load Shedding is happening: Not pollute service metrics with failed requests (load shedding latency is pretty amazing)
- Load Shedding Effects on Automatic Scaling and AZ failure
  - Load Shedding might about CPU grow and prevent proactive-scaling
  - Load Shedding Might make the server operate much close to its limits and might exceed the limit provisioned by AZ.
- Load Shedding can save costs by shaping off off-peak non-critical traffic
- Load Shedding mechanisms
  - 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)
  - The cost of Dropping Requests
    - 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
  - Prioritize Requests
    - 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
  - 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
    - ITs 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

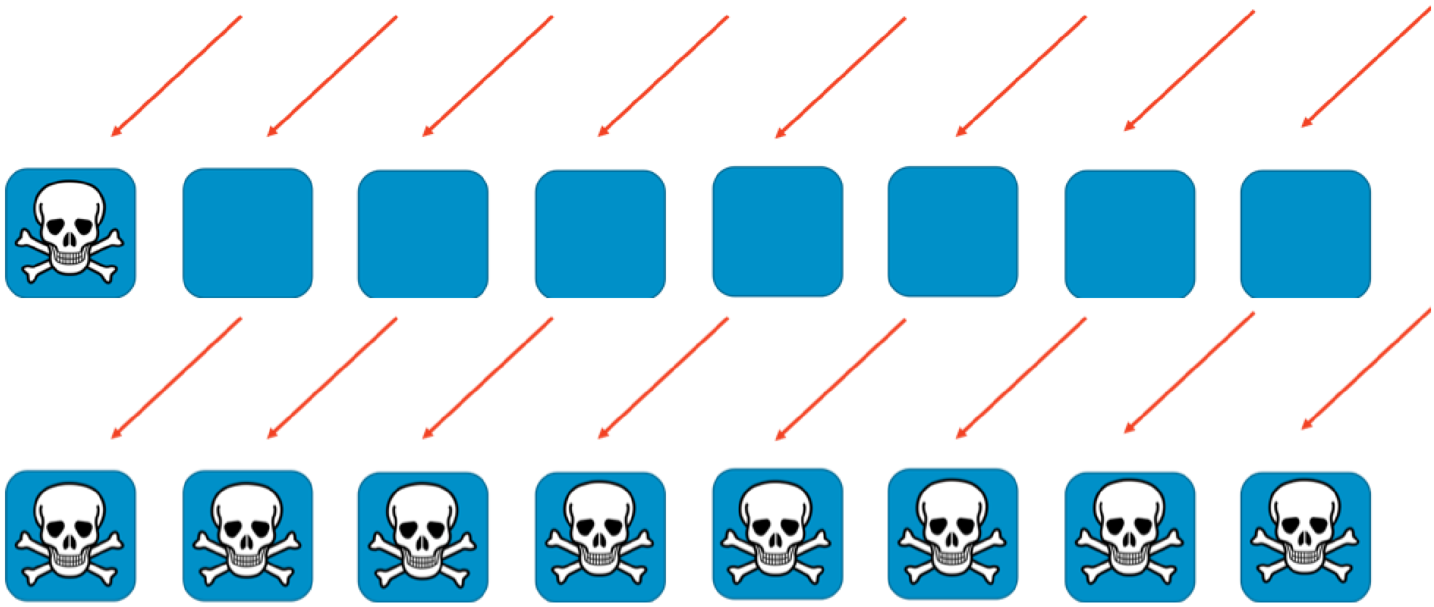
❎ 11 - Avoiding insurmountable queue backlogs
https://aws.amazon.com/builders-library/avoiding-insurmountable-queue-backlogs/?did=ba_card&trk=ba_card

- Queues that suppose to increase availability could backfire and increase recovery time after outage
- Queue system that the system is down but the queue keeps receiving messages message debt could build a large backlog
- Work can be done too late for the results be useful
- Another way to put it is that queue-bases systems has 2 modes of operations, or, bimodal behavior
  - When there is no backlog on the queue the system latency is low and system is fast mode.
  - IF loads or failure happens - it flips into a sinister mode - end-2-end latency grows higher and take a lot of time to go through the backlog.
- Queue-based systems
  - AWS Lambda has a durable queue in order to make sure your function runs even in face of failures
  - IoT Pub/Sub
  - SQS Durable Queue
- Failures in Async Systems
  - During failures could build up a huge backlog vs sync systems that just drop all messages
  - Async systems need to care about latency - traditional wisdom say we should not worry about latency.
- 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
- Backlogs in multi tenant async systems
  - Amazon dont 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 -
    - Realtime systems use FIFO but prefer LIFO behavior
- Amazon Approach: Creating Resilient multi tenant async systems
  - Separate workloads in different queues
  - Shuffle sharding - Aws lambda and IoT does have queues for every device/function

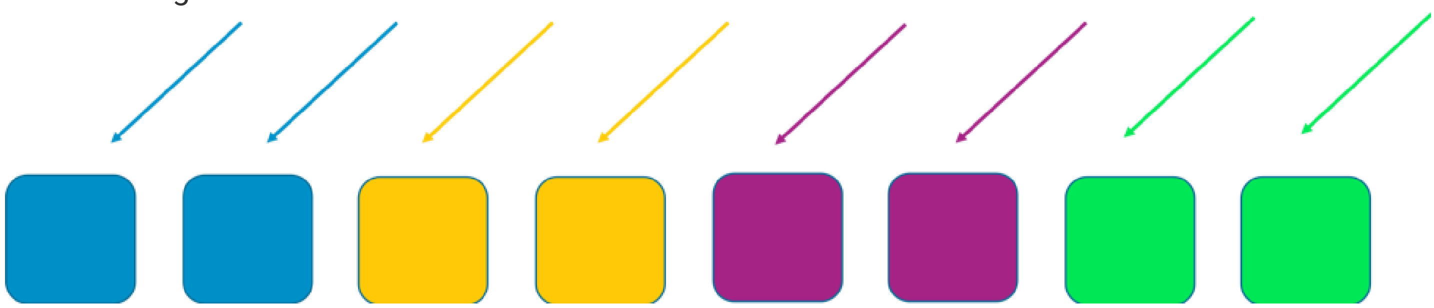
- Sideling excess traffic to separate queue
- Sideling old traffic to separate queue
- Dropping old messages
- Limiting Threads and other resources per queue
- Sending Back Pressure upstream - Amazon MQ
- Delay Queues
- Avoid many in-flight messages
- DLQ for messages that cannot be processed
- Ensuring additional buffer for polling threads workloads - to absorb bursts
- Heartbeating long running messages
- Plan for Cross-host debugging

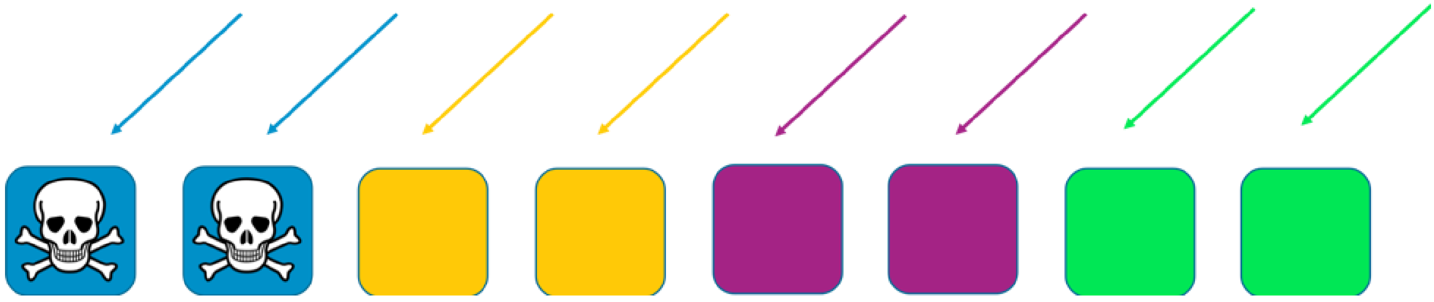
12 - Workload isolation using shuffle-sharding
https://aws.amazon.com/builders-library/workload-isolation-using-shuffle-sharding/?did=ba_card&trk=ba_card

- 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 other 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

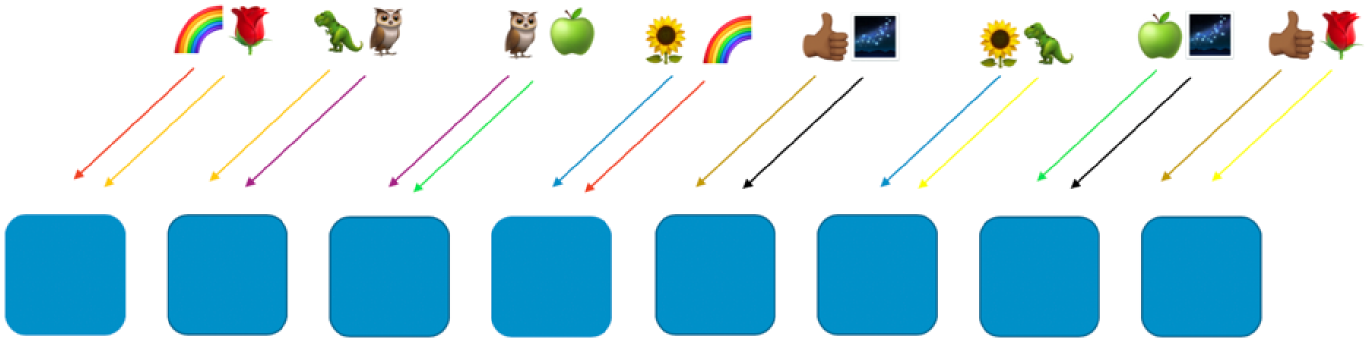


- Dividing the workers in 4 shards we can reduce the blast radius





- With shuffle sharding virtual shards are created



- 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.
- Route53 has 2048 virtual name servers == 730 billion shuffle shards == unique shuffle shard to every domain
- https://github.com/awslabs/route53-infima

- [ ] 13 - Instrumenting distributed systems for operational visibility
https://aws.amazon.com/builders-library/instrumenting-distributed-systems-for-operational-visibility/?did=ba_card&trk=ba_card

- 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.
- Instrumentation Sample Snippet

Java

```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)
- Request Log Best Practices
  - Emit 1 request log entry per unit of work
  - Emit no more than 1 request log entry for a given request
  - Break Long-running task (minutes / hours) in multiple logs entry
  - Record Request details before doing validations
  - Sanitize request before logging (encode, escape, and truncate)
  - Dont add 1MB Strings into the log just because is on the request
  - Keep metric names short but not too short
  - 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.
  - Synchronize clocks -> https://aws.amazon.com/blogs/aws/keeping-time-with-amazon-time-sync-service/
  - Amazon also uses Chrony https://chrony.tuxfamily.org/
  - Emit zero counts for availability metrics
    - 1 request succeed

- 0 when request fails
- What to Log?
  - Log Availability and latency of dependencies
  - Break out dependency metrics per call, per resource, per status code
    - Amazon Dynamodb: latency Metrics per table, per error code, per number of retries
  - Record memory queue depth when accessing them
  - Add Additional counter for error reason (Diego Pacheco Note: I did this in the past - called "Error Observability" - Also expose via REST)
  - Organize Errors by Category of Cause
  - Log Important metadata about the unit of work
  - Protect logs with access control and encryption
  - Avoid putting overly sensitive information in logs
  - Log Trace ID and propagate to backend calls (Diego Pacheco Note: I did this alot 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
- 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.
- High Throughput Service 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 requests and failures over successful ones https://en.wikipedia.org/wiki/Reservoir_sampling
  - 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.