Prometheus Stuff

Analyzing Prom Metric Data.

- when you have a large series of data points, that have been scraped over a definite interval of time, to make sense of the values, we tend to take aggregations over time, by applying a moving window.
- Prometheus provides some functions called <something>_over_time()
  - they can be only applied to range vectors.
  - they are window aggregation functions
  - It takes a range vector and produces an instant vector.
- Prom has (range vectors) and (instant vectors)
  - ↳ have a time dimension. ↳ only defines values that have been most recently scraped.
- The time range to consider depends on a sensible multiple of the scrape interval (ideally 4x the interval).
  - ↳ too small and the results are sharp
  - ↳ too large and the results are smooth, harder to spot the spikes.

- rate() - how does it work?

  rate(x[35sec]) = difference in value over 35 seconds / 35 sec

  ↳ takes into account, all the data points

  irate() only uses the first & last data points

  metric type : counter, is something that is constantly increasing.

  metric type : guage, is something that can increase or decrease.

- How to find the avg. req
histogram_quantile (0.95, sur [redacted] n])) by (le))
↑ 95th percentile ↑ less than

-> rate() only works
- rate() also sou [redacted] that don't roll up.
- if you want to use other aggregations, you need to use rate first, to account of data point resets.
- DO NOT USE rate() on gauges as the reset-detection logic will mistakenly catch the values going down as a "counter reset", and we will get wrong results.
- Rate performs extrapolation; when rate() is executed at a point in time, there is bound to be some data missing

- A few thoughts about Gauge metric type:
  - When a value of a metric can go up or down
  - When you don't need to query its rate.
  - memory usage, queue size, no. of requests in progress etc.

Prom QL - avg_over_time(metric_1[time]) ; avg_over_something( )

- Histogram: [when you want to take many measurements of a value, to calculate % later]
  - Frequency of value operations that fall into specific predefined buckets.
  - eg. request duration for specific type (HTTP, gRPC etc) calls.
  - Rather than storing each duration for each request,
  - Default bucket values: [0.005, 0.01, 0.025, 0.05, 0.075, 0.1, 0.25, 0.5, 0.75, 1, 2.5, 5, 7.5, 10]

-eg: Req duration, Res size

- Cardinality : overall count of values for one label.
  - low cardinality = 1:5
  - standard cardinality = 1:80
  - high cardinality = 1:10,000

  } label : value ratio.

Cardinality corresponds to number of metric series.

- When does this happen?
  When you instrument your code and add a new metric, sometime you attach more content than you need.

- Prometheus is NOT a TSDB. It is a monitoring system that uses a TSDB under the hood.

- Timeseries : It is a stream of time stamped values sharing the same metric and set of labels.
- Metric : It is a group of time series.
  Labels : They are just key-value pairs
  * Different metrics can have same labels.

Questions to consider about PROM integration:

Q1. How metric collection works? Push vs Pull, aggregation on server or client?
Q2. How metrics are stored? - raw samples or aggregated data? Roll up or Retention strategies.
Q3 How to query metrics? what does execution look like?
Q4. How to plot query results? what approximation errors get introduced by graphing tools?

92000 Ondo

Prometheus Operator

- why do we need it?

  so, if we have a monolithic prometheus, we run into:

  - Deployment that has some no. of desired replicas
  - But, the configmap that has the config that prom pod will use, will be massive, based on all the scrape configs.
  - Managing this will be a huge pain

- The role of the operator is to remove the management of this configmap and management of the pods, makes it easy to manage multiple prom instances.
- The operator, creates CustomResourceDefinitions
- we use a new object call serviceMonitor, That will monitor the apps or nodes that we want

typically [req duration / response sizes]

Histograms in Prometheus

- Histograms and summaries both sample observations
- calculate — count of observations
  \ sum of observed values

- metric_count : counter, only goes up, can do a rate() on it.
- metric_sum : also behaves like a counter, as long as the values are not negative
- to calculate the average metric_duration in the last 5m,

```
rate(metric_duration_sum[5m]) /
rate(metric_duration_count[5m])
```

- Histogram use-case

Let's say I want to have: 95% of requests < 300ms

- Configure a histogram with a bucket, upper limit of 0.3 sec
- for requests served in the last 5m,

```
sum(rate(metric_duration_bucket{le="0.3"}[5m])) by (some-label)
/
sum(rate(metric_duration_count[5m])) by (some-label)
```

- Apdex score
  - Bucket with target le upper bound
  - Bucket with tolerated le (usually 4x of target)
  - if target = 300 ms
    toleration = 300 x 4 = 1.2s

Apdex score:

```
( sum(rate(metric_duration_bucket{le="0.3"}[5m])) by (label)
+ sum(rate(metric_duration_bucket{le="1.2"}[5m])) by (label)
) / 2 / sum(rate(metric_duration_count[5m])) by (label)
```

- we divide the sum of both buckets, because the buckets are cumulative $\left[M_i = \sum_{j=1}^{i} m_j\right]$ histogram (counts the cumulative no of observations in all of the bins up to the specified bin)
  - cumulative histogram

- The "le = 0.3" is also contained in "le = 1.2". Dividing by 2, corrects that.

Quantiles:

- The q-quantile is the observation value that ranks at number $q \times N$, for N observations. $q = 0.5$ is median. } Basically, Quantile = Percentile.

- where can I use quantiles?
  I do not want an SLO saying 95% with 300ms.
  I WANT the metric_duration within which, I have been served, 95% of the metric_duration

- To do this, you can configure buckets around your target (let's say 300 ms), so: le = (0.1, 0.2, 0.3)
- If your service runs with multiple instances, you will collect request durations from each one and aggregate them.

WARNING: DO NOT DO: avg (metric_duration{quantile="0.95"}) //BAD

histogram_quantile (0.95, sum (rate (metric_duration_bucket[5m])) by (le))

1) If you need to aggregate, choose histograms
2) Choose histogram, if you have an idea of the range and distribution of values.
3) Choose Summary if you need accurate quantile

— A closer look at rate() in the SLO/SLI context-

1) SLI: metric_duration_bucket{le="0.3"} // represents histogram of durations.

Rate: rate (metric_duration_bucket{le="0.3"}[5m])

Let's assume:	(value)
10:00	=> 100
10:01	=> 120
10:02	=> 130
10:03	=> 140
10:04	=> 160

at 10:00 => 100
at 10:04 => 160
} 5 mins window

$$\text{rate} = \frac{(160-100)}{(300)} = 0.2$$

∴ On average, there were 0.2 metric_durations that were <= 0.3 seconds over 5m.

2) SLI : metric_duration_count{} // represent total count of durations
rate(metric-duration_count[5m])

10:00 => 100	@ 10:00 => 100
10:01 => 120	@ 10:04 => 160
10:02 => 130	
10:03 => 140	∴ rate : $\frac{160-100}{300s} = 0.2$
10:04 => 160	

On an average, there were 0.2 new durations recorded per second over 5 minute window

- why 5m window:
  - It really depends BUT
  - → 5m provides a reasonable balance between granularity and computational efficiency
  - → 5m allows you to detect changes quickly, sensitivity (accounts for sudden spikes)
  - → 5m is regarded as statistically significant collection
  - → 5m is a good tradeoff between recent behaviour & fluctuations.

- For the SLO, why take a rate and then the sum of that rate?

- So, if we are aggregating by a specific label, we first find the per second rate of change of observations in our bucket with our le qualifier, this is over time [t]. This tells us how fast the values are increasing or decreasing over time
- So, for each "label" we want to consider, we will have individual rate for each "label" value.
- So, then we take the sum to aggregate all of them so we get the total rate of change of metric/sec for each (label)

- we do this for metric_duration_bucket{le="something"} & metric_duration_count{}

Then we divide the $\frac{\text{sum of bucket rates}}{\text{sum of total counts}}$

- This enables us to compare and analyse the behaviour and performance of different (label) based on duration.

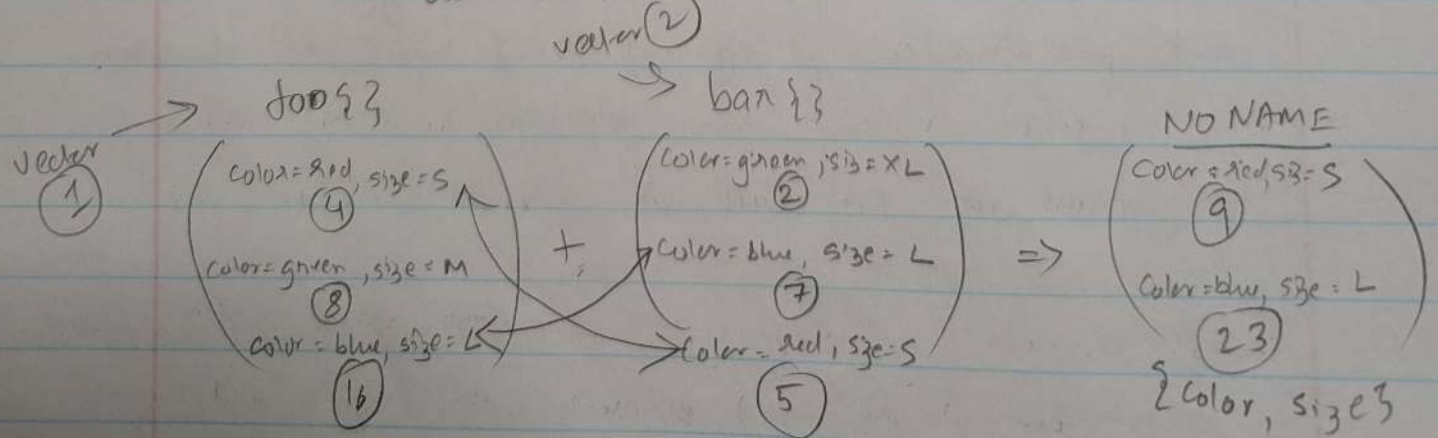
Matching in Prometheus

Scalar: Single numeric value associated with a metric at a given point in time.
They represent instantaneous measurements of a metric
They do not have a label/dimension associated.
eg. CPU usage, memory usage, no. of requests
cpu_usage {instance = "server1"} 3.62 @ 10:04pm
↑ scalar

Vector: Set of time series datapoints, each associated with specific labels.
eg. http_requests_total {job = "web-server", status = "200"}
represents a total number of HTTP requests with status code = 200 for job = web-server

Matching:

→ One - to - One matching
- for every element in the vector on the left, the operator tries to find a single matching element on the right.
- all labels are compared
- Elements without match are discarded.



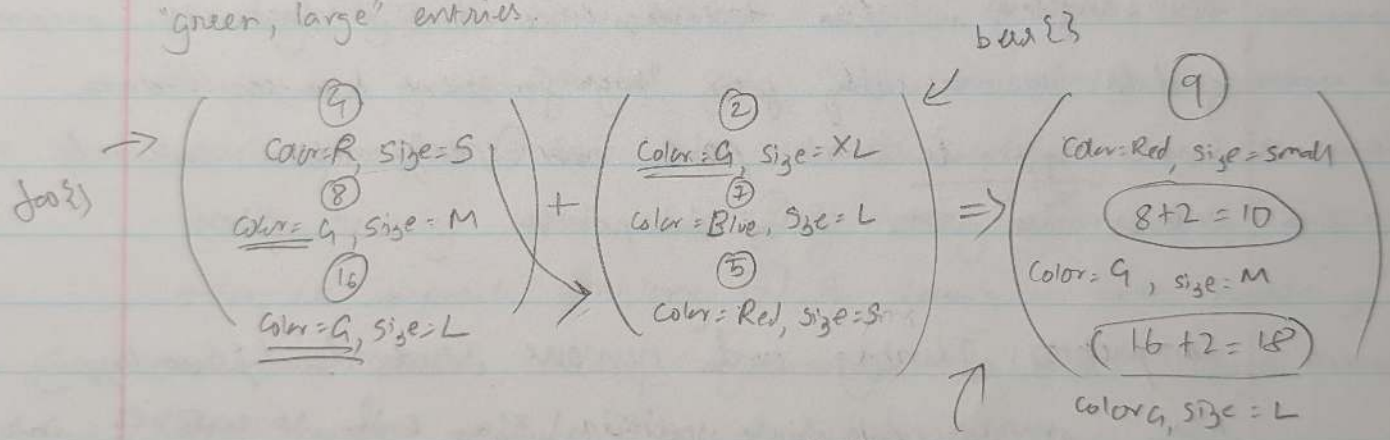
→ One-to-one matching on (label-subset)

foo{} + on (color) bar{} ⇒ new_vector{color} = value

OR foo{} + ignoring (size) bar{} ⇒ new_vector{color} = value

→ One-to-Many & many-to-One

with 'group_left', operator will try to find the matching element from the right side for both "green, medium" and "green, large" entries.



foo{} + on (color) group_left bar{} [as we are only grouping left side]

→ Many to Many

AND: creates a vector with elements from the left, that match with AT LEAST ONE element on the right.

UNLESS: creates a vector with elements from left, that don't match any elements on the right.

OR: creates a vector with ALL elements from the left and complementary elements from the right.

× — × — × — × —

* Sharding, downsampling, compaction, hashing and Long Term Storage in Thanos.

Sharding is distribution and storage of Prometheus time series data across multiple storage nodes. This helps with querying massive amounts of time series data, by breaking it down in smaller chunks.

Downsampling is the technique used to reduce the amount of data. we usually fetch high-frequency time-series data. So we aggregate data points over specific time intervals. This reduces the total no. of data points.

Compaction: Identify and remove redundant/overlapping data. Also, merge data points within the same time-series into a single aggregated data-point. Params of relevancy (compaction window, aggregation function, retention policy)

Hashing: If two data points are hashed and result in the same hash value, they are duplicate. Helps data-deduplication. Also helps in indexing, as they can map unique identifiers to storage locations. Then it becomes easier to retrieve data. Also helps spread data out to different storage buckets. Can also be used to verify data integrity. Can also determine which shard or partition of data, a specific time-series / data point belongs to.

REDIS STUFF

need to buy keyboard cover. buy it on AMAZON!

REDIS Outage and Recovery

- Redis is used as a in memory data store for speed sensitive transactions.

- Scalability Limitations

→ Redis is single threaded, which is great for speed but might cause issues when the workload is high.

→ Since this is an architectural limitation, horizontal scaling is the only available approach.

→ Why does REDIS only use a single thread? ① Because using a single thread reduces the complexity of managing access and handling synchronization between multiple threads, and processes. ② We also avoid thread sync tasks like locks, semaphores & atomic operations. ③ REDIS can also take advantage of CPU cache locality, reducing cache misses. ④ By processing commands in a serial order, we improve consistency. ⑤ By having a large number of concurrent clients, having asynchronous I/O and a single thread helps handle multiple concurrent connections.

→ You can have multiple REPLICAS, but it's not fun when a replica goes down.

- Memory Issues

→ As an in-memory storage, it can run out of memory, during its snapshotting phase.

→ Everytime you back your REDIS data, the snapshotting process is launched in the background.

→ Memory goes up if data backup is taking place during heavy WRITE activity

→ This can also happen if you scale up during a period of heavy write.

- Data Inconsistency

Consistency means that the data between the master and replica nodes of a cluster are always in sync.

→ we run into a problem : when data gets written to a 'master' redis node, it sends back a message saying that the WRITE was successful, before that (WRITE has been propagated to its Replicas.)

→ If something happens to the master node during this whole process, the replica gets promoted to master, and that latest data WRITE is lost forever.

→ 'SPLIT BRAIN' : A network partition failure splits redis nodes across two partitions, with master being in one partition and replica being in the other partition. The replica will then be promoted to a master. Once this happens, it is possible that different data is now being written to two different master nodes, causing them to get out of sync.

- Replication loops

→ as the database gets larger, it takes longer to copy data from master to the replica.

→ The data gets into a buffer. If the buffer runs out before the copying is done, data copy can fail.

→ This means that master and replicas are out of sync.
→ But REDIS relies on a check that the master and replica are in sync before the next update step, this means that the condition will never be met.
→ Redis deals with this fact by restarting the whole replication process. - now you have an infinite loop!
→ The only solution is to assign 3x the memory.

— : Memory management best practices : —

Memory fragmentation: this happens when operating system allocates memory pages which Redis cannot fully utilize after repeated write and delete operations. The accumulation of such pages can result in system running out of memory.

setting 'activedefrag' config helps, but comes with a CPU tradeoff. check mem usage ratio before fiddling with this config.

→ You can get out of memory errors, even though there is free memory available, and this could be due to fragmentation issues.
→ MEMORY DEFRAG: attempts to defragment by allocating new memory pages and copying data from the fragmented pages to the newly allocated pages.
→ other tips:
1) appropriate eviction policies 3) TTL values
2) appropriate data structures 4) Defrag.

- BGSAVE
  use copy-on-write, can double the space requirements.

- Scaling (Redis in K8s)
  → Each redis pod is either a master or a replica.
  If one replica fails, the others can continue serving requests.
  → Each shard operates as an independent REDIS cluster
  In K8s, this would be separate deployments or stateful sets, run is how you scale horizontally by scaling out the read & write operations.
  → We use external load balancers to distribut the incoming load, across different pods.

What happens when the pod dies:

- Pod gets terminated, clients trying to connect to that pod will get connection errors and timeouts. Replicas helps with this scenario.
- Writes that are not replicated, might be lost.
- K8s detects this and schedules another pod on a node. New pod (redis instance) starts resynchronization.
- If a <u>master</u> pod dies, the replicas detect this, and an election for a new master is initiated, amongst the remaining replicas.

The remaining replicas sync their data with the new master, to ensure consistency. The new master replays any missing write operations, to catch up with the state of the failed master.

: Pulsar callouts

1) DG online services use Pulsar as the primary backbone for facilitating DG calls between services.

2) DG services were setting 'producerDeduplication' flag, which is really expensive to execute, during restarts of a high no. of producers.

3) Poorly tuned backoff mechanism, designed to limit overloading reads to the pulsar storage tier, kicked in and lowered the read rates.

Userproxy queues are different for EU -
US -
APAC -

[Userproxy has a threshold of 2000 connection]

~~X 1.0.2: ... has not signed off on it. X~~

~~X 1.0.2 - X~~

- If a server gets rebooted, then the change is gone.
pushing the stress test to noon.

- if the number of partitions >
- If services are not coming up, make sure consul agents are up first.
- Bouncing consul server with a lock resolves the issues

Persists take a long time to come up because they have a database contention with cassandra.

To change partition ownership without restarting the pods, we can ~~the~~ invalidate the session on the lock in consul:

Key/value → / // → Lock / Partition...

∴ We can only have one core cluster w3 consul KV store

- Why is the retry rate high early + then goes down?

each partition service can only live in a single consul cluster because there is no global KV store.

...bal consul need
...ax latency of 100ms
... avg latency of 50ms

TCMalloc FreeCheckMemoryThreshold

Stalled game, Rubber banding SLO.

What adds to the memory loads = game or player behavior?

→ farm.v1 log message :("Rubber banding" OR "Stalled")
loge_level = or C

↳ could be a CPU issue, cuz CPU was not.
The CPU was not because we stuffed the game servers with a lot of bots.

- when there are resource utilization issues, the pod configuration only indicate the values that are being set, but not necessarily the reason why the CPU/mem usage is high. For those kinds of issues, always look at the application behavior. That is going to have the reason.

- Token validation API,
↳ Consul validation takes place on each pod

Restoring the consul server works for resource clearance because the when each client issues a Token validation, there is a rate limit in ~~the~~ that is applied the API calls, but a lot of clients overwhelms

Q Can we put labels on the node pool via GKE?

- The only way to implement this is on the 'google_container_cluster' block and not
  ↳ google_container_node_pool (X Does not work on this)
- node_config.labels and GCP node pool labels are different.

-: Lessons from Launch :- 1/7/2023

- In game shop access: how long should players need to wait to get into the shop.
  - It tends to load slow when a lot of players are trying to access the shop

- Queued Player rate high SLO
  - How much should it be, whats acceptable?

- Matchmaking wait times
- look into why USW1 & USW4 are so high in
- matchmaking wait time.
- How many RTT's are we taking.

- Ping service tests latency for the clients.
- Get ping

is it N/w latency?

- Login rate is low, because the players in queue, the ones that

use mtr to figure this out.
To do that; Spin up a temporary pod with N/w tools; kubectl run tmp-shell --rm -i --tty --overrides = '{"spec": {"hostNetwork": true}}' --image nicolaka/netshoot.

→

- Unscheduled Pods
  - hpa increase trigger more pods in an unscheduled state
  - more nodes
  - ↳ that's why auto scaler kicks up

- ERROR CODE: 34202 went up by 38716, and the reason could be that people are trying to access the game without having the ultimate edition.

- How our server pod's performance are compared to no. of players / game server.
  → make sure we are not underprovisioned.

- Exceptions from game-server pods were not being caught: goal: variable reduction.
  1) get the pod_name
  2) Look at the logs to see when the server crash mailer issued the request.
  3) If we don't have a error response, that means that our code is correct.
  4) The next step is to dive on the exceptions side.

11:20 UTC / 4:20 am PDT

Goal:

Triage steps:

1) check connectivity
2) check logs
3) go into the pod and run the UUID gen command,
4) nc -v iir.blizzard.com 3724
   → this confirms that we have connection
5) Reach out to networking to see the logs on the loadbalancer (iir.blizzard.com), there should be traffic sent from the pod/node,
6) Obtain the source IP: (IP of the node)
   Obtain the destination IP: (IP of the exception LB)
   to get the address of iir.blizzard.com, use:
   getent ahosts iir.blizzard.com
   ↳ slightmaster
7) we could crash a ping server, so that it does not
8) 24.105.29.24 → AIO →
   figure out the NAT IP address of the GCP node:

Path

10.123.97.2 ⟶ 35.226.97.29 →
GCP NAT

9) Test out the curl functionality
   1) create a dummy zip file
OR 2) create a dummy XML
   3) CURL a -X POST
   4) user agent. "FerrisServerCrashMailer/2.0 (linux)"
↳ we expect to get a 302 followed by a 404
   5) adding /v2/submit?response=true

works

```
curl -o /dev/null -s -w "%{http_code}\n"
```

we can rule out the issue was A10 because in our response, we see server_ID, which means we are hitting the right.

10) we then went about sending multiple guids for exceptions folks to check

11) we then try to crash a service 7 builds in devs to see if there are code changes to the server crash mw

40177
41650 ✓
41851 till 05/26th.
works 06/02
4210

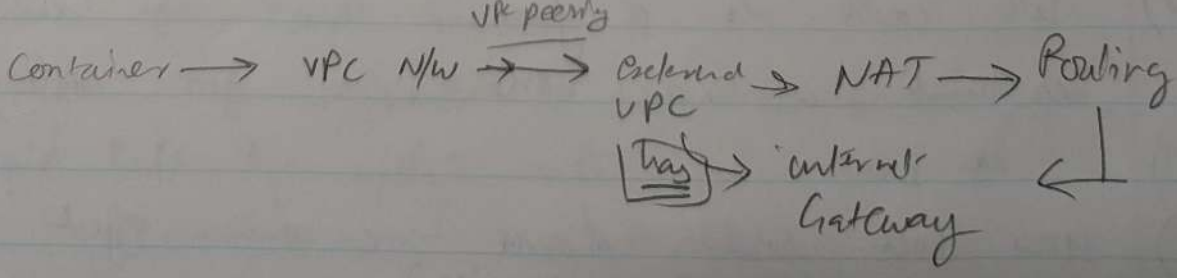
1) How to roll out a change without killing the Pod?
- warm up
- daemon set that caches images and uses imagePull policy

gcloud compute addresses list | grep NAT_AUTO
↳ this gets you a list of all the IPs for a given region.

Journey of a packet from a Pod to a destination IP outside a private GKE Cluster.

1) Packet is sent to a pod's network namespace. It is then encapsulated into a container specific network interface, that is within the pod.
2) Pod IP table: packet then routed based on pod's IP table config.
3) If there is no overlay network, the packet then goes to the Node's interface.

NODE level.



Red/Blue D4

- We could scale up the core site before we scale the game servers
- We have the red/blue/green deployment Alpha, so the logic exists.
- Copy of the application set, and

This is to segregate the

1) Two different values file within the same application - separate value file for color.
2) Both will point to same pulsar, different events.
3) We throw only a 100k of players.
4) What is the impact on pulsar by doubling the core services, we don't know our behaviour of core service scaling down.
5) We may have the IP space & Quota.
6) Client behavior is not a concern - Client is unaware of the server side.
7) | Job, Consul, search | - the newest version of the color handles both data, so needs the same consul authority
8) We could do a focused boot on the upside so that a new version can get rolling update.
9) We want
10) How does scaling down core-services, affect other dependencies services?

CONSUL
STUFF

Fastest change
→ give more memory to the consul-clients, we do this by changing the chart resource limits

Game Core Regional
game server mem: 32Gig
X ____ X ____ X ____

Consul ARCHITECTURE Notsuall.

1) Recommended deploying 5 nodes within the Consul cluster distributed between 3 AZ. This can withstand the loss of two nodes or an entire AZ.
2) They all run RAFT driven consistent state store for updating catalog, session, prepared query, ACL & KV state
3) GCP Disk size & IOPs achieved.
   512 GB Pd-balanced ⇒ 15000 IOPS @ 240MB/
   1000 GB Pd-SSD ⇒ 30 000 IOPS @ 480 MB/
4) Avoid "burstable" CPU & storage options.
5) Consul Server uses consensus protocol to process all write and read ops. Server is I/O bound for writes & CPU bound for reads.
6) High IOPs is needed on the disk because of the rapid Raft log update rate.
   ↑ This is raft log replication.

7) The latency threshold for Raft is calculated by total RTT between all the agents.
for data between all consul agents.
Avg RTT < 50ms
RTT for p99 < 100ms

8) Both: client & server participate in Gossip.

9)
RPC [8300 TCP] client/server → server, handle incoming req from other agents.
Serf LAN [8301 TCP/UDP] all ⇔ all, gossip
Serf WAN [8302 TCP/UDP] server ⇔ server, gossip over WAN
HTTP/HTTPS [8500, 8501 TCP] localhost → server, HTTP API
DNS [8600 TCP/UDP], localhost client → server, DNS resolution
gRPC (optional) [8502 TCP] client server client
sidecar Proxy (optional) [21000 - 21255 TCP]

10) In service mesh, mutual TLS is implemented by sidecar proxies to encrypt and authenticate service to service comms.
Service mesh CA issues TLS certs for each service & client agents

11) more client agents = more time for gossip to converge.

12) when a new agent joins an existing large datacenter, with a large kv store, it may take more time to replicate the store with new server logs and update rate may increase

13) How fast are nodes joining/leaving/failing: large spikes or persistent gossip churn. stress the system more than large no. of Consul Catalog services or high kv read rate.

CPU Resource Utilization in K8s.

1) CPU Requests :
   - Min amount of CPU resources that a container needs to run.
   - Value represents the gaurantud share of CPU time that the container should receive when the system is under contention.

2) CPU Limits :
   - MAX amount of CPU resources that a container is allowed to consume.
   - If container exceeds this, kernel throttles the container on CPU.

3) cpu.cfs_period_us = 100 000 micro seconds (100 ms) by default.

which means : Scheduling period for CPU allocation = 100 ms.

4) cpu.cfs_quota_us = CPU Limit specified for a container

cpu.cfs_quota_us = CPU limit × cpu.cfs_period_us

∴ if CPU Limit = 0.5 or 500m CPU

cpu.cfs_quota_us = 0.5 × 100,000

$= \frac{5}{10} \times 100,000 = 50,000$ ↑ micro seconds

if : for a 100 ms period, container allowed to consume CPU resources for 50 ms.

Core: Independent execution unit capable of executing instructions

Thread: sequence of instructions that can be executed.

Threads are scheduled and executed ON cores.

- Some systems support multithreading: each core handles multiple threads.

1) Incoming request arrives.
2) K8s scheduling: app container is scheduled on to the node that has the CPU & memory.
3) Core allocation: K8s assigns a specific core from the node, for execution.
4) Thread allocation: The specific core handles multiple threads.
5) Req Handling: Container running on the assigned core utilizes the available threads; all app specific logic gets handled within the thread. CPU usage is monitored.
6) If core supports hyper threading, multiple threads are processed in parallel.

Consul commonly fails

1) If the rate of agent join/leave is high, or high K/V usage, it's bound by disk I/O because the underlying RAFT log store performs a sync to disk every time an entry is appended.

2) Gossip protocol used to manage group memberships of the cluster and to send broadcast messages, through the serf library. Consul uses LAN gossip pool & WAN gossip pool.

- LAN Gossip pool: This has all the clients & servers in the datacenter. Membership info provided by the LAN pool allows clients to automatically discover servers, reducing the amount of config. It also shares failure detection, and fast + reliable event broadcasts.

- WAN Gossip pool: It is globally unique. ALL servers needs to participate in WAN pool. Membership info allow servers to do cross data center re

* If serf Health check flaps, the agents might be CPU throttled or N/W exhausted.

3) Consensus: Consul uses RAFT consensus algorithm -
allows nodes to be in — LEADER, Only SERVER agents
Follower
CANDIDATE use RAFT consensus

Client only forward requests to the server
RAFT uses RPC to make comms between client & server. Basically Raft is used to:
↳ respond to client requests
↳ replicate information (logs) between servers.
basically, a) leader gets a value, it notifies all followers
b) then it commits the entry, & notifies all followers
c) followers also commit the entry
d) log replication is successful.

Leader Election:
1) Nodes have a election timeout, random between 150 ms & 300 ms.
2) After election timeout, follower becomes a candidate, new election has begun, it votes for itself.
3) Candidate sends vote-requests to other nodes.
4) If receiving node hasn't voted yet, it votes for the candidate, election timeout is reset.
5) Node with most votes becomes a leader.
6) Leader sends 'Append entries' to followers. The follow respond back. (↳ heartbeats) Election term continues until follower stops receiving heartbeats & becomes candidate.
The election timeout keeps getting reset.

Append entries can be clean heartbeats or state change messages.

Once a log entry is written to durable storage disk by Consul leader, it attempts to replicate it across all members of the quorum (followers). Once followers respond back that they have replicated the log entry, leader commits the entry, tells the followers that it has committed it. Once committed, it can be applied to the Finite State Machine. FSM is application specific, MemDB for Consul. Consul writes are blocked until log entry is both committed & applied.

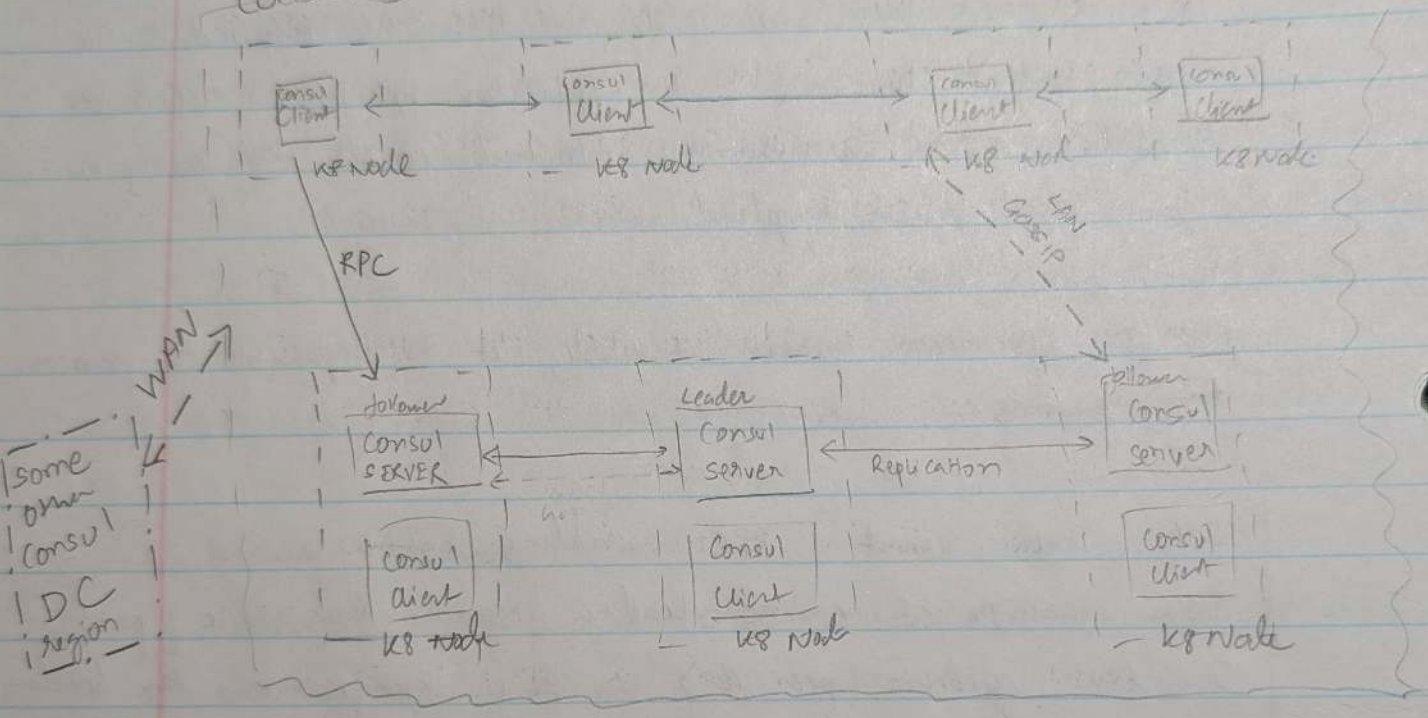
* If quorum is unavailable, it is impossible to process log entries

Lets talk about RPC handling

1) RPC arrives at a non-leader. Non-leaders are part of the quorum (peer set) so they know who the leader is. They forward the req to the leader.
2) If RPC = Query type, its read-only, leader generates result based on FSM's current state
3) If RPC is transaction type, ie, it modifies the state (eg new node added), leader generates a new log entry and applies it using Raft., once entry is committed & FSM is updated (applied), transaction is complete.

Consul on K8s

1) Consul SERVER agents are responsible for datacenter state, responding to RPC queries and processing all write operations.



2) Agents communicate over LAN Gossip.
servers participate in RAFT consensus
Client Requests are forwarded to servers using RPCs.

3) Multiple Consul DC's can be joined by WAN links.

4) Catalog sync allows you to sync services between consul and kubernetes. services synd are discoverable with the built in kubernetes DNS.

→ 3) Why even do this? K8's services synced to Consul catalog enable K8's services to accessed by any node that's part of the Consul cluster.

this is different from service MESH!

5) WAN federation of consul clusters allows you to deploy the same services in different data center locations or cloud regions, and discover services across different regions.

6) Consul servers operate independently & will only return results local to their DC.

7) WAN federation (basic) is over TCP/UDP 8302 (WAN Gossip) & TCP 8300 (Remote RPC)

8) All consul clusters must be connected as full service mesh; ie, all servers must be able to comms over RPC & Gossip.

9) Each consul cluster maintains its own LAN Gossip.

10) All servers in a federated DC must use RPC certificates signed by the same CA. All DC names need to be referenced in the SAN cert.

11) First consul cluster created is designated as primary and is authority for some global state. (ACL, Intentions, Connect-CA)

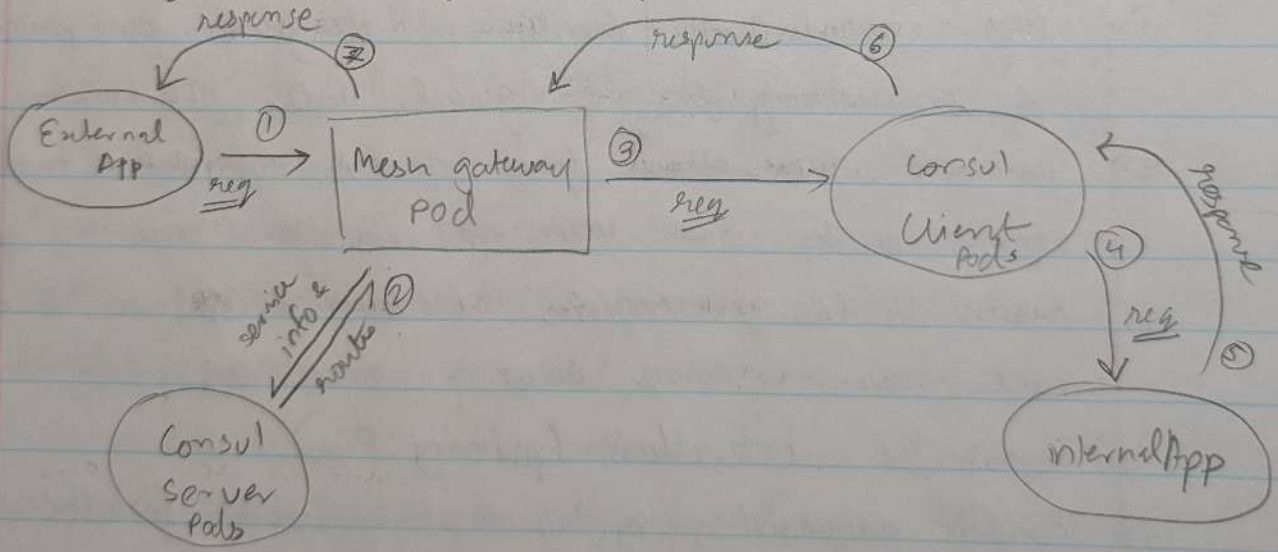
12) Advanced WAN allows for a hub-&-spoke model of comms, which is better for a large number of DCs across a lot of regions. Because then you do not need constant comms between each DC. Everyone talks via the hub-cluster (primary DC)

13) consul's prepared query (help implement failover policies, which have static list of alternate DCs, next-best DCs etc) allows failover to a different consul cluster for service discovery.

14) Service Mesh Gateway enables communication between services outside consul DC and services inside consul DC.

How the mesh works:

1) Consul service mesh: Collection of proxies deployed alongside services. They handle the traffic and offer load balancing, observability & service discovery.
2) You deploy mesh-gateway pods alongside your 'external services'
3) External service sends request, which hits the mesh gateway pod, and that pod decides if it needs to send it over to a service within the Consul cluster.
4) The mesh gateway pods talk to the consul server pods to obtain service information and use that info to route traffic between external services and services within the consul service mesh, and leverage the service discovery and other features of consul control plane.



5) To establish a connection to the pod using service mesh, a client must connect to the client mesh proxy to do service discovery, to find other mesh proxies in the target DC.

Consul Server Performance

1) Consul servers run a consensus protocol to process all write operations and are contacted on nearly all read ops.
2) servers are I/O bound as Raft writes its log to disk first.
3) servers are CPU bound for reads since they work from fully from the in-memory store, optimized for concurrent access.
4) Performance can be tuned using 'raft_multiplier'. It is a scaling factor, affecting the following parameters.

HeartbeatTimeout	= 1000 ms	e.g. if raft_multiplier = 5
ElectionTimeout	= 1000 ms	=> 1000 × 5 = 5000 ms
LeaderLeaseTimeout	= 500 ms	1000 × 5 = 5000 ms
		500 × 5 = 1000 ms

5) Short multiplier reduces failure detection & election time but may be triggered frequently.
6) Large multiplier reduces the chances of spurious failures, but takes longer to detect real failures.
7) Spurious leader elections can be caused due to N/w issues or unsufficient CPU.
8) For apps that perform high volume of reads against consul, using stale consistency mode would allow reads from all the servers, not just leaders.
9) There is a new config param on the client called 'limits' that you can use to limit the no. of RPC req/sec rate. After hitting the limit, requests will start to return rate limit errors.

10) What gets synced on the Disk by the servers?

a) K/V entries
b) service catalog
c) prepared queries
d) ACL
e) Sessions in memory

} as a snapshot + log of changes on the disk

11) Memory determination:

Also: consul.runtime.heap_objects

First, determine the value of working set size:
consul.runtime.alloc_bytes (it is telemetry)

RAM minimum => 3x or 4x that value

second, KV pair size also dictates memory:
RAM needed = number of key x avg key size x 3

12) Disk determination:

writes need to be synced on a quorum of servers before they are comitted, so use an SSD!!

13) Allow stale DNS so any server can respond back with a stale result

14) If memory = full or Disk = full, server stops responding, loses its election, cannot move past the last commit spot. setting 'max-stale' allows for reponses to continue on, even in an outage

15) Increase the 'ulimit' from default (1024) to prevent leader from running out of file descriptors.

16) Also note, that if you swap the CA cert, in a service mesh, all services gonna hit your servers with the new cert signing vib make sure to control that: csr_max_per_second.

Multiple D[redacted] Gossip

1) By default, each consul cluster (datacenter) runs independently, each having a dedicated group of servers, and a private LAN gossip protocol.
2) To check the members in a WAN, : consul members -wan
3) Client requests are forwarded by local-servers to the destination consul datacenter server.
4) Since consul 0.8.0, WAN join flooding enables for one consul server in a DC to join the WAN and all other local servers within that DC to know about this via LAN Gossip.
5) To check federation DC participation:
   curl http://localhost:8500/v1/catalog/datacenters
6) For service discovery across regions, networks must be able to route traffic between IP addresses across regions, use VPN tunnels and firewall rules to ensure this.
7) For RPC calls, the IP address /N/w interface address for a given service (aka bind address), needs to be reachable accross datacenters/regions. The service initiating the req. connects to the bind address of the target service. Gotta set bind_addr to a public address or a private addr with VPN/FW rules sorted out.

Multiple Datacenter Federation using WAN Gossip

1) By default, each consul cluster (datacenter) runs 'independently', each having a dedicated group of servers, and a private LAN gossip protocol.
2) To check the members in a WAN, : consul members -wan
3) Client requests are forwarded by local-servers to the destination consul datacenter server.
4) Since consul 0.8.0, WAN join flooding enables for one consul server in a DC to join the WAN and all other local servers within that DC to know about this via LAN Gossip.
5) To check federation DC participation:
   curl http://localhost:8500/v1/catalog/datacenters
6) For service discovery across regions, network must be able to route traffic between IP addresses across regions, use VPN tunnels and firewall rules to ensure this.
7) For RPC calls, the IP address /N/w interface address for a given service (aka bind address), needs to be reacha accross datacenters/regions. The service initiating the re connects to the bind address of the target service. Gotta set bind_addr to a public address or a priv addr with VPN/FW rules sorted out.

K8's STUFF

Setting Pod Mem (Req & Limits)

1) If limit > req, you're allowing pods to potentially ask for more memory than they are gauranteed.
2) This ends up in overutilization of kubernetes nodes.
3) This is bad because if ya take more mem (upto the limit), other pods, who might need their req memory cannot find it. The sum of memory limits of all pods exceeds the available memory on the node itself. So:
4) other pods get OOM killed, because they cannot find the memory upto the limit they are configured to burst up to.
5) OOM killer, kills pods to free up the memory, by selecting a process on the pod(container)
6) Pods with limit > req = Burstable QoS
   limit = req => Gauranteed QoS
   limit = null & req = null => Best-Effort QoS
7) After OOM killer kills a pod, it also helps schedule it to a new node with available memory.
8) If mem limit = mem req = low, it only kills that pod and does not affect other deployments
7) Once you give memory, you can only take it back by killing the process that has it.
8) Caveats: Page Cache behaviour (memory pages = fixed length contiguous block of virtual memory; caches: frequently accessed disk stuff stored in RAM)

If a process requests memory & there is pressure to free it up on the node, OS targets application pages rather than Page-cache.

```
spec:
  containers:
    volumeMounts:
      - name: secret-volume
        mountPath: /etc/myapp
  volumes:
    - name: secret-volume
      secret:
        secretName: kube-secret-name
```

d) ConfigMap
data:
secret-key

9) Affinity rules: pods can be pending despite having available memory. Its a good idea to giving dedicated node pods for certain workloads

10) CPU pinning also helps to avoid pods jumping between different CPU's on the same nodes.

--cpu-manager-policy=static

- CPU allocation to pods is a fixed amount, remains constant.
- Strict resource isolation, each container gets a gaurenteed CPU.
- Disadvantages :- If pods dont use allocated CPU, it get wasted.
  - Doesn't help when CPU usage is spiky
  - Bad to support other pods that might need extra, also hard to scale efficiently.

11) How to tell an OOM kill

a) Error code 137
b) kubectl get pods, OOMkilled in the status
c) command terminated with exit code 137
d) Host-OomKillDetected & KubernetsContainerOomkiller in Prometheus Alerts
e) Pods suddenly drops connections

12) How to reference or wedge in a kube secret value :

a) have a kubesecret created, its going to have data: key: value (b64enc) somekey: value

b) Referencing using ENV :

```
spec -> containers ->
env:
  - name: SOME_NAME
    valueFrom:
      secretKeyRef:
        name: kube-secret-name
        key: somekey
```

c) mount as a File :

look at the top

Admission Webhooks in K8

1) HTTP callbacks that receive admission requests and do something with them
2) Mutating Admission webhooks are invoked first and can modify objects sent to the API server to enforce custom defaults.
3) Validating Admission webhooks are invoked AFTER all object modifications are complete and after incoming object is validated by the API server.

Kube Admission Controllers

1) They help implement security features such as pod security policies that enforce a security baseline across an entire namespace.
2) Act as gatekeeper that intercept : authenticated requests to the API server and may change the request object or deny the request.
3) They have a mutating phase & a validating phase.
4) The two admission controller webhooks referenced above do not implement any policy decision themselves. The action is obtained from a REST endpoint of a service running inside the cluster.
5) Example: - PodSecurityPolicy (helps disable root on containers)
   - disallowing: 'privileged: false'
   - label validation
   - automatically add annotations
   - add resource limits, validate resource limits.

You can also set taints based on conditions.

K8 Taints & Tolerations

1) Taints are a property of nodes that enable them to repel a set of pods.
2) Tolerations are applied to pods. They allow the scheduler to schedule pods with matching taints.
3) They allow scheduling but DO NOT gaurantee it.
4) When a taint is applied to a node, basically the node will NOT accept any pod that does not tolerate that taint.

5) Lets say a node has:

taint:

key = "key1"
value = "value1"
effect = "NoSchedule"

for a pod to be scheduled on that node, it needs

in PodSpec →

```
tolerations:
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoSchedule"
```

```
tolerations:
- key: "key1"
  operator: "Exists"
  effect: "NoSchedule"
```

effects: 1) NoSchedule - Dont put pods on node, but let old ones run
2) NoExecute - Evict all pods from node that dont tolerate (can set tolerationSeconds)
3) PreferNoSchedule - Prefer to not schedule pods on that node.

GKE HPA

1) Scales pods up or down based on an evaluation.
2) This can be CPU (raw or %), memory (raw or %) or custom metrics (raw or average).
3) Example: If using a message bus, you can create an external metric called: queue size, you can choose to scale the pod up or down based on queue size.
4) There is a dedicated HPA for each workload type. It periodically checks for the metric against the target value in a control loop.
5) For per-pod resource allocation, the hpa checks for metrics (via metrics API) for each container.
6) Thrashing: when HPA attempts to perform subsequent autoscaling actions before pods finish their prior autoscaling actions. To avoid this, hpa uses largest value in the last 5 minutes.
7) DO NOT use hpa & vpa together for CPU & mem.
8) DO NOT configure hpa on ReplicaSet or Replication Controller just do it on Deployment itself. When you do a rolling update, it replaces the replication controller with a new one.
9) HPA DO NOT work on Daemonsets.
10) DO NOT Enable .spec.replica when you have hpa enabled, as if you do a kubectl apply on the manifest with the spec.replica value, the controller will scale that deployment to that particular value, and NOT the hpa value.

Scaling behaviour

1) You can use the behaviour field to configure separate scale up and scale-down behaviours.
2) You can use a stabilisation window that prevents flapping (same as thrashing). This also helps you control the rate of change of replicas while scaling
3) When multiple policies are specified, the one with the most amount of change is selected by default.
4) behavior:

scaleDown:

policies:

- type: Pods
  value: 4
  periodSeconds: 60

allows 4 replicas to be scaled down in 1 minute

- type: Percent
  value: 10
  periodSeconds: 60

length of time in the past, for which the policy must hold true

% of current replica.

allows at most 10% of the replicas to be scaled down in one minute. Only goes into effect if no. of replica > 40

5) If current replica = 80, and needs to be reduced to 10!

Step 1: 10% of 80 $\Rightarrow \frac{10}{100} \times 80 = 8$, reduce by 8 pods

Step 2: Current: 72; 10% of 72 = 7.2 ~ 8, reduce by 8 pods

Step 3: Current: 64; 10% of 64 = 6.4 ~ 7, reduce by 7 pods

Step 4: Current: 57, 10% of 57 = 5.7 ~ 6, reduce by 6 pods

Step 5: Current: 51, 10% of 51 = 5.1 ~ 6, reduce by 6 pods

Step 6: Current: 45, 10% of 45 = 4.5 ~ 5, reduce by 5 pods

Step 7: Current: 40, Policy no. (1) kicks in, reduce by 36 pods.

Current: 4 pods.

6) You can pick a policy by selecting 'selectPolicy'

7) Stabilisation Window:
(good to have to avoid flapping or you expect late load spikes)

behavior:

scaleDown:

stabilizationWindowSeconds: 300 ← its 5 mins by default

5 mins.

it's the wait time enforced after each scaling event

when the metric indicate that the target should be scaled down, algorithm looks at previously computed values. The time window that it 'considers' to code at these values (and pick the max value within the window).

8) Prescriptive scaling behaviour:

a) Apps that handle business critical web traffic:
- Scale up fast as possible
- Scale Down really slowly

b) Apps that process critical data:
- scale up fast as possible to reduce data processing time.
- scale down fast as possible to reduce cost.

c) Apps that process regular web-traffic
- Regular scale up/scale Down to minimize jitters.

d) Game workloads - It depends on player pacing per pod, and time to move players from one pod to the other pod before shutting the pod down.

9) Non modifiable vars on the actual hpa code:
- scaleUpLimitFactor = 2.0
- scaleUpLimitMinimum = 4.0

} determines how fast can a target scale up

10) If you want to scale down/up 1 pod every 10 mins

policies:
- type = Pods
  value: 2
  periodSeconds: 600

∴ If you want to scale down n pods every t seconds:

type = Pods
value = n
periodSeconds = t seconds

11) If you want to DISABLE scale down,

behavior:
  scaleDown:
    selectPolicy: Disabled

12) Working of the stabilization window:
stabilizationWindowSeconds : 600 (10 minutes)
current replica = 10, hpa controller cycles per minute.

9 recommendations → 10 recommendations for the 9 minutes: recommendations = [10, 9, 9, 8, 7, 8, 7, 8, 8] ← MAX

→ at 10th minute: recommendation = [10, 9, 9, 8, 7, 8, 7, 8, 8, 7] ← picked up MAX.

→ at 11th minute: recommendation = [9, 9, 8, 7, 8, 7, 8, 8, 7, 7] ← added
(10 removed; 9 = max)

∴ replica ⇒ 10 → 9 = 9 ← new replica count.

Scale Down => Pick largest value of desired replica
Scale Up => Pick smallest value of desired replica.

General flow:
1) gather recommendations for that window
2) pick the largest value/smallest value from step (1).
3) scale down/up no more than n pods or x % of value (2) if policy sustained for t time.
↑ upper bound.

13) if policy.type = pods
current Replica ± policy.Value
if policy.type = percent
current Replica × (1 ± policy.Value / 100)
(+) for desired > current
(−) for desired < current

14) → ScaleDown.stabilizationWindowSeconds = 300 ~ 5 mins,
wait for 5 mins, get largest value, scale down to that.
→ if windowSeconds = 0, DO NOT gather recommendations, just scale.

15) scaleUp.policies
→ policy = percent
periodSeconds: 60
value = 100, no. of replicas can be doubled every minute
→ policy = pod
periodSeconds = 60
value = 4, means 4 replicas can be added every min.

16) Implicit maintenance-mode: if you set hpaMax = 0 } hpaMin > 0 }, hpa STOPS adjusting the target.

Removal of spec.replicas value:

1) The default value is still 1, even when you remove it.
2) So, with the new manifest update, you will still have 1 remaining pod while the others terminate.
3) To avoid this:

→ Client-side Apply

a) kubectl apply edit-last-applied deployment/<deployment_name>
b) edit the manifest, save & exit. Kubectl applies this change BUT, no changes to the pod can't takes place.
c) When you want the changes to get "properly applied", do kubectl apply -f deployment.yaml

→ Server Side Apply

a) This uses a more declarative approach, tracking a user's field management rather than last-applied state.
b) The user who last made the assertion about the value of the field, will be recorded by the current field manager.
c) You can do a POST, PUT or PATCH against a server side apply endpoint.
d) The fields themselves are stored: metadata.managedFields
e) for example
   .) kubectl apply -f <somejaml> --server-side
   .) Enable HPA: kubectl autoscale deployment <name> --cpu-percent=50 --min=1 --max=10
   .)

Now, that replica value is going to cause a bit of a race condition. You can:

a) Leave the replica value in the config. HPA eventually writes to that field, there will be a conflict, and at that point, you can take it out of the config. (Gotta WAIT)

b) To not have to deal with that race-condition:

1) Define a new config, with only replica value.

2) kubectl apply -f <file_from_rep-1.yaml>

--server-side --field-manager=handover-to-hpa

--validate=false

3) Let it happen. Even if it conflicts, it will not

Server side is better than client side because:

In client side Apply, outdated values overwritten by a user are left in the appliers local config. These only become accurate when the user updates the specified field.

Also, on the client side apply, you cannot change the API version.

GKE Cluster Auto Scaling

1) You set the min & max node counts, autoscaler scales the node up or down.
2) If resources are moved or deleted, eg: if your workload consists of a controller with a single replica, it may be moved to a different node as part of scaling (node deletion).
3) If that pod (which is getting moved) is critical, you gotta add taint to the node and toleration to the pod spec.
4) Autoscaler works by adding or removing VMs from the underlying GCP Compute Engine Managed Instance Groups (MIG), for the node-pool.
5) Scaling is based on resource request, NOT resource Utilization.
6) Scenarios
   a) If not enough nodes in the pool for pods to get scheduled, add more nodes.
   b) Nodes under-utilized and all pods can be scheduled with fewer nodes, remove nodes.
   c) Pods tolerating the taints, remain, no node deletion.
   d) If pods can be moved to other nodes, but node cant be drained gracefully, NUKE that NODE, after the timeout.
   e) That Node Termination grace period is Static. Its always 10 minutes. The Pod Termination grace period is configurable.
7) Labels added to the node pool after cluster creation are not factored in.

8) currentNode < min, scale up to min, scale down disabled
min < currentNode < max, scale within range
current node > max, scale down to max, scale up disabled.

9) You NEVER scale down to 0.

10) GKE 1.21>, you cannot scale up from current = 0
GKE 1.22>, you CAN scale up from current = 0.

11) Autoscaling Profiles: balanced or optimize-utilization.

12) In the optimize-utilization mode, GKE sets the scheduler name in the Pod spec to: gke.io/optimize-utilization-scheduler

13) During scaledown, auto scaler respects:
- pods affinity-/anti affinity rules.
- in GKE control plane > v 1.22, pods with local storage no longer block scaling down.
- If pod has annotation: cluster-autoscaler.kubernetes.io/safe-to-evict: false
- Nodes will not scale up if PriorityClass < -10.
- Nodes might not scale up if they run out of IP space → eventResult event will give you a reason: scale.up.error.ip.space.exhausted

14)

Why would a Pod be ruled?

1) Hardware failure
2) Cluster Admin deletes the Pod / VM, by mistake
3) Cloud Provider / hypervisor failure, makes VM disappear
4) Kernel Panic (low level data error, kills the instance to stop corruption
5) Node goes away due to network partition
6) Eviction due to node being out of resources.
7) Deleting a deployment / pod manually
8) Updating the pod template, makes pod restart
9) Node draining (for repairs or cluster autoscaling)

Pod Disruption Budget

Limits the number of Pods of a replicated application that are down simultaneously from voluntary disruptions. For eg. for quorum based applications you can set up a budget where the number of replicas are never below the minimum number needed to maintain quorum.

- Involuntary evictions cannot be prevented by PDB but they do count towards the budget.
- Set 'AlwaysAllow' unhealthy Pod Eviction policy to PDB to support the eviction of misbehaving applications during a drain, otherwise node will wait for pod to be healthy before it can start draining.
- For StatefulSets, whatever pod that is being terminated, (it needs to be terminated completely.) will need to come back with the same name but different UID.

what if: metric = 0, hpa then picks the minimum no of replicas specified in the hpa.
If we have a stabilization window of n minutes, it will consider, current replicas for each minute, but will pick the max out of the list/array]

More thoughts on scaling:

1) Policy is just constraints as to how much is allowed during the period.
2) The actual replica count is determined by the HPA algorithm
3) Example:

$$\text{desiredReplicas} = \text{currentReplicas} \times 1.1$$

The above is derived from:

$$\text{desiredReplica} = \text{ceil}\left[\text{currentReplicas} \times \left(\frac{\text{currentMetricValue}}{\text{desiredMetricValue}}\right)\right]$$

If currentMetricValue = 88
desiredMetricValue = 80

$$\text{desiredReplica} = \text{ceil}\left[\text{currentreplica} \times \left(\frac{88}{80}\right)\right]$$

$$\searrow 1.1$$

The reason is because HPA wont scale if its close to 1 and it has a tolerance of 0.1

4) Example 2:
Target CPU Utilization: 50%.
current Replicas: 2

a) HPA controller queries the metrics API server to get current CPU Utilization.

current CPU Utilization = 40%.
which is lower than target.

$5\overline{)6}\ 1.2$ $\quad -5\ \ 1$ $\qquad$ $5\overline{)8}\ 1.6$ $\quad -5\ \ 30\ \ -30\ \ 0$

b) HPA decid

desired Replica(b) = ceil $\left(2 \times \frac{10}{50}\right) = \frac{\cancel{0}}{5\cancel{0}}$ ceil $= (1.67) = 2$

desired Replica(b) = 2 { No scaling action needed.
current Replica = 2 }

c) current CPU Utilization = 60%.

desired Replica(c) = ceil $\left(2 \times \frac{6\cancel{0}}{5\cancel{0}}\right) = \frac{12}{5} = \text{ceil}(2.4)$ $= 3$

desired Replica(c) = 3 { scale up to 3
current Replica(c) = 2 }

d) current CPU Utilization(d) = 20%.
current Replica(c) = 3

desired Replica(d) = ceil $\left(3 \times \frac{2\cancel{0}}{5\cancel{0}}\right) = \frac{6}{5} = \text{ceil}(1.2)$ $= 2$

scale down to 2.

— x — x — x — x — x — x —

Note about CPU.

1) Core: Processing unit that is capable of executing instructions.
No. of cores determines the maximum number of instructions that can be executed in parallel. CPU with 8 cores, can process 8 instructions simultaneously.

$$5\overline{)6}\ 1.2 \quad (5, 10, 10, 0) \qquad 5\overline{)12}\ 2.4 \quad (10, 20, 20, 0) \qquad 5\overline{)8}\ 1.6 \quad (5, 30, 30, 0)$$

b) HPA decides to scale up:

$$\text{desired Replica}_{(b)} = \text{ceil}\left(2 \times \frac{40}{50}\right) = \frac{80}{50} \overset{\text{ceil}}{=} (1.6) = 2$$

desired Replica(b) = 2 } No scaling action needed.
current Replica = 2 }

c) current CPU Utilization = 60%.

$$\text{desired Replica}_{(c)} = \text{ceil}\left(2 \times \frac{60}{50}\right) = \frac{12}{5} = \text{ceil}(2.4) = 3$$

desired Replica(c) = 3 } Scale up to 3
current Replica(b) = 2 }

d) current CPU Utilization(d) = 20%.
current Replica(c) = 3

$$\text{desired Replica}_{(d)} = \text{ceil}\left(3 \times \frac{20}{50}\right) = \frac{6}{5} = \text{ceil}(1.2) = 2$$

Scale down to 2.

— x — x — x — x — x — x —

Note about CPU.

1) Core: Processing unit that is capable of executing instructions. No. of cores determines the maximum number of instructions that can be executed in parallel. CPU with 8 cores, can process 8 instructions simultaneously.

CPU & Threads & CORES

If you want to get the no. of threads used by a PID, do: ps -o nlwp (pid) or top + press H

2) Thread: A sequence of instructions that can be executed independently of other threads
You can have a SINGLE thread per core OR you can have MULTIPLE threads per core.
Intel calls this - Hyper-Threading
AMD calls this - SMT

Execution => no. of cores X (no. of threads per core)

3) Frequency or clock speed is the RATE at which CPU's clock cycles. ~ maximum number of instructions that can be executed per second.

4) Cache size: small amount of high speed memory that is used to store frequently accessed data, so that we don't have to talk to RAM that much.
L1 - smallest (fastest) -> L2 -> L3 (largest (slowest)
% of memory accesses satisfied by cache -> Cache hit rate: measure of how effectively a cache is able to allow fast reads and storage of frequently accessed data.
We want this to be high (depends on access pattern and cache replacement policy

5) Memory Bandwidth: measure of the amount of data that can be transferred between the CPU and the memory in a given amount of time.

6) Example

CPU A	CPU B
Cores : 4	Cores : 6
Threads : 2 threads/core	Threads : 2 threads/core
Clock Hz : 3.5 GHz	Clock Hz : 3.0 GHz
memory BW : 60 GB/s	memory BW : 60 GB/s
L3 cache : 20 MB	L3 cache : 16 MB
Total threads = 2 X 4 = 8	Total Threads = 6 X 2 = 12

Q Is the application single threaded or multi-threaded?

→ If SINGLE Threaded, CPU A maybe better than CPU B
- Higher clock speed, which allows it to execute instructions more quickly
- Larger cache, means it can store more data that is frequently accessed, so it reduces the amount of time it takes to access memory.

→ If MULTI-THREADED, CPU B maybe better than CPU A
- Higher no. of cores, which means more threads total which means more ability to transfer more data to & from memory, more quickly.

cache-hit ratio also plays a role.

Factors that affect the handling of requests:

1) No. of cores and Threads: Multiple cores and threads can allow handling of multiple requests, simultaneously, in parallel.

2) Cache size & cache hit rate: Large cache size means CPU can store more frequently access data in its cache memory. Higher cache hit rate can reduce the time taken to fetch data.

3) Clock speed: Higher clock speed means the instructions can be executed more quickly. Higher clock speeds are directly proportional to higher power consumption & heat.

4) Memory Bandwidth: High bandwidth means CPU can transfer to and from memory, more quickly. This reduces the amount of wait time CPU spends to fetch.

5) IPC: Instructions Per Clock Cycle, usually calculated by a benchmarking test.

→ : Lifecycle of CPU core : ←

1) Fetch: CPU core reads instructions from memory and stores it in local cache for fast access.
2) Decode: CPU core then decodes the instructions into a series of micro-operations, which are instructions that CPU can execute.
3) Execute: CPU performs arithmetic & logical operations on data, stores result in registers or memory.

4) Write-back: The final stage is to write back results to memory or registers, so other parts of the system can access it.

* Container Runtime manages the allocation and execution of instructions on available CPU cores by using kernel-level abstractions and resource isolation mechanisms, provided by the host OS.

cgroups →

↳ container gets its own: file-system, process space, NW interface

↳ other procedures that the runtime manages: CPU scheduler, CPU isolation (CPU Pinning + CPU affinity), CPU throttling.

1) CPU Pinning: A process or a thread is assigned to a specific CPU core, preventing it from being scheduled on other CPU cores. Helps avoid the overhead of context switching and cache invalidation.

2) CPU Affinity: A process or thread is assigned to a specific SET of CPU cores, rather than just one, this is helpful in tasks that need parallel processing.

3) Linux kernel provides tools to control this:

taskset for CPU Pinning

numactl for CPU Affinity

NUMA, CoreDumps & Flamegraphs, GCP NW, intro

Some notes about NUMA (Non Uniform Memory Access)

1) It is used in multiprocessor systems.
2) Multiple processors (or cores) are connected to a shared memory system.
3) Access time depends on location of the memory relative to the core, that is accessing it.
4) In NUMA, memory is divided into multiple banks, each bank serving a subset of processors. (UMA is all cores all memory)
5) To take advantage of NUMA, software processes should be aware of the location of data in memory and should aim to keep the data close to the processors that most frequently accessed.
6) 'numactl' command is used to control and monitor NUMA Policy: [$ numactl -H]. Using this, you can further optimize your software by:
   - Assign threads or processes to specific NUMA nodes, using: `numactl --cpunodebind` (also helps with workload balancing)
   - Control memory allocation policy: `numactl --membind`
   - `numactl --interleave` : interleave memory allocations across all available NUMA nodes, helps allocate memory to local NUMA node
7) NUMASTAT [options] [pid]

If you don't give it a pid, it will output a summary of system-wide NUMA statistics.

`numastat -p 1234`

put: memory usage and access patterns for the specified process. (1234)
You get stats on usage for different types of data:
page tables, slab allocation, anonymous memory etc.

a) AnonPages: anonymous memory used by the process on each NUMA node, including memory allocated by the process using 'malloc' and 'calloc', as well as memory that the kernel allocated on behalf of the process.

b) Mapped: Amount of memory that is mapped by the process on each NUMA node, including 'mmap' and kernel based allocations.

c) Slab: Amount of memory used by kernel data structures on each NUMA node (eg inode, dentry caches etc)

conclusion: This is important because, for eg, if you notice that a process is using a lot of memory on a remote NUMA node, you can adjust the policy to allocate memory to local numa node, or if process is pulling too much memory from a non-local NUMA node, you can make the process run on the same NUMA node as the memory it is accessing.

Note: NUMA stats and topology within the container is/can be different from the host machine (unless its inherited)

Suggestions:
1) NUMA Optimized Docker Images
2) NUMA Aware Orchestration (K8s, Docker Swarm)
3) NUMA Aware Libraries & APIs (eg. libNUMA)
4) Optimize NUMA Policy

Core Dumps + flamegraphs

1) Core Dump is a file that contains a snapshot of a programs memory when it crashes or terminates abnormally.
2) It is a complete image of the programs memory including : values of all VARS, call stack, state of all threads, process at the time of crash
3) Upon a crash, a file with pid is generated. Enable this by command : ulimit or coredumpctl utility.
4) Tools to analyse this : [GDB, LLDB, Valgrind, Crash, Gcore]
5) While analyzing core dumps, you will encounter symbol files for the binary that gets generated. This file contains the mappings between the addresses in the binary and the names of the functions. (Make sure that the binary is not stripped)

Flamegraphs

1) It is a visualization tool that represents the call stack of a program as a series of rectangles, each one representing a function call.
2) Width of each rectangle represents the amount of time spent in that function. It is also proportional, and represents functions that consume more CPU time.
3) You can generate flame graphs using DTrace & SystemTap which actually create call stack trace, that you then use to generate the flame graph.

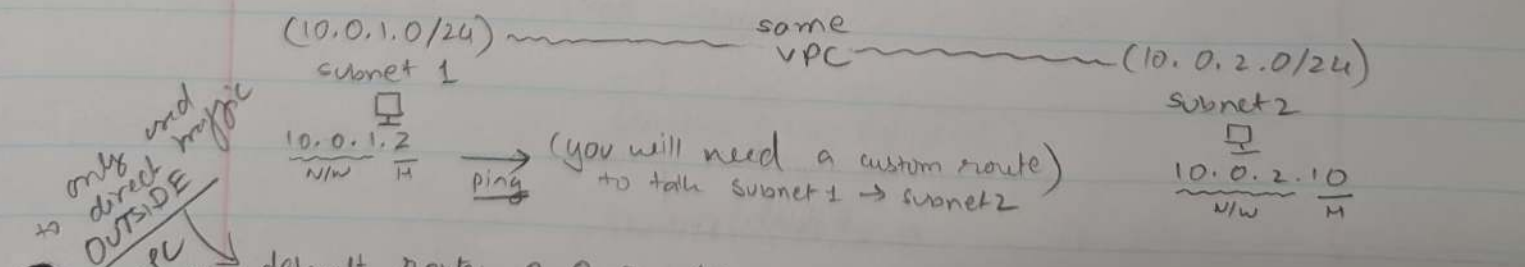
GCP Networking Landscape

1) GCP N/W landscape covers VPC (Virtual Private Cloud), Cloud Load Balancing & Cloud CDN.

2) VPC
- Each VPC can have one or more subnets (can span multiple zones within a region). Each subnet is associated with a region, and have a range of IP addresses. Remember, whatever '<IP>/n', no. of hosts always is $2^{(32-n)}$
- Routes dictate the path of traffic within the VPC. You get a default route when you create a subnet, but you can create custom routes. A route is a set of instructions that tells a network where to send traffic that is destined for a particular IP range.

order of priority:
1) subnet routes
2) static routes
3) default routes

- When you create a VM instance or a Cloud VPN tunnels, a system generated route gets automatically created. Custom routes can be created to direct traffic between specific destinations (on-premise N/W or other VPC's)
- If you do not define the next hop in a custom route, traffic is gonna go via the default internet gateway, which allows outbound traffic to the internets.



- VM instances within the same VPC N/W can communicate using private IP addresses, IF they are in the same region!

- You will need a VM instance in subnet-1, acting as a next-hop ~ gateway, so that your box can talk to it, which then passes along the ping, to subnet-2 via the static route that you set up.
- You can ALSO USE GCP Load Balancers & Routers as your gateways in GCP.

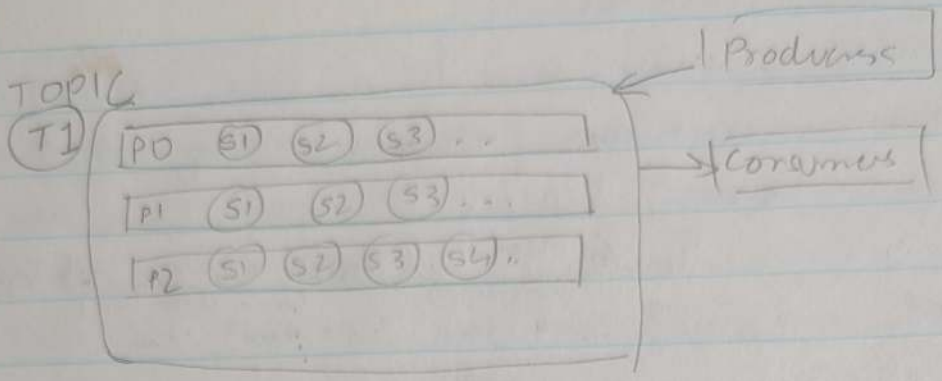
PULSAR
inmo
briefly

Pulsar Architecture

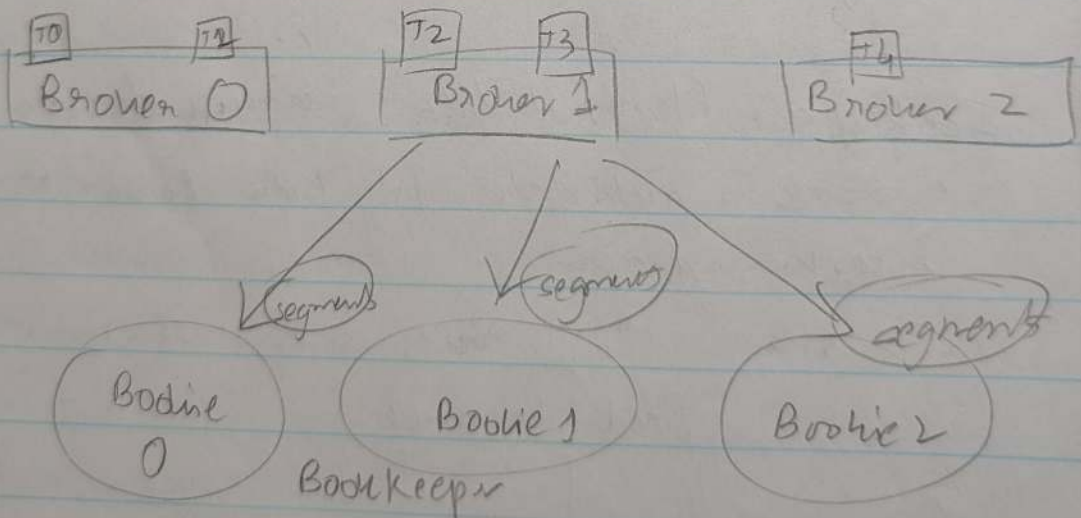
1) Instance has a no. of clusters
2) Clusters have – brokers
   – Zookeeper (Quorum, cluster-level config & co-ordination)
   – bookies (storage)
3) BROKER
   - handles all message routing & connection
   - stateless, but with caches
   - Automatic Load balancing
   - Topics & composed of multiple segments
4) Bookies
   - Stores segments (collection of Topics) and the messages with
   - A group of bookies form an ensemble to store a ledger
5) Zookeeper (for metadata & service discovery)
   - stores metadata for both Pulsar & Bookkeeper
   - services discovery.
6)

Prod Instance
|
Clusters (US, EU, ANE)
|
TENANT (diablo4 [illegible])
|
NAMESPACE ([illegible])
|
TOPIC

7) TOPIC



- Topic partitions are partitioning over space, (compute)
- segments are partitions over time
- Topic partions are composed of a sequence of segments
- segments contain messages, only one segment can be opened at a time.
- Brokers SERVE Topics
  Bookies STORE segments



- You chose a subset of Bookies to be an ensemble

8) BOOKIE (Bookkeeper)
- Separation of writes & Reads

write - segments get (WAL) Journal written to Journal File, on a FAST but small dedicated disk..

write Flush - Long Term storage (ledger disk), useful for Reads
- sort & flush to ledger disk (async write)
  Log Entrys: active writes, log entries)
- write to RocksDB ledger Indexe

Main Storage is a dedicated disk.

READS - Reads go to Write cache (has active writes)
↳ which goes to Read cache
↳ Lookups Read cache (pre-fill)

with 3.2 : The topic gets transferred to a new broker before the current broker get decomissioned

JOURNEY of PACKET (@ the NIC)

Journey of a Packet (when it arrives at the NIC)

1) Packet arrives at the NIC from the network
2) Packet is copied to a ring buffer in kernel memory (via DMA)
3) Hardware interrupt is generated to let the system know that the packet is in the memory. Driver clears the IRQ with the NIC. (since the IRQ is cleared, NIC can issue other IRQs for other arriving packets.)
4) Driver calls into NAPI (New API, helps with high speed networking) to start a poll loop. 'ksoftirqd' process runs on each CPU and polls packets off of the ring buffer (using NAPI poll function) ← (to harvest incoming packets)
5) Memory regions in the ring buffer that have incoming packet data, are unmapped.
6) Data DMA'd into memory are passed up the networking layer as an 'skb' for more processing
7) Packets are handed to the protocol layers from the queues.
8) Protocol layers add them to the receive buffers attached to the sockets

→ Why does softIRQ exists?

When the packet arrives, the NIC has to let the system know of it, but that IRQ interrupt is super high priority so it blocks other IRQs from being generated. The "soft" processes work OUTSIDE the device driver IRQ context.

1) net_rx_action loop starts by checking the NAPI poll list for NAPI structures
2) The budget and elapsed time are checked to ensure that the softIRQ will not monopolize the CPU time.
3) Registered poll function is called; this harvests packets from the ring buffer in the RAM.

4) Packets are handed over to napi_gro_receive. GRO, aka Generic Receive Offloading helps by reducing (combining) the no of packets passed up the network stack, if they are 'similar enough', like a large file transfer, that has contiguous data chunks.

5) So, packets are either held for GRO and the call ends here or they are passed on to net_receive_skb (which basically means the next layers of network stack now handle packets)

→ netif_receive_skb (using hash algorithm based on IP addrs & Ports, to distribute received traffic across multiple cores.)

- If RPS (Receive Packet Steering) is enabled:

1) netif_receive_skb passes data on to enqueue_to_backlog
2) Packets are placed on per-CPU input queue for processing.
3) The remote CPU's NAPI structure is added to that CPU's poll_list, and an IPI triggers the softIRQ kernel thread on the remote CPU to wake up.
4) ksoftirqd runs on remote CPU, but now, the poll function is process_backlog which harvests packets from the current CPU's input queue.

regular / (long RPS off)

→ 5) Packets are passed on to __net_receive_skb_core
6) __net_receive_core passes data to taps (like PCAP)
7) __net_receive_core delivers data to registered protocol layer handlers (usually ip_rcv function for IPv4 protocol stack)

Protocol Stack & Userland sockets.

1) Packets are received by the IPV4 stack using ip_rcv.
2) Netfilter and routing optimizations are performed.
3) Data is delivered to TCP / UDP.
4) Taking UDP as an example, data arrives to udp_rcv, and gets queued to the receive buffers of a userland socket by udp_queue_rcv_skb and sock_queue_rcv.
5) Prior to queuing, berkeley packet filters are processed.

→ Ports : an endpoint for a communication in a N/w.
(0 - 1023) : Reserved, don't use this.
(1024 - 49151) : have at it and (49152 - 65535) : for short lived stuff.

→ Socket = A combination of IP & PORT

SOCKET

1) chrome decides to use Port 80 for HTTP traffic.
2) When you type www.example.com, DNS translation takes place and now you have an IP for example.com: (203.0.113.0:80)
3) Data is sliced into packet, each packet has source socket (your IP : your Port) & destination socket (203.0.113.0:80)
(from the ephemeral port range)

HELM Templating Compendium

Helm / Go Templating Essentials (in helm context)

- The format is usually of type {{ .SomeVarName }}. These are action sequences or placeholder, that get added or replaced.
- data can also be referred to as 'context'.
- Actions
  - {{ .FieldName }} inserts the data value in the field.
  - {{ .MethodCall }} calls a method on a data object
  - {{ range .Collection }} ... {{ end }} iterates over a collection
  - {{ if .Condition }} ... {{ else }} ... {{ end }} conditional statements

- what even is 'helpers.tpl'?

This file contains reusable template snippets and functions that can be included in other templates using 'include' directive. example:

```
{{ /* helpers.tpl */ }}
{{- define "name-of-function" -}}
someKey: {{ .Values.someOtherKey.fromValuesFile }}
{{- end -}}
```

How to use this? in /templates/someTemplateFile.yaml

```
apiVersion: apps/v1
kind: Deployment
something:
  somethingElse:
    - someKey: someVal
    - BringMeHelp: {{- include "name-of-function" . | nindent 12 }}
```

need to calculate this

1) Define variables

{{ $appName := .Values.app.name }} // set var

name: {{ $appName }} // reference var

2) control structures: (can also use variables)

{{- if eq .Values.something "some value" }}

someKey: someValue

{{- else }} or {{- else if <something> }} (can also be >, <, <=, >=, eq, ne)

someKey: someOtherValue

{{- end }}

3) Functions (comes from helpers.tpl)

{{- include "function_name" . | nindent 12 }}

4) Pipelines & Text (built-in function)

{{ .Values.someValue | printf "%s : %s" .Values.someOtherV

{{- printf "%s (%d)" .Values.<string> .Values.<num> }}

5) Whitespace control

{{ .Values.someValue | trim All "/" }}

trim Prefix "/"

trim Suffix "/"

6) Comments: {{ /* comment */ }}

7) Nested templates

{{- define "some-function" -}}

stuff happens

{{- end -}}

{{- template "some-function" . }}

8) Handling newlines:

{{ something }} this will produce a newline

{{- something }} this will NOT produce a newline

9) setting default values and enforcing them:

{{ .Values.someValue | default "abc" }}

{{ required "provide mandatory value 'someValue'" .Values.someValue }}

{{ .Values.someValue | required "some message" }}

10) Text Manipulation

{{ .Values.someVal | squote }} // for single quotes

{{ .Values.someVal | quote }} // for double quotes

{{ .Values.someVal | b64enc }} // Base 64 encoding
b64dec

{{ .Values.someval | upper }} // uppercase / lowercase everything
or lower

{{ .values - someval | nindent <num> }} // add indentation

{{ randAlphaNum <num> }} // generate random alphanumeric of len(n

{{ "someString" | repeat <num> }} // repeats "someString", num times.

{{ .Values.someVal | replace "{placeholder}" "example" }} // replace stuff

11) Complex Data structures.

{{ .Values.hash.key }} // accessing a single hash key

→ iterating over hashmap

{{- range $key, $value := .Values.myhash }}

{{- printf "key %s : value %s \n" $key $value }}

{{- end }}

→ Grab a single value from the map.

{{- pluck .Values.env .Values.ip | first }}

↑ single key ↑ map

12) Check for stuff.

→ Check hash for key

hasKey .Values.mymap "mykey"

→ [>, <, eq]

gt .Values.SomeKey 2.0 // someKey > 2.0

lt .Values.SomeKey 5.0 // someKey < 5.0

eq .Values.SomeKey 1.0 // someKey == 1.0

13) Complex conditions

not (and (hasKey .Values.mymap "mykey") (lt .Values.someKey 2.0))

↳ and(bool, bool) → bool ↳ bool

↳ not (bool)

↳ bool!!

14) Reading Files

{{ print $.Template.BasePath "/someFile.yaml" }}

↑ path

Including Tpls,

{{ include mytemplate.tpl }}

↳ gotta be native to directory

15) Saving Context

all of the .Values map gets stored in the variable $root

{{- $root := . }}
{{- with .Values.something }}
...
someGlobal : {{ $root.Values.someGlobal }}
{{- end }}

16) Functions

{{- define "myfunction" }}
Hello World!
{{- end }}

Define a function

{{ myfunction }} // invoke a function, not quotations

→ if you want to pass values to the function,
{{ myfunction "some string" }} // pass a string
{{ myfunction 4 }} // pass a number
{{ myfunction . }} // pass the whole context
{{ myfunction $someVar }} // pass a variable

17) Built-ins

{{ .Release.Name }}

{{ .Release.Namespace }}

{{ .Chart.Name }}

{{ .Chart.Version }}

{{ .Files.Get config.ini }}

18) Loops

{{ range .Values.someArray }}

- someKey: {{ . }} // loop through all

OR

{{ range $someVar := .Values.someArray }}

- someKey: {{ $someVar }}

{{ end }}

OR

{{ range $key, $value := .Values.someArray }}

- {{ $key }} : {{ $value }}

{{ end }}

19) Render To YAML

{{ .values.something | toYaml }}

20) Lookup & fail

{{ $someVar := lookup "v1" "secret" .someValue }}

using fail in else: {{ else }}

{{ fail "message for fail" }}

21) Dates ← formatting

{{ now | date "2006-01-02T15:04:05" }}

22) UUID

id: {{ uuidv4 }}

23) Cryptographing

{{ .Values.SomeData | sha256sum }}

{{ .Values.SomeData | encrypt_AES "secret key" }}

{{ .Values.SomeData | decrypt_AES "secret key" }}

* journald - centralized logging system for linux kernel
* Sysctl - it is used to list, read and set kernel tunables.
* journalctl - query & display logs from journal, which is a part of systemd.
* systeroid - its a more powerful alternative to sysctl.
  procfs: virtual file system, typically mapped to '/proc' at boot time.
  ↳ This means checking a kernel parameter involves opening /proc, reading its contents, parsing them and closing the file

Usage: docker run --rm -it "orhun/systeroid:latest" --tui

Docker containers share the host systems kernel and its settings thus access to /proc & /sys are restricted. so run is read only.

* tar

J => xz
z => gzip
j => bzip2

compress → tar -cJf archive.tar.xz /path/to/dir_or_files
archive → tar -xJf archive.tar.xz

* sed

replace all occurrences → sed 's/pattern/replacement/g' file.txt
insert before match → sed '/pattern/i\inserted-text' file.txt
after match → sed '/pattern/a\appended-text' file.txt

* scp & rsync

scp username@remote_host:/path_to_remote_file /path_to_local
if youre ssh'd in, use destination as: user@mac_host:/path

rsync -azv --progress user@remote:/path_to_file /path_to/local

-v verbose
-a preserve permission
-z compress

TLS, SSL & ACME

TLS, SSL and basic flow of secure comms over HTTPS

1) SSL, TLS, are cryptographic protocols that allow for secure communications
2) We needed this because communications over HTTP are plaintext.
3) Features include Encryption, Authentication, Data Integrity, Handshakes, certificates, and a widespread use in Web Traffic (HTTPS), email (SMTP), POP/IMAP, and some other network protocols.
4) How to go about creating a SSL/TLS certificate? (using OpenSSL)
   a) Create a private key: `openssl genpkey -algorithm RSA -out my.key`
      for PEM & encryption, add: `-outform PEM -aes256`
   b) Generate CSR: `openssl req -new -key my.key -out my.csr`
      ↳ add the FQDN or pass a conf file
   c) Submit CSR to the CA for validation.
5) CA & validation: This step determines the authenticity of a CSR
   a) Domain Control Validation (DCV): CA sends an email to admin@mydomain.com
   OR
   b) Org verify (check with govt for org identity), Extended verify (background check)
   c) ACME also does domain validation process automatically.
6) Types of Certs
   a) Domain validation: validating www.mycoolsite.com ONLY
   b) Org validation: validating www.company.com ONLY
   c) Extended validation: validating www.bank.com, for finance, e-com platform
   d) Wildcard cert: validate websites with multiple domains: *.mycoolsite.com
   e) Multi-Domain SAN: validate www.site1.com, blog.site2.com, lol.site3.com
   f) code-sign cert: sign a downloadable binary, validate identity of code-signer
   g) TLS server cert: for webservers (can be DV, OV, EV)
   h) Client cert: authentication of VPN client with VPN server
   i) self-signed cert: for local testing, signed by yourself (NOT CA)

7) Certificate Chains : Part of the TLS protocol, and is designed to establish trust in the authenticity and identity of the server. The chain links a the server's cert to a trusted root CA.

a) cert issued by the CA (aka your cert) has your public key & FQDN.

b) chain is made up of root CA cert & one or more intermediate certs.

c) Root CA cert : self signed, exists already on OS & browsers, well known.

d) Intermediate cert : Belonging to intermediate CA's, but they get this from the root CA, using their own private key.

e) server cert (aka your cert) : This is your cert, which you made.

f) i) CA, basically uses its private key to create a digital signature.

ii) CA also compiles data (public key of you, identity, expiration date etc).

iii) CA then applies SHA-256 to all of this cert data.

iv) CA then uses private key to encrypt this hash value

v) cert data + digital signature = Your cert

8) Signature validation :

a) server sends its cert to the client

b) Client retrieves the public key of the CA

c) Client creates a hash of cert data from the server cert.

d) Client does a comparison : if (myhash == decrypted (cert_hash)

e) If two hash values match, signature on the server cert is valid.

f) This process is repeated for all certs till we reach the root cert.

9) Different certificate formats

a) PEM (Privacy Enhanced Mail) : Base64 encoded X.509 cert, widely used

b) DER (Distinguished Encoding Rules) : Binary format, used in Java apps.

c) PKCS#7 or P7X : Password protected, used in Windows.

d) PKCS#12 or PFX : Password protected, used in Windows.

(Combine cert + private key, securely ✓)

9)

10) e) PEM + key pair: Secure, convenient

f) CER/CRT / DER / PEM : used interchangably

g) JKS (Java KeyStore) : Securely store → (pvt. keys, certs, trusted certs) for Java Apps

h) P7R/P7S: for digitally signed messages & documents

i) CRL (Certificate Revocation List): a list of revoked certs by the CA.

j) CSR (Certificate Signing Request): info needed by CA to issue a cert.

10) ACME (Automated Certificate Management Environment): This is a protocol designed to automate the issuance, renewal and managment of SSL/TLS certificates. Usually used to get free certs from LetsEncrypt.

a) User/server provides info & domain to ACME Client to issue CSR.

b) ACME client generates RSA key pair (public + pvt key) & sends CSR to ACME server

c) ACME client issues a challenge (DNS or http) (token in DNS TXT; token on server path) to validate your ownership/control over a domain (that you requested the cert for)

d) The result of that challenge is sent to ACME server.

e) When all validation completes, ACME server issues the certificate.

f) ACME client receives the certificate & installs on your server.

BONUS

* PGP (Pretty Good Privacy) & GPG (GNU Privacy Guard) are encryption & decryption programs that provide privacy & authentication.
* Used in: Encrypting email, digital signatures, file encryption, Authentication,
* 1) gen key: gpg --gen-key
2) export public key: gpg --output public.asc --armor --export user@mail.com
3) import public key: gpg --import public.asc

4) Encrypt: gpg --encrypt --recipient you@mail.com --output email.gpg --armor email.txt

5) Decrypt: gpg --output decrypted.email.txt --decrypt email.gpg

6) Sign: gpg --output signed.gpg --armor --sign email.txt

7) Verify: gpg --verify signed.gpg

8) File encrypt: gpg --output myfile.gpg --encrypt myfile.txt

9) Verify Package: gpg --verify mypkg.tar.gz.sig mypkg.tar.gz

10) SSH key auth: gpg --export-ssh-key key-id > ~/.ssh/id_gpg

11) Commit signing: git commit -S -m "my commit message"

12)

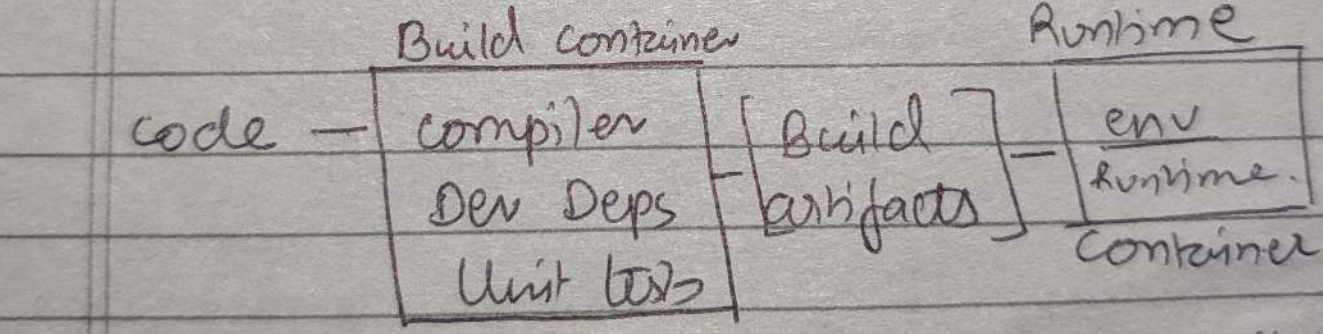
- kind & configs for API
- consumers, ratelimit, Gateway

Kubernetes Best Practices

(1) Dont trust arbitrary base images.
- use quay.io to scan containers.

(2) Overhead endured with base images. Use scratch or alpines.

(3) use the builder pattern.



(4) Build dockerfile
FROM golang:alpine

Runtime dockerfile

Build container

```
FROM golang:alpine AS build-env
WORKDIR /app
ADD . /app
RUN cd /app && go build -o goapp
```

Runtime container

```
FROM alpine
WORKDIR /app
COPY --from=build-env /app/goapp /app
EXPOSE 8080
ENTRYPOINT ./goapp
```

④ Inside your container.
- dont need to be root

```
FROM node:alpine
RUN apk update && apk add imagemagick

RUN groupadd -r nodejs
RUN useradd -m -r -g nodejs nodejs
USER nodejs

ADD package.json package.json
RUN npm install
ADD index.js index.js
CMD npm start
```

⑤ Force security via helm / yaml declaration

```
securityContext:
    runAsNonRoot: true
    readOnlyRootFilesystem: true
```

⑥ One process per container

⑦ Dont restart on failure, crash cleanly instead.
kubernetes will detect the crash, log the info and automatically restart the application.

⑧ Log everything to stdout & stderr.

⑨ Deployment - record option for easier rollback.

$ kubectl apply -f dep.yaml --record

⑩ Use plenty of descriptive labels.
You can use labels to do canary deployments.

(11) sidecar containers
proxy to the DB

or proxy to incoming requests.
use it for things that always happen.

(12) init-containers
the text below needs to be defined in the deployment.yaml file.

annotations:
pod.beta.kubernetes.io/init-containers: '
[{
"name": "init-myapp",
"image": "busybox",
"command": ["sh", "-c", "until nslookup myapp;
do echo waiting for myapp;
sleep 2; done;"]
},
{
"name", "init-mydb",
"image": "busybox",
"command": ["sh", "-c", "until nslookup mydb;
do echo waiting for mydb;
sleep 2; done;"]
}]

This waits for your downstream dependencies to be up and running.
Use this for bootstrapping.

(13) Dont use 'latest' or 'no tag' to avoid [redacted].

(14) Readiness & liveness probes.
↓ is app ready to start serving traffic?
↓ is app still running?

(15) Dont always use type: Loadbalancer
Ingress is great: Lets you loadbalance multiple services through one endpoint

(16) type: Nodeport can be good enough.
exposes the service on a VM on a port.
If VM goes down, access goes down.

(17) Use Static IP's.

for GCP

gcloud compute addresses create
myingress --global

gcloud compute addresses create mysanic
--region = us-east1

for LB

spec:
type: LoadBalancer
loadBalancerIP: xxx.yyy.zzz.qqq

for Ingress

annotations:
kubernetes.io/ingress.global-static-ip-name:
'myingress'

(8) Map external services to external one.

Lets say you have a hosted database.
my.database.example.com

for that, the external nonane service is:

```
kind: Service
apiVersion: v1
metadata:
  name: mydatabase
  namespace: prod
Spec:
  type: ExternalName
  externalName: my.database.example.com
```

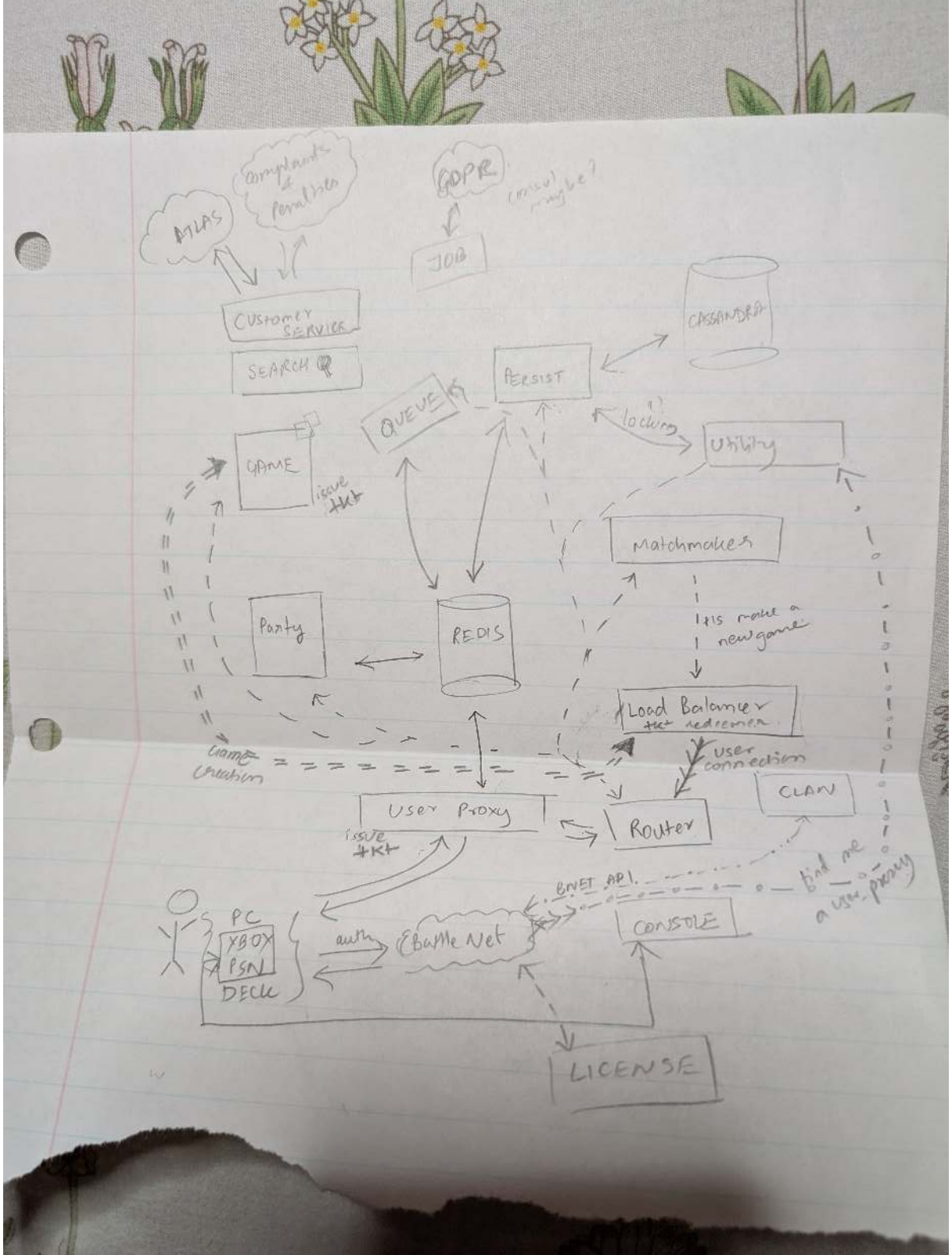
So this basically does a CNAME redirect to the external name.

Your internal services can just use the mydatabase name.

(19) If you dont have Canonical Names, create and Endpoint and list a bunch of IP's.

```
kind: Endpoints
apiVersion: v1
metadata:
  name: mydatabase
subsets:
  - addresses:
      - ip: x.x.x.x
    ports:
      - port: 12345
```

(20) Role based Access control


ATLAS
complaints + penalties
GDPR
(once) maybe?
JOB
Customer SERVICE
SEARCH
CASSANDRA
PERSIST
QUEUE
locking
Utility
GAME
save tkt
Matchmaker
Party
REDIS
this make a new game
Load Balancer
user connection
CLAN
Game Creation
User Proxy
Router
issue tkt
BNET API
bind me a user proxy
PC
XBOX
PSN
DECK
auth
Battle Net
CONSOLE
LICENSE

CA(L)MS

- Culture
- Automation
- Lean
- Measurement
- Sharing & Collaboration

- You cannot ensure 100% availibility on any system.
- Select appropriate SLOs with the product team.
- SLO violations bring teams back to the drawing board, blamelessly.
- Try to reduce MTTR.
- SRE Expertise around
  - availibility
  - latency
  - performance
  - efficiency
  - Change Mgmt
  - Monitoring
  - emergency response
  - Capacity Planning.

Stack:
frontend
backend
libraries
storage
kernels
physical machine

Get to a state
- Stakeholder approved SLO's for the product.
  - Identify all the right stakeholders
- The people responsible for ensuring that the service meets the SLO have agreed that it is possible to meet the SLO under normal circumstances.
- The org has committed to use error budgets for decision making and prioritizing this commitment is formalized as an error budget policy.
- Process in place for refining SLO's

- You can go from 99% to 99.9% to 99.99% to 99.999% reliability, each extra nine comes at at increased cost, but the marginal utility to your customers approaches 0!

- Someone needs to make a tradeoff between feature velocity and reliability.

- SLI - service level indicator is a ratio between two numbers examples:
  - → $\frac{\text{No. of successful HTTP reqs}}{\text{total HTTP reqs.}}$
  - → $\frac{\text{No. of gRPC calls finished} < 100\text{ms}}{\text{total gRPC reqs.}}$

- SLI always going to be a %.

Error Budget : 100% − SLO%

: SLI Implementation :-

⊖ SLI - Availability

$$\frac{\sum (\text{reqs for host } \forall \text{ status} \neq 500)}{\sum (\text{reqs for host})}$$

for certain t window.

⊖ Latency

·) decide a threshold : 100 ms , 500 ms

·) Find the rate at those thresholds.

·) Find the 90% percentile
99% percentile
& the rate you get.

·) Build a histogram over t window.

SLO & Error Budget DOC

DOC/SLO should include

- Authors & SLOs
- Reviewers
- Date & approval
- Date & next review
- Brief description
- Details of SLO: objectives & SLI implmn.
- Details on how error budget is calculated & consumed
- Rationale behind numbers:
  - Experimental or observational.

*

DOC/Error budget

- Authors, reviewers, approvers.
- date & approval and next review date.
- description & service to give context.
- Action to take in response to budget exhaustion.
- Escalation path

*

* Dashboards in NR

- Reports and qs Q1. 1) SLO met
  2) Error budget burndown.
- Graph Budget Loss / Day
  Vs. Tickets per day

4 .- Construct SLO Decision Matrix:

SLO	Toil	Satisfaction	Action.

Possible SLI's

- DOM content Loaded =

  initial navigation (t) to end of DOMContentLoaded

- Start Render =

  start of initial nav to first non-white paint

  → important in understanding how long users have to wait before anything is displayed on the screen.

- Visually complete: all the content in viewport has finished, and nothing is changed.

- Page Load: start of initial nav to start of window load event. (NOT A GOOD measure)

Percentiles

A percentile is a value below which a certain % of observations lie.

$$\text{Percentile rank of } x = \left(\frac{\#\text{ of values below } x}{n} \right) \times 100$$

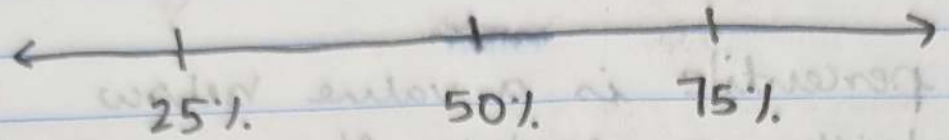
We usually work out the value of x for a given percentile.

- basically, what x would we have 90% or 99% values below it.

$$x = \frac{\%\text{ tile}}{100} \times (n+1)$$

where n is the total number of values (data set)

Quartiles



They're basically 25, 50 & 75 percentiles.

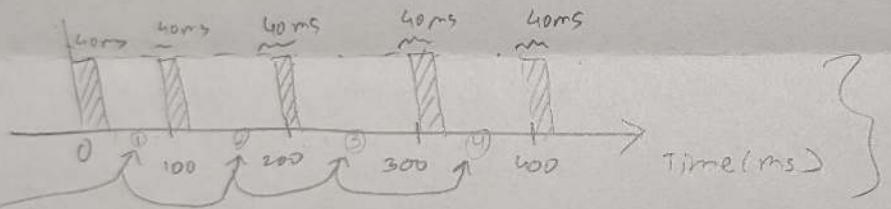
CPU Throttling Maths.

- CPU allocation uses { Quota / Period }
- when an application has used its allocated CPU Quota for given period, it gets throttled until the next period.
- settings: cpu.cfs_quota_us
  cpu.cfs_period_us
- within cpu.stat, if nr_throttled goes up, the response times can be slow.
- lets say an app takes 200 milliseconds to finish a request.

(1) No constraints: Req comes in, gets served in 200 ms.

(2) Constraint of 0.4 CPU:

App gets 40 ms run time per 100 ms per



Total = 440 ms

For 1000 ms time:

nr_periods = 5 ∵ from 440 ms to 1000 ms, not runnable

nr_throttled = 4

throttled_time = 240 ms ∵ for eg 100ms, app runs = 40 ms
throttled = 60 ms
60 x 4 = 240 ms

throttled_% = $\frac{4}{5} \times \frac{20}{100}$ = 80%

- The kernel tracks the amount of time still available.
- Kernel allocates slices of cfs_bandwidth → quota to each core. Default slice is 5 ms, defined in kernel.sched_cfs_bandwidth_slice_us.

Lets consider multi-threaded daemon with two worker threads, each pinned to their own core.

CPU = 0.2 ⇒ 20ms of Quota ← that's the Global Available Quota.

Time	Action
30 ms	• Request comes in for worker 1. • W1 needs only 1ms to process the request. 4ms left on the per CPU bucket. (5ms slice) • we have 4ms remaining on per CPU run queue, but there are no more runnable threads on W1 CPU • A timer is set to return the slack quota back to the global bucket. • Timer = 7ms after W1 stops working
38 ms	• SLACK timer on W1 CPU triggers and returns all but 1ms of quota back to the global pool of quota. • This leaves 1ms of quota on CPU 1
41 ms	• W2 gets a long request. • All remaining time is transferred to CPU W2's per CPU bucket, W2 uses all of it.
49 ms	W2 CPU is now throttled without completing the request. Throttled despite of the fact that CPU 1 still has 1ms of Quota.

1ms adds up on all cores. If ya have a 88 core machine, we could lose upto 87 ms per period, that's 87ms ⇒ 870 millicores or 0.87 CPU unusable.

no. of slices = no. of cores used.

if CPU = 100 ms on 1 CPU
slice period = 5 ms
$\frac{100}{5}$ = 20 slices = 20 cores used.