Objectives for NFR

**Monitoring**

* Monitoring should never require a human to interpret any part of the alerting domain
* Three valid kinds of monitoring output
  + Alerts: human needs to take action immediately
  + Tickets: human needs to take action eventually
  + Logging: no action needed
  + Note that, for example, graphs are a type of log

#### **Emergency Response**

* Reliability is a function of MTTF (mean-time-to-failure) and MTTR (mean-time-to-recovery)
* For evaluating responses, we care about MTTR
* Humans add latency
* Systems that don’t require humans to respond will have higher availability due to lower MTTR
* Having a “playbook” produces 3x lower MTTR
  + Having hero generalists who can respond to everything works, but having playbooks works better

#### 

#### **Change management**

* 70% of outages due to changes in a live system. Mitigation:
  + Implement progressive rollouts
  + Monitoring
  + Rollback
* Remove humans from the loop, avoid standard human problems on repetitive tasks

#### 

#### **Provisioning**

* Adding capacity riskier than load shifting, since it often involves spinning up new instances/locations, making significant changes to existing systems (config files, load balancers, etc.)
* Expensive enough that it should be done only when necessary; must be done quickly
  + If you don’t know what you actually need and overprovision that costs money

### **Embracing risk**

* DLT solutions, what does it mean 99% reliability,can’t we tell the difference between 99.99% and 99.999% reliability

#### **Managing risk**

* Reliability isn’t linear in cost. It can easily cost 100x more to get one additional increment of reliability
  + Cost associated with redundant equipment
  + Cost of building out features for reliability as opposed to “normal” features
  + Goal: make systems reliable enough, but not too reliable!

#### **Measuring service risk**

* Standard practice: identify metric to represent property of system to optimize
* Possible metric = uptime / (uptime + downtime)
  + Problematic for a globally distributed service. What does uptime really mean?
* Aggregate availability = successful requests / total requests
  + Obv, not all requests are equal, but aggregate availability is an ok first order approximation
* Usually set monthly targets

#### **Risk tolerance of services**

* Usually not objectively obvious
* Work with TPM (Duk) to translate business objectives into explicit objectives

#### **Identifying risk tolerance of consumer services**

TODO

#### **Motivation for error budgets**

TODO - negotiate with TPM

### **Service level objectives**

#### 

#### 

#### **What do you and your users care about?**

* Too many indicators: hard to pay attention
* Too few indicators: might ignore important behavior
* Different classes of services should have different indicators
  + User-facing: availability, latency, throughput
  + Storage: latency, availability, durability
* All systems care about correctness

#### **Collecting indicators**

* Can often do naturally from server, but client-side metrics sometimes needed.

#### **Aggregation**

* Use distributions and not averages
* User studies show that people usually prefer slower average with better tail latency
* Standardize on common defs, e.g., average over 1 minute, average over tasks in cluster, etc.
  + Can have exceptions, but having reasonable defaults makes things easier

#### **Choosing targets**

* Don’t pick target based on current performance
  + Current performance may require heroic effort
* Keep it simple
* Avoid absolutes
  + Unreasonable to talk about “infinite” scale or “always” available
* Minimize number of SLOs
* Perfection can wait
  + Can always redefine SLOs over time
* SLOs set expectations
  + Keep a safety margin (internal SLOs can be defined more loosely than external SLOs)
* Don’t overachieve
  + Example is making sure that the system isn’t too fast under light loads

### **Eliminating toil**

If a human operator needs to touch your system during normal operations, you have a bug. The definition of normal changes as your systems grow.

* Def: Toil
  + Not just “work I don’t want to do”
  + Manual
  + Repetitive
  + Automatable
  + Tactical
  + No enduring value
  + O(n) with service growth
* In surveys, find 33% toil on average
  + Numbers can be as low as 0% and as high as 80%
  + Toil > 50% is a sign that the manager should spread toil load more evenly
* Is toil always bad?
  + Predictable and repetitive tasks can be calming
  + Can produce a sense of accomplishment, can be low-risk / low-stress activities

### **Monitoring distributed system**

* Why monitor?
  + Analyze long-term trends
  + Compare over time or do experiments
  + Alerting
  + Building dashboards
  + Debugging

#### **Setting reasonable expectations**

* Monitoring is non-trivial
* Team typically has 1-2 people building and maintaining monitoring
* General trend towards simpler/faster monitoring systems, with better tools for post hoc analysis
* Avoid “magic” systems
* Limited success with complex dependency hierarchies (e.g., “if DB slow, alert for DB, otherwise alert for website”).
  + Used mostly (only?) for very stable parts of system
* Rules that generate alerts for humans should be simple to understand and represent a clear failure

**Avoiding magic**

* Lots of white-box monitoring
* Some black-box monitoring for critical stuff
* Four golden signals
  + Latency
  + Traffic
  + Errors
  + Saturation

#### **The long run**

* There’s often a tension between long-run and short-run availability
* Can sometimes fix unreliable systems through heroic effort, but that’s a burnout risk and also a failure risk
* Taking a controlled hit in short-term reliability is usually the better trade

### **Evolution of automation**

* “Automation is a force multiplier, not a panacea”
* Value of automation
  + Consistency
  + Extensibility
  + MTTR
  + Faster non-repair actions
  + Time savings

### **Release engineering**

* Define best practices
  + Compiler flags, formats for build ID tags, etc.
* Releases automated
* Models vary between teams
  + Could be “push on green” and deploy every build
  + Could be hourly builds and deploys
  + etc.
* Hermetic builds
  + Building same rev number should always give identical results
  + Self-contained -- this includes versioning everything down the compiler used
  + Can cherry-pick fixes against an old rev to fix production software
* Virtually all changes require code review
* Branching
  + All code in main branch
  + Releases are branched off
  + Fixes can go from master to branch
  + Branches never merged back
* Testing
  + CI
  + Release process creates an audit trail that runs tests and shows that tests passed
* Config management
  + Deceptively simple, can cause instability
* Many possible schemes (all involve storing config in source control and having strict config review)
* Use mainline for config -- config maintained at head and applied immediately
  + Originally used for k8s
  + Binary releases and config changes decoupled!
* Include config files and binaries in same package
  + Simple
  + Tightly couples binary and config -- ok for projects with few config files or where few configs change
* Package config into “configuration packages”
  + Same hermetic principle as for code
* Release engineering shouldn’t be an afterthought!
  + Budget resources at beginning of dev cycle

### **Simplicity**

* Stability vs. agility
  + Can make things stable by freezing -- need to balance the two
  + Reliable systems can increase agility
  + Reliable rollouts make it easier to link changes to bugs
* Virtue of boring!
* [Essential vs. accidental complexity](https://en.wikipedia.org/wiki/No_Silver_Bullet)
  + We should push back when accidental complexity is introduced
* Code is a liability
  + Remove dead code or other bloat
* Minimal APIs
  + Smaller APIs easier to test, more reliable
* Modularity
  + API versioning
  + Same as code, where you’d avoid misc/util classes
* Releases
  + Small releases easier to measure
  + Can’t tell what happened if we released 100 changes together

### **Altering from time-series data**

* [Prometheus](https://prometheus.io/)
* Common data format for logging
* Data used for both dashboards and alerts
* Formalized a legacy data format, “varz”, which allowed metrics to be viewed via HTTP
  + To view metrics manually, go to [http://foo:80/varz](http://foo/varz)
* Adding a metric only requires a single declaration in code
  + low user-cost to add new metric
* Prom fetches /varz from each target periodically
  + Also includes synthetic data like health check, if name was resolved, etc.,
* Time series arena
  + Data stored in-memory, with checkpointing to disk
  + Fixed sized allocation
  + GC expires oldest entries when full
  + conceptually a 2-d array with time on one axis and items on the other axis
  + 24 bytes for a data point -> 1M unique time series for 12 hours at 1-minute intervals = 17 GB
* Prom rules
  + Algebraic expressions
  + Compute time-series from other time-series
  + Rules evaluated in parallel on a threadpool
* Counters vs. gauges
  + Def: counters are non-decreasing
  + Def: can take any value
  + Counters preferred to gauges because gauges can lose information depending on sampling interval
* Altering
  + Prom rules can trigger alerts
  + Have minimum duration to prevent “flapping”
  + Usually set to two duration cycles so that missed collections don’t trigger an alert
* Scaling
  + Prom can take time-series data from other Prom (uses binary streaming protocol instead of the text-based varz protocol)
  + Can have multiple tiers of filters
* Prober
  + Black-box monitoring that monitors what the user sees
  + Can be queried with varz or directly send alerts to Altertmanager
* Configuration
  + Separation between definition of rules and targets being monitored

### **Being on-call**

* Typical response time
  + 5 min for time-critical tasks
  + 30 min for less time-sensitive stuff
* Response times linked to SLOs
  + Ex: 99.99% for a quarter is 13 minutes of downtime; clearly can’t have response time above 13 minutes
  + Services with looser SLOs can have response times in the 10s of minutes (or more?)
* Primary vs secondary on-call
  + Work distribution varies by team
  + In some, secondary can be backup for primary
  + In others, secondary handles non-urgent / non-paging events, primary handles pages
* Balanced on-call
  + Def: quantity: percent of time on-call
  + Def: quality: number of incidents that occur while on call

### **Managing incidents**

### **Load balancing**

* Flow control
* Need to avoid unhealthy tasks
* Naive flow control for unhealthy tasks
  + Track number of requests to a backend
  + Treat backend as unhealthy when threshold is reached
  + Cons: generally terrible
* Health-based flow control
  + Backend task can be in one of three states: {healthy, refusing connections, lame duck}
  + Lame duck state can still take connections, but sends backpressure request to all clients
  + Lame duck state simplifies clean shutdown
* Def: subsetting: limiting pool of backend tasks that a client task can interact with
  + Clients in RPC system maintain pool of connections to backends
  + Using pool reduces latency compared to doing setup/teardown when needed
  + Inactive connections are relatively cheap, but not free, even in “inactive” mode (reduced health checks, UDP instead of TCP, etc.)
* Choosing the correct subset
  + Typ: 20-100, choose base on workload
* Subset selection: random
  + Bad utilization
* Subset selection: round robin
  + Order is permuted; each round has its own permutation
* Load balancing
  + Subset selection is for connection balancing, but we still need to balance load
* Load balancing: round robin
  + In practice, observe 2x difference between most loaded and least load
  + In practice, most expensive request can be 1000x more expensive than cheapest request
  + In addition, there’s random unpredictable variation in requests
* Load balancing: least-loaded round robin
  + Exactly what it sounds like: round-robin among least loaded backends
  + Load appears to be measured in terms of connection count; may not always be the best metric
  + This is per client, not globally, so it’s possible to send requests to a backend with many requests from other clients
  + In practice, for larg services, find that most-loaded task uses twice as much CPU as least-loaded; similar to normal round robin
* Load balancing: weighted round robin
  + Same as above, but weight with other factors
  + In practice, much better load distribution than least-loaded round robin

### **Handling overload**

* Even with “good” load balancing, systems will become overloaded
* Typical strategy is to serve degraded responses, but under very high load that may not be possible
* Modeling capacity as QPS or as a function of requests (e.g., how many keys the requests read) is failure prone
  + These generally change slowly, but can change rapidly (e.g., because of a single checkin)
* Better solution: measure directly available resources
* CPU utilization is *usually* a good signal for provisioning
  + With GC, memory pressure turns into CPU utilization
  + With other systems, can provision other resources such that CPU is likely to be limiting factor
  + In cases where over-provisioning CPU is too expensive, take other resources into account

*How much does it cost to generally over-provision CPU like that?*

* Client-side throttling
  + Backends start rejecting requests when customer hits quota
  + Requests still use resources, even when rejected -- without throttling, backends can spend most of their resources on rejecting requests
* Criticality
  + Seems to be priority but with a different name?
  + First-class notion in RPC system
  + Client-side throttling keeps separate stats for each level of criticality
  + By default, criticality is propagated through subsequent RPCs
* Handling overloaded errors
  + Shed load to other DCs if DC is overloaded
  + Shed load to other backends if DC is ok but some backends are overloaded
* Clients retry when they get an overloaded response
  + Per-request retry budget (3)
  + Per-client retry budget (10%)
  + Failed retries from client cause “overloaded; don’t retry” response to be returned upstream

### **Addressing cascading failures**

* Typical failure scenarios?
* Server overload
* Ex: have two servers
  + One gets overloaded, failing
  + Other one now gets all traffic and also fails
* Resource exhaustion
  + CPU/memory/threads/file descriptors/etc.
* Ex: dependencies among resources
  + 1) Java frontend has poorly tuned GC params
  + 2) Frontend runs out of CPU due to GC
  + 3) CPU exhaustion slows down requests
  + 4) Increased queue depth uses more RAM
  + 5) Fixed memory allocation for entire frontend means that less memory is available for caching
  + 6) Lower hit rate
  + 7) More requests into backend
  + 8) Backend runs out of CPU or threads
  + 9) Health checks fail, starting cascading failure
  + Difficult to determine cause during outage
* Note: policies that avoid servers that serve errors can make things worse
  + fewer backends available, which get too many requests, which then become unavailable
* Preventing server overload
  + Load test! Must have realistic environment
  + Serve degraded results
  + Fail cheaply and early when overloaded
  + Have higher-level systems reject requests (at reverse proxy, load balancer, and on task level)
  + Perform capacity planning
* Queue management
  + Queues do nothing in steady state
  + Queued reqs consume memory and increase latency
  + If traffic is steady-ish, better to keep small queue size (say, 50% or less of thread pool size)
  + Ex: Gmail uses queueless servers with failover when threads are full
  + For bursty workloads, queue size should be function of #threads, time per req, size/freq of bursts
  + See also, adaptive LIFO and CoDel
* Graceful degradation
  + Note that it’s important to test graceful degradation path, maybe by running a small set of servers near overload regularly, since this path is rarely exercised under normal circumstances
  + Best to keep simple and easy to understand
* Retries
  + Always use randomized exponential backoff
  + See previous chapter on only retrying at a single level
  + Consider having a server-wide retry budget
* Deadlines
  + Don’t do work where deadline has been missed (common theme for cascading failure)
  + At each stage, check that deadline hasn’t been hit
  + Deadlines should be propagated (e.g., even through RPCs)
* Bimodal latency
  + Ex: problem with long deadline
  + Say frontend has 10 servers, 100 threads each (1k threads of total cap)
  + Normal operation: 1k QPS, reqs take 100ms => 100 worker threads occupied (1k QPS \* .1s)
  + Say 5% of operations don’t complete and there’s a 100s deadline
  + That consumes 5k threads (50 QPS \* 100s)
  + Frontend oversubscribed by 5x. Success rate = 1k / (5k + 95) = 19.6% => 80.4% error rate