# Contents

# 1 Introduction

# 2 The Production Environment at Google, from the Viewpoint of an SRE

# 3 Embracing Risk

- 100% reliability is unhelpful given that upstream are things which are less reliable than that (3G connections; ISP last mile; etc)

## 3.1 Managing Risk

- Striving for too much uptime has costs
  - > product development is slower
  - > redundant resources are costly
  - > incremental improvements in reliability have super-linear costs past a point

## 3.2 Measuring Service Risk

- duration-based availability measures (availability = time up / (time up + time down)) doesn't work well for systems with any fault isolation
- what's better? aggregate availability (availability = succeeded requests / received requests)
  - > non-ideal: this metric considers important, difficult requests (perform a transaction, payment included) equal to all others (show the FAQ)
- thsi approach works well enough for any unit of work other than (web) requests

## 3.3 Risk tolerance of services

### 3.3.1 Identifying the risk tolerance of consumer services

- what level of availability is required?
- do different types of failures have different effects on the service?
- how can we use the service cost to help locate a service on the risk continuum?
- what other service metrics are important to take into account

#### 3.3.1.1 target level of availability

- what level of service do users expect?
- does this service tie directly to revenue?
- is this a paid service or a free one?
- if there are competitors, what level of service do they provide?
- is this service targeted at consumers or enterprises?

- e.g., Google Apps for Work
  - > an outage blocks work for paying customers
  - > possibly: external availability target of 99.9% (with contract stipulating penalties paid on violation), and a higher internal target

#### 3.3.1.2 Types of failures

#### 3.3.1.3 Cost

- What additional revenue would we get from adding one more 9 of availability?
- Does that offset the cost of obtaining and maintaining that additional 9?
- We've measured the typical background error rate for ISPs as falling between 0.01% and 1%.

#### 3.3.1.4 Other service metrics

- service latency of adwords is pegged to the latency of search
- adsense's target latency is in relation to the third-party page

### 3.3.2 Identifying the risk tolerance of infrastructure services

#### 3.3.2.1 target level of availability

- bigtable may have some units of work in the path of an online request (latency-prioritizing), others may be bulk workloads (throughput-prioritizing)

#### 3.3.2.2 types of failures

- low-latency uses benefit when queues are empty; high-throughput uses benefit when queues are never depleted

#### 3.3.2.3 cost

## 3.4 Motivation for Error Budgets

- SRE goal: define an objective metric agreed upon by both product and SRE that can be used to guide negotiations on velocity vs reliability in a reproducible way.

### 3.4.1 Forming your Error Budget

- product management defines a quarterly error budget based on service level objective
- new releases can be pushed iff the measured uptime exceeds the SLO (i.e., if some of the error budget remains)

#### 3.4.2 Benefits

- the halt of releases incentivizes product development to self-police
- outages with external causes (network problems) may also eat into error budget; since product also wants/needs uptime, this is not treated specially

# 4 Service Level Objectives

## 4.1 Service Level Terminology

### 4.1.1 Indicators

- A carefully defined quantitative measure of some aspect of the level of service that is provided
  - > request latency, error rate, system throughput
  - > aggregated over a measurement window and turned into a rate, average, or percentile
  - > availability: fraction of time that a service is usable
  - > yield: fraction of well-formed requests that succeed
  - > durability: likelihood that data will be retained over a long period of time

### 4.1.2 Objectives

- a target value or range of values for a service level that is measured by an SLI
- naturally expressible as $SLI <= target$ or $lower\ bound <= SLI <= upperbound$

### 4.1.3 Agreements

- An explicit or implicit contract with your users that includes consequences of meeting (or missing) the SLOs they contain

## 4.2 Indicators in Practice

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

- A handful of representative indicators are enough to approximate system health
- Broad system categories
  - > User-facing serving systems
    - * availability, latency, and throughput
  - > Storage systems
    - * latency, availability, and durability
  - > Big data systems
    - * throughput, end-to-end latency

#### 4.2.2 Collecting Indicators

#### 4.2.3 Aggregation

- even "requests per second" seems easy but hides complexity
  - > measurmeent obtained once per second, or by averaging requests over a minute (thus hiding bursts)
- usually better to think in terms of distributions
- "User studies have shown that people typically prefer a slightly slower system to one with high variance in response time"

#### 4.2.4 Standardize Indicators

- Template these sorts of assumptions so they don't need to be re-stated all the time
  - > aggregation intervals: "averaged over 1 minute"
  - > aggregation regions: "all the tasks in a cluster"
  - > how frequently measurements are made: "every 10 seconds"
  - > which requests are included: "HTTP GETs from black-box monitoring jobs"
  - > How the data is acquired: "Through our monitoring, measured at the server"
  - > Data-access latency: "Time to last byte"

### 4.3 Objectives in Practice

- For SLOs, try to find out what users actually care about, not what's easy to measure

#### 4.3.1 Defining Objectives

- "For maximum clarity, SLOs should specify how they're measured and the conditions under which they're valid"
- e.g., "99% (averaged over 1 minute) of Get RPC calls witll complete in less than 100ms (measured across all backend servers)"
  - > or a set of SLos, like "90%, 99%, 99.9% of Get RPC calls will complete in less than 1, 10, 100ms, respectively"
- Track the SLO violations on a daily or weekly level to inform an error budget and as a general health check.

#### 4.3.2 Choosing Targets

- Don't pick a target based on current performance – to avoid locking in on a target that's hard to maintain
- Keep it simple - complicated aggregations are hard to instrument and hard to reason about
- Avoid absolutes: avoid infinite load, always available, and other fairy tales
- have as few SLOs as possible - choose a small list of SLOs that are super relevant
  - > "if you can't ever win a conversation about priorities by quoting a particular SLO, it's probably not worth having that SLO"
  - > "if you can't ever win a conversation about SLOs, it's probably not worth having an SRE team for the product"
- Perfection can wait - better to start with a loose target and tighten it than go the other way around
- SLOs should be a driver for prioritizing work for SRE and product dev alike.

#### 4.3.3 Control Measures

- Create monitoring on SLIs to catch possible upcoming SLO violations

#### 4.3.4 SLOs Set Expectations

- To set realistic expectations for your users, consider these tactics:
  - > Keep a safety margin
    - * Use a tighter internal SLO than the published one, so you have a buffer – room to respond to problems you've underestimated, or to take risks in service changes
  - > Don't overachieve
    - * users over-depend on services whose actual performance beats stated SLOs (see: Global Chubby Planned Outage in chapter 3)
    - * some options: deliberately take the system offline occasionally; throttle some requests; design the system so it isn't faster under light loads

### 4.4 Agreements in Practice

## 5 Eliminating Toil

mzabaro TODO: fill this in

## 6 Monitoring Distributed Systems

mzabaro TODO: fill this in

## 7 The Evolution of Automation at Google

### 7.1 The Value of Automation

#### 7.1.1 Consistency

"In this domain–the execution of well-scoped, known procedures–the value of consistency is in many ways the primary value of automation."

#### 7.1.2 A Platform

"Designed and done properly, automatic systems also provide a *platform* that can be extended, applied to more systems, or perhaps even spun out for profit."

- "A platform also *centralizes mistakes*"
- "a bug fixed in the code will be fixed there once and forever"
- "can be extended to perform additional tasks more easily than humans"
- "can run either continuously or much more frequently than humans could"
- "can export metrics about its performance [...] because these details are more easily measurable"

#### 7.1.3 Faster Repairs

"If automation runs regularly and successfully enough, the result is a reduced mean time to repair (MTTR) for those common faults. You can then spend your time on other tasks instead, thereby achieving increased developer velocity because you don't have to spend time either preventing a problem or (more commonly) cleaning up after it."

#### 7.1.4 Faster Action

"In the infrastructural situations where SRE automation tends to be deployed, humans don't usually react as fast as machines."

#### 7.1.5 Time Saving

"It's easy to overlook the fact that once you have encapsulated some task in automation, anyone can execute the task. Therefore, the time savings apply across anyone who would plausibly use the automation. Decoupling the operator from operation is very powerful."

### 7.2 The Value for Google SRE

### 7.3 The Use Cases for Automation

"automation is "metasoftware"–software to act on software"

#### 7.3.1 Google SRE's Use Cases for Automation

#### 7.3.2 A Hierarchy of Automation Classes

The evolution of automation follows a path

1. *No automation*
   Database master is failed over manually between locations.
2. *Externally maintained system-specific automation*
   An SRE has a failover script in his or her home directory.
3. *Externally maintained generic automation*
   The SRE adds database support to a "generic failover" script that everyone uses.
4. *Internally maintained system-specific automation*
   The database ships with its own failover script.
5. *Systems that don't need any automation*
   The database notices problems, and automatically fails over without human intervension.

## 7.4 Automate Yourself Out of a Job: Automate ALL the Things!

## 7.5 Soothing the Pain: Applying Automation to Cluster Turnups

### 7.5.1 Detecting Inconsistencies with Prodtest

### 7.5.2 Resolving Inconsistencies Idempotently

### 7.5.3 The Inclination to Specialize

Automation processes can vary in three respects:

- *Competence*, i.e., their accuracy
- *Latency*, how quickly all steps are executed when initiated
- *Relevance*, or proportion of real-world processes covered by automation

"Automation code, like unit test code, dies when the maintaining team isn't obsessive about keeping the code in sync with the codebase it covers."

By relieving teams who ran services of the responsibility to maintain and run their automation code, we created ugly organizational incentives:

- A team whose primary task is to speed up the current turnup has no incentive to reduce the technical debt of the service-owning team running the service in production later.
- A team not running automation has no incentive to build systems that are easy to automate.
- A product manager whose schedule is not affected by low-quality automation will always prioritize new features over simplicity and automation.

"The most functional tools are usually written by those who use them. A similar argument applies to why product development teams benefit from keeping at least some operational awareness of their systems in production."

and the more effort we can devote to making those methods as they can possibly be. Again, a recurring theme appears: the conscious decision to not take on certain problems allows us to focus on our core problem and make the solutions we explicitly set out to create substantially better."

addition to making sure that updates to our incident management procedures are clearly communicated to all relevant parties."

"We now require thorough testing of rollback procedures before such large-scale tests."

## 13.3 Change-Induced Emergency

### 13.3.1 Details

### 13.3.2 Response

### 13.3.3 Findings

#### 13.3.3.1 What went well

"Once the problem was detected, incident management generally went well and updates were communicated often and clearly. Our out-of-band communications systems kept everyone connected even while some of the more complicated software stacks were unusable. This experience reminded us why SRE retains highy reliable, low overhead backcup systems, which we use regularly."

"command-line tools and alternative access methods [...] enable us to perform updates and roll back changes even when other interfaces are inaccessible [...] with the caveat that engineers needed to be more familiar with the tools and to test them more routinely."

"Google's infrastructure provided yet another layer of protection in that the affected system rate-limited how quickly it provided full updates to new clients."

#### 13.3.3.2 What we learned

"reinforced the need for thorough canarying, regardless of the perceived risk."

## 13.4 Process-Induced Emergency

### 13.4.1 Details

### 13.4.2 Response

### 13.4.3 Findings

#### 13.4.3.1 What went well

#### 13.4.3.2 What we learned

"Reinstallations of machines were slow and unreliable."

"The machine reinstallation infrastructure was unable to handle the simultaneous setup of thousands of machines. This inability was partly due to a regression that prevented the infrastructure from running more than two setup tasks per worker machine. The regression also used improper QoS settings to transfer files and had poorly tuned timeouts. It forced kernel reinstallation, even on machines that still had the proper kernel and on which Diskerase had yet to occur."

## 13.5 All Problems Have Solutions

## 13.6 Learn from the Past. Don't Repeat It.

### 13.6.1 Keep a History of Outages

### 13.6.2 Ask the Big, Even Improbable Questions: What If...?

### 13.6.3 Encourage Proactive Testing

## 13.7 Conclusion

# 14 Managing Incidents

## 14.1 Unmanaged Incidents

## 14.2 The Anatomy of an Unmanaged Incident

### 14.2.1 Sharp Focus on the Technical Problem

### 14.2.2 Poor Communication

### 14.2.3 Freelancing

## 14.3 Elements of Incident Management Process

> "Incident management skills and practices exist to channel the energies of enthusiastic individuals. Google's incident management system is based on the Incident Command System, which is known for its clarity and scalability."

### 14.3.1 Recursive Separation of Responsibilities

> "a clear separation of responsibilities allows individuals more autonomy than they might otherwise have, since they need not second-guess their colleagues"
>
> Distinct Roles:

- *Incident Command*
  "The incident commander holds the high-level state about the incident. They structure the incident response task force, assigning responsibilities according to need and priority. *De facto*, the commander holds all positions that they have not delegated."
- *Operational work*
  "The operations team should be the only group modifying the system during an incident."
- *Communication*
  "Their duties most definitely include issuing periodic updates to the incident response team and stakeholders (usually via email), and may extend to tasks such as keeping the incident document accurate and up to date."
- *Planning*
  "The planning role supports Ops by dealing with longer-term issues, such as filing bugs, ordering dinner, arranging handoffs, and tracking how the system has diverged from the norm so it can be reverted once the incident is resolved."

#### 14.3.2 A Recognized Command Post

"Interested parties need to understand where they can interact with the incdient commander."

"Google has found IRC to be a huge boon in incident response. IRC is very reliable and can be used as a log of communications about this event, and such a record is invaluable in keeping detailed state changes in mind. We've also written bots that log incident-related traffic (which is helpful for postmortem analysis), and other bots that log events such as alerts to the channel."

#### 14.3.3 Live Incident State Document

"The incident commander's most important responsibility is to keep a living incident document."

#### 14.3.4 Clear, Live Handoff

### 14.4 A Managed Incident

### 14.5 When to Declare an Incident

My team follows these broad guidelines–if any of the following is true, the event is an incident:

- Do you need to involve a second team in fixing the problem?
- Is the outage visible to customers?
- Is the issue unsolved even after an hour's concentrated analysis?

### 14.6 In Summary

## 15 Postmortem Culture: Learning from Failure

"A postmortem is a written record of an incident, its impact, the actions taken to mitigate or resolve it, the root cause(s), and the follow-upactions to prevent the incident from recurring."

### 15.1 Google's Postmortem Philosophy

"The primary goals of writing a postmortem are to ensure that the incident is documented, that all contributing root cause(s) are well understood, and, especially, that effective preventative actions are put in place to reduce the likelihood and/or impact of recurrence."

"Postmortems are expected after any significant undesirable event."

Teams have some internal flexibility, but common postmortem triggers include:

- User-visible downtime or degradation beyond a certain threshold
- Data loss of any kind
- On-call engineer intervention (release rollback, rerouting of traffic, etc.)
- A resolution time above some threshold
- A monitoring failure (which usually implies manual incident discovery)

"When an outage does occur, a postmortem is not written as a formality to be forgotten. Instead the postmortem is seen by engineers as an opportunity not only to fix a weakness, but to make Google more resilient as a whole."

#### 15.1.1 Sidebar: Best Practice: Avoid Blame and Keep It Constructive

### 15.2 Collaborate and share knowledge

Regardless of the specific tool you use, look for the following key features:

- *Real-time collaboration*
  rapid collection of data and ideas
- *An open commenting/annotation system*
  Makes crowdsourcing solutions easy and improves coverage
- *Email notifications*

Writing a postmortem also involves formal review and publication. In practice, teams share the first postmortem draft internally and solicit a group of senior engineers to assess the draft for completeness. Review criteria might include:

- Was key incident data collected for posterity?
- Are the impact assessments complete?
- Was the root cause sufficiently deep?
- Is the action plan appropriate and are resulting bug fixes at appropriate priority?
- Did we share the outcome with relevant stakeholders?

#### 15.2.1 Sidebar: Best Practice: No Postmortem Left Unreviewed

### 15.3 Introducting a Postmortem Culture

#### 15.3.1 Sidebar: Best Practice: Visibly Reward People for Doing the Right Thing

#### 15.3.2 Sidebar: Best Practice: Ask for Feedback on Postmortem Effectiveness

### 15.4 Conclusion and Ongoing Improvements

## 16 Tracking Outages

""Outalator", our outage tracker, [...] is a system that passively receives all alerts sent by our monitoring systems and allows us to annotate, group, and analyze this data."

"Postmortems [...] are only written for incidents with a large impact, so issues that have individually small impact but are frequent and widespread don't fall within their scope. Similarly, postmortems tend to provide useful insights for improving a single service or set of services, but may miss opportunities that would have a small effect in individual cases, or opportunities that have a poor cost/benefit ratio,b ut that would have large horizontal impact."

### 16.1 Escalator

"all alert notifications for SRE share a central replicated system that tracks whether a human has acknowledged receipt of the notification. If no acknowledgement is received after a configured interval, the system escalates to the next configured destination(s)–e.g., from primary on-call to secondary."

## 16.2 Outalator

"Multiple escalating notifications ("alerts") can be combined into a single entity ("incident") in the Outalator. These notifications may be related to the same single incident, may be otherwise unrelated and uninteresting auditable events such as privileged database access, or may be spurious monitoring failures."

### 16.2.1 Aggregation

"While it is worthwhile to attempt to minimize the number of alerts triggered by a single event, triggering multiple alerts is unavoidable in most trade-off calculations between false positives and false negatives."

"The ability to group multiple alerts into a single *incident* is critical in dealing with this duplication. Sending an email saying "this is the same thing as that other thing; they are symptoms of the same incident" works for a given alert: it can prevent duplication of debugging or panic. But sending an email for each alert is not a practical or scalable solution for handling duplicate alerts within a team, let alone between teams or over longer periods of time."

### 16.2.2 Tagging

"Of course, not ever alerting event is an incident. False-positive alerts occur, as well as test events or mistargeted emails from humans. The Outalator itself does not distinguish between these events, but it allows general-purpose *tagging* to add metadata to notifications, at any level."

### 16.2.3 Analysis

"historical information is far more useful when it concerns systemic, periodic, or other wider problems that may exist."

"counting and basic aggregate statistics [...] include information such as incidents per week/month/quarter and alerts per incident. The next layer is more important, and easy to provide: comparison between teams/services and over time to identify first patterns and trends. This layer allows teams to determine whether a given alert load is "normal" relative to their own track record and that of other services. "That's the third time this week" can be good or bad, but knowing whether "it" used to happen five times per day or five times per month allows interpretation."

"The next step in data analysis is finding wider issues, which are not just raw counts but require some semantic analysis. For example, identifying the infrastructure component causing most incidents, and therefore the potential benefit from increasing the stability or performance of this component."

#### 16.2.3.1 Reporting and communication

"the Outalator also supports a "report mode," in which the important annotations are expanded inline with the main list in order to provide a quick overview of lowlights."

### 16.2.4 Unexpected Benefits

"There are additional nonobvious benefits. To use Bigtable as an example, if a service has a disruption due to an apparent Bigtable incient, but you can see that the Bigtable SRE team has not been alerted, manually alerting the team is probably a good idea."

"Some teams across the company have gone so far as to set up dummy escalator configurations: no human receives the notifications send there, but the notifications appear in the Outalator and can be tagged, annotated, and reviewed."

# 17 Testing for Reliability

"One key responsibility of Site Reliablity Engineers is to quantify confidenct in the systems they maintain."

"As the percentage of yoru codebase covered by tests increases, you reduce uncertainty and the potential decrease in reliablity from each change. Adequate testing coverage means that you can make more changes before reliablity falls below an acceptable level. If you make too many changes too quickly, the predicted reliability approaches the acceptability limit. At this point, you may want to stop making changes whil new monitoring data accumulates. The accumulating ddata supplements the tested coverage, which validates the reliability being asserted for revised execution paths."

Sidebar: Relationships between Testing and Mean Time to Repair "It's possible for a testing system to identify a bug with zero MTTR [(when a test detects the exact same problem that monitoring would detect)]. Such a test enables the push to be blocked so the bug never reaches production. [...] THe more bugs you can find with zero MTTR, the higher the *Mean Time Between Failures* (MTBR) experienced by your users."

## 17.1 Types of Software Testing

### 17.1.1 Traditional Tests

#### 17.1.1.1 Unit tests

"A *unit test* is the smallest and simplest form of software testing. These tests are employed to assess a separable unit of software, such as a class or function, for correctness independent of the larger software system that contains the unit."

#### 17.1.1.2 Integration tests

#### 17.1.1.3 System tests

A *system test* is the largest scale test that engineers run on an undeployed system. All modules belonging to a specific component, such as a server that passed integration tests, are assembled into the system. Then the engineer tests the end-to-end functionality of the system. System tests come in many different flavors:

- *Smoke tests*
  *Smoke tests*, in which engineers test very simple but critical behavior, are among the simplest type of system tests. Smoke tests are also known as *sanity testing*, and serve to short-circuit additional and more expensive testing.
- *Performance tests*
  A performance test ensures that over time, a system doesn't degrade or become too expensive.
- *Regression tests*

### 17.1.2 Production Tests

"Production tests interact with a live production system, as opposed to a aystem in a hermetic testing environment."

#### 17.1.2.1 Configuration test

"For each configuration file, a separate *configuration test* examines production to see how a particular binary is actually configured and reports discrepancies against that file. Such tests are inherently not hermetic, as they operate outside the test infrastructure sandbox."

"These nonhermetic configuration tests tend to be especially valuable as part of a distributed monitoring solution since the pattern of passes/fails across production can identify paths through the service stack that don't have sensible combinations of the local configurations."

"Any matches found by the rules become alerts that ongoing releases and/or pushes are not proceeding safely and remedial action is needed."

#### 17.1.2.2 Stress test

"In many cases, individual components don't gracefully degrade beyond a certain point–instead, they catastrophically fail. Engineers use *stress tests* to find the limits on a web service."

#### 17.1.2.3 Canary test

"A canary test isn't really a test; rather, it's structured user acceptance. Whereas configuration and stress tests confirm the existence of a specific condition over deterministic software, a canary test is more ad hoc. It only exposes the code under test to less predictable live production traffic, adn thus, it isn't perfect and doesn't always catch newly introduced faults."

> consider a given underlying fault that relatively rarely impacts user traffic and is being deployed with an upgrade rollout that is exponential. We expect a growing cumulative number of reported variances $CU = RK$ where $R$ is the rate of those reports, $U$ is the order of the fault (defined later), and $K$ is the period over which the traffic grows by a factor of $e$, or 172
>
> In the short time it takes automation to observe the variances and respond, it is likely that several additional reports will be generated. Once the dust has settled, these reports can estimate both the cumulative number $C$ and rate $R$.
>
> Dividing and correcting for $K$ gives an estimate of $U$, the order of the underlying fault. Some examples:
>
> - $U = 1$: The user's request encountered code that is simply broken.
> - $U = 2$: This user's request randomly damages data that a future user's request may see.
> - $U = 3$: The randomly damaged data is also a valid identifier to a previous request.
>
> Most bugs are of order one: they scale linearly with the amount of user traffic. You can generally track doewn these bugs by converting logs of all requests with unusual responses into new regression tests.

"Keeping the dynamics of higher- versus lower-order bugs in mind, when you are using an exponential rollout strategy, it isn't necessary to attempt to achieve fairness among fractions of user traffic."

## 17.2 Creating a Test and Build Environment

You can start by asking these questions:

- Can you prioritize the codebase in any way? Can you stack-rank the components of the system you're testing by any measure of importance?
- Are there particular functions or classes that are absolutely mission-critical or business-critical?
- Which APIs are other teams integrating against?

> When the build system notifies engineers about broken code, they should drop all of their other tasks and prioritize fixing the problem. It is appropriate to treat defects this seriously for a few reasons:

- It's usually harder to fix what's broken if there are changes to the codebase after the defect is introduced.
- Broken software slows down the team because they must work around the breakage.
- Release cadencies, such as nightly and weekly builds, lose their value.
- The ability of the team to respond to a request for an emergency release (for example, in response to a security vulnerability disclosure) becomes much more complex and difficult.

The concepts of stability and agility are traditionally in tension in the world of SRE. The last bulletpoint provides an interesting case where stability actually drives agility. When the build is predictably solid and reliable, developers can iterate faster!

## 17.3 Testing at Scale

### 17.3.1 Testing Scalable Tools

#### 17.3.1.1 Sidebar: Barrier Defenses Against Riskky Software

"Software that bypasses the usual heavily tested API (even if it does so for a good cause) could wreak havoc on a live service."

Avoid this risk of havoc with design:

1. Use a separate tool to place a barrier in the replication configuration so that the replica cannot pass its health check. As a result, the replica isn't released to users.
2. Configure the risky software to check for the barrier upon startup. Allow the risky software to only access unhealthy replicas.
3. Use the replica health validating tool you use for black-box monitoring to remove the barrier.

### 17.3.2 Testing Disaster

Many disaster recovery tools can be carefully designed to operate *offline*. Such tools do the following:

- Compute a *checkpoint* state that is equivalent to cleanly stopping the service
- Push the checkpoint state to be *loadable* by existing nondisaster validation tools
- Support the usual release *barrier* tools, which trigger the *clean start* procedure

In many cases, you can implement these phases so that the associated tests are easy to write and offer excellent coverage.

"Online repair tools inherently operate outside the mainstream API and therefore become more interesting to test. One challenge you face in a distributed system is determining if normal behaior, which may be eventually consistent by nature, will interact badly with the repair."

#### 17.3.2.1 Sidebar: Using Statistical Tests

"Statistical techniques, such as Lemon for fuzzing, and Chaos Monkey and Jepsen for distributed state, aren't necessarily repeatable tests. Simply rerunning such tests after a code change doesn't definitively prove that the observed fault is fixed."

### 17.3.3 The Need for Speed

#### 17.3.3.1 Sidebar: Testing Deadlines

"Tests that require orchetration across many binaries and/or across a fleet that has many containers tend to have startup times measured in seconds. Such tests are usually unable to offer interactive feedback, so they can be classified as batch tests. Instead of saying "don't close the editor tab" to the engineer, these test failures are saying "this code is not ready for review" to the code reviewer."

"The informal deadline for the test is the point at which the engineer makes the next context switch."

### 17.3.4 Pushing to Production

### 17.3.5 Expect Testing Fail

"In order to remain reliable and to avoid scaling the number of SREs supporting a service linearly, the production environment has to run mostly unattended. To remain unattended, the environment must be resilient against minor faults. When a major event that demands manual SRE intervention occurs, the tools used by SRE must be suitably tested. Otherwise, that intervention decreases confidence that historical data is applicable to the near future. The reduction in confidence requires waiting for an analysis of monitoring data in order to eliminate the uncertainty incurred."

When viewed from the point of view of reliability:

- A configuration that exists to keep MTTR low, and is only modified when there's a failure, has a release cadence slower than the MTBF. There can be a fair amount of uncertainty as to whether a given manual edit is actually truly optimal without the edit impacting the overall site reliability.
- A configuration file that changes more than once per user-facing application release (for example, because it holds the release state) can be a major risk if these changes are not treated the same as application releases. If testing and monitoring coverage of that configuration file is not considerably better than that of the user application, that file will dominate site reliability in a negative way.

### 17.3.6 Integration

"The role of SRE generally includes writing systems engineering tools. Not because software engineers shouldn't write them. Tools that cross between technology verticals and span abstraction layers tend to have weak associations with many software teams and a slightly stronger association with systems teams."

"A key element of delivering site reliability is finding each anticipated form of misbehaior and making sure that some test (or another tool's tested input validator) reports that misbehavior. The tool that finds the problem might not be able to fix or even stop it, but should at least report the problem before a catastrophic outage occurs."

### 17.3.7 Production Probes

Given that testing specifies acceptable behavior in the face of known data, while monitoring confirms acceptable bbehavior in the face of unknown user data, it would seem that major sources of risk–both the known and the unknown–are covered by the combination of testing and monitoring. Unfortunately, actual risk is more complicated.

Known good requests should work, while known bad requests should error. Implementing both kinds of coverage as an integration test is generally a good idea. You can replay the same bank of test requests as a release test. Splitting the known good requests into those that can be replayed against production and those that can't yields three sets of requests:

- Known bad requests
- Known good requests that can be replayed against production
- Known good requests that can't be replayed against production

You can use each set as both integration and release tests. Most of these tests can also be used as monitoring probes.

It would seem to be superfluous and, in principle, pointless to deploy such monitoring because these exact same reqeusts have already been tried two other ways. However, those two ways were different for a few reasons:

- The release test probably wrapped the integrated server with a frontend and a fake backend.
- The probe test probably wrapped the release binary with a load balancing frontend and a separate scalable persistent backend.
- Frontends and backends probably have independent release cycles. It's likely that the schedules for those cycles occur at different rates (due to their adaptive release cadences).

Therefore, the monitoring probe running in production is a configuration that wasn't previously tested.

Those probes should never fail, but what does it mean if they do fail? Either the frontend API (from the load balancer) or the backend API (to the persistent store) is not equivalent between the production and release environments. Unless you already know why the production and release environments aren't equivalent, the site is likely broken.

### 17.4 Conclusion

"You can't fix a problem until you understand it, and in engineering, you can only understand a problem by measuring it."

## 18 Software Engineering in SRE

### 18.1 Why Is Software Engineering Within SRE Important?

SREs are in a unique position to effectively develop internal software for a number of reasons:

- The breadtha nd depth of [...] production knowledge [...] allows its [SREs] to design and create [...] for [...] scalability, graceful degradation during failure, and the ability to easily interface with other infrastructure or tools.
- Because SREs are embedded in the subject matter, they easily understand the needs and requirements of the tool being developed.
- A direct relationship with the intended user–fellow SREs–results in frank and high-signal user feedback. Releasing a tool to an internal audience with high familiarity with the problem space means that a development team can launch and iterate more quickly. Internal users are typically more understanding when it comes to minimal UI and other alpha product issues.

"Beyond the design of automation tools and other efforts to reduce the workload for engineers in SRE, software development projects can further benefit the SRE organization by attracting and helping to retain engineers with a broad variety of skills. The desirability of team diversity is doubly true for SRE, where a variety of backgrounds and problem-solving approaches can help prevent blind spots. To this end, Google always strives to staff its SRE teams with a mix of engineers with traditional software development experience and engineers with systems engineering experience."

## 18.2 Auxon Case Study: Project Background and Problem Space

### 18.2.1 Traditional Capacity Planning

"capacity planning is a neverending *cycle*"

#### 18.2.1.1 Brittle by nature

Traditional capacity planning produces a resource allocation plan that can be disrupted by any seemingly minor change. For example:

- A service undergoes a decrease in efficiency, and needs more resources than expected to serve the same demand.
- Customer adoption rates increase, resulting in an increase in projected demand.
- The delivery date for a new cluster of compute resources slips.
- A product decision about a performance goal changes the shape of the required service deployment (the service's footprint) and the amount of required resources.

#### 18.2.1.2 Laborious and imprecise

"when it is time to find capacity to meet this future demand, not all resources are equally suitable. For example, if latency requirements mean that a service must commit to serve user demand on the same continent as the user, obtaining additional resources in North America won't alleviate a capacity shortfall in Asia. Every forecast has *constraints*, or parameters around how it can be fulfilled; constraints are fundamentally related to intent"

"When service owners face the challenges of fitting a series of requests for capacity from various services into the resources available to them, in a manner that meets the various constraints a servie may have, additional imprecision ensues. Bin packing is an NP-hard problem that is difficult for human beings to compute by hand. Furthermore, the capacity request from a service is generally an inflexible set of demand requirements: $X$ cores in cluster $Y$. The reasons why $X$ cores in or $Y$ cluster are needed, and any degrees of freedom around those parameters, are long lost by the time the request reaches a human trying to fit a list of demands into available supply."

### 18.2.2 Our Solution: Intent-Based Capacity Planning

"*Intent-based Capacity Planning.* The basic premise of this approach is to programmatically encode the dependencies and parameters (*intent*) of a service's needs, and use that encoding to autogenerate an allocation plan that details which resources go to which service, in which cluster. If demand, supply, or service requirements change, we can simply autogenerate a new plan in response to the changed parameters, which is now the new best distribution of resources."

"With a service's true requirements and flexibility captured, the capacity plan is now dramatically more nimble in the face of change, and we can reach an optimal solution that meets as many parameters as possible."

## 18.3 Intent-Based Capacity Planning

### 18.3.1 Precursorts to Intent

#### 18.3.1.1 Dependencies

"imagine user-facing service Foo, which depends on Bar, an infrastructure storage service. Foo expresses a requirement that Bar must be located within 30 milliseconds of network latency of Foo. This requirement has important repercussions for where we place both Foo *and* Bar, and intent-driven capacity planning must take these constraints into account."

#### 18.3.1.2 Performance metrics

"Understanding the chain of dependencies helps formulate the general scope of the bin packing problem, but we still need more information about expected resource usage. How many compute resources does service Foo need to serve $N$ user queries? For every $N$ queries of service Foo, how many Mbps of data do we expect for service Bar?"

"Performance metrics are the glue between dependencies. They convert from one or more higher-level resource type(s) to one or more lower-level resource type(s)."

#### 18.3.1.3 Prioritization

### 18.3.2 Introduction to Auxon

"Auxon is Google's implementation of an intent-based capacity planning and resource allocation solution"

"Auxon provides the means to collect intent-based descriptions of a service's resource requirements and dependencies. These user intents are expressed as requirements for how the owner would like the service to be provisioned."

"Requirements can be prioritized, a feature that's useful if resources are insufficient to meet all requirements, and therefore trade-offs must be made. These requirements–the intent–are ultimately represented internally as a giant mixed-integer or linear program. Auxon solves the linear program, and uses the resultant bin packing solution to formulate an allocation plan for resources."

"*Performance Data* describes how a service scales: for every unit of demand $X$ in cluster $Y$, how many units of dependency $Z$ are used? This scaling data may be derived in a number of ways depending on the maturity of the service in question. Some services are load tested, while others infer their scaling based upon past performance."

"*Per-Service Demand Forecast Data* describes the usage trend for forecasted demand signals. Some services derive their future usage from demand forecasts–a forecast of queries per second broken down by continent. Not all services have a demand forecast: some services (e.g., a storage service like Colossus) derive their demadn purely from services that depend upon them."

### 18.3.3 Requirements and Implementation: Successes and Lessons Learned

#### 18.3.3.1 Approximation

### 18.3.4 Raising Awareness and Driving Adoption

"As with any product, SRE-developed software must be designed with knowledge of its users and requirements. It needs to drive adoption through utility, performance, and demonstrated ability to both benefit Google's production reliability goals and to better the lives of SREs. The process of socializing a product and achieving buy-in across an organization is key to the project's success."

#### 18.3.4.1 Set expectations

"it's important to differentiate aspirational goals of the product from minimum success criteria (or Minimum Viable Product). Projects can lose credibility and fail by promising too much, too soon; at the same time, if a product doesn't promise a sufficiently rewarding outcome, it can be difficult to overcome the necessary activation energy to convince internal teams to try something new. Demonstrating steady, incremental progress via small releases raises user confidence in your team's ability to deliver useful software."

#### 18.3.4.2 Identify appropriate customers

#### 18.3.4.3 Customer service

#### 18.3.4.4 Designing at the right level

### 18.3.5 Team Dynamics

## 18.4 Fostering Software Engineering in SRE

"Because Google SRE teams are currently organized around the services they run, SRE-developed projects are particularly at risk of being overly specific work that only benefits a small percentage of the organization. Because team incentives are aligned primarily to provide a great experience for the users of one particular service, projects often fail to generalize to a broader use case as standardization across SRE teams comes in second place. At the opposite end of the spectrum, overly generic frameworks can be equally problematic; if a tool strives to be too flexible and too universal, it runs the risk of not quite fitting any use case, and therefore having insufficient value in and of itself. Projects with grand scope and abstract goals often require significant development effort, but lack the concrete use cases required to deliver end-user benefit on a reasonable time frame."

### 18.4.1 Successfully Building a Software Engineering Culture in SRE: Staffing and Development Time

### 18.4.2 Getting There

"In order to develop useful software, you're effectively creating a product team. That team includes required roles and skills that your SRE organization may not have formerly demanded. Will someone play the role of product manager, acting as the customer advocate?"

## 18.5 Conclusions

# 19 Load Balancing at the Frontend

"This chapter focuses on high-level load balancing–how we balance user traffic *between* datacenters"

## 19.1 Poser Isn't the Answer

Let's start by reviewing two common traffic scenarios: a basic search request and a video upload request. Users want to get their query results quickly, so the most important variable for the search request is latency. On the other hand, users expect video uploads to take a non-negligible amount of time, but also want such requests to succeed the first time, so the most important variable for the video upload is throughput. The differing needs of the two requests play a role in how we determine the optimal distribution for each request at the *global* level:

- The search request is sent to the nearest available datacenter–as measured in round-trip time (RTT)–because we want to minimize the latency on the request.
- The video upload stream is routed via a different path–perhaps to a link that is currently underutilized–to maximize the throughput at the expense of latency.

But on the *local* level, inside a given datacenter, we often assume that all machines within the building are equally distant to the user and connected to the same network. Therefore, optimal distribution of load focuses on optimal resource utilization and protecting a single server from overloading.

### 19.2 Load Balancing Using DNS

"The simplest soution is to return multiple A or AAAA records in the DNS reply and let the client pick an IP address arbitrarily."

"provides very little control over the client behavior: records are selected randomly, and each will attract a roughly equal amount of traffic. Can we mitigate this problem? In theory, we could use SRV records to specify record weights and priorities, but SRV records have not yet been adopted for HTTP."

"the client cannot determine the closest address. We *can* mitigate this scenario by using an anycast address for authoritative nameservers and leverage the fact that DNS queries will flow to the closest address."

> end users rarely talk to authoritative nameservers directly. Instead, a recursive DNS server usually lies somewhere between end users and nameservers. This server proxies queries between a user and a server and often provides a caching layer. The DNS middleman has three very important implications on traffic management:

- Recursive resolution of IP addresses
  > "only allows reply optimization for the shortest distance between resolver and the nameserver."
  > A possible solution is to use the EDNS0 extension (not yet an official standard)
- Nondeterminic reply paths
  > "[A nonauthoritative] nameserver may be responsible for serving thousands or millions of users, across regions varying from a single office to an entire continent."
- Additional caching complications

mzabaro TODO: fill this in

## 20 Load Balancing in the Datacenter

"Assume there is a stream of queries arriving to the datacenter–these could be coming from the datacenter itself, remote datacenters, or a mix of both–at a rate that doesn't exceed the resources that the datacenter has to process them (or only exceeds it for very short amounts of time). Also assume that there are *services* within the datacenter, against which these queries operate. These services are implemented as many homogeneous, interchangeable server processes mostly running on different machines. The smallest services typically have at least three such processes (using fewer processes means losing 50% or more of your capacity if you lose a single machine) and the largest may have more than 10,000 processes (depending on datacenter size). In the typical case, services are composed of between 100 and 1,000 processes. We call these processes *backend tasks* (or just *backends*). Other tasks, known as *client tasks* hold connections to the backend tasks. For each incoming query, a client task must decide which backend task should handle the query. Clients communicate with backends using a protocol implemented on top of a combination of TCP and UDP"

### 20.1 The Ideal Case

"In the ideal case, the load for a given service is spread perfectly over all its backend tasks and, at any given point in time, the least and most loaded backend tasks consume exactly the same amount of CPU."

"More formally, let $CPU[i]$ be the CPU rate consumed by task $i$ at a given point of time, and suppose that task 0 is the most loaded task. Then, in the case of a large spread, we are wasting the sum of the differences in the CPU from any task to $CPU[0]$: that is, the sum over all tasks $i$ of $CPU[0] - CPU[i]$ will be wasted. In thsi case, "wasted" means reserved, but unused."

## 20.2 Identifying Bad Tasks: Flow Control and Lame Ducks

### 20.2.1 A Simple Approach to Unhealthy Tasks: Flow Control

"Assume our client tasks track the number of active requests they have sent on each connection to a backend task. When this active-request count reaches a configured limit, the client treats the backend as unhealthy and no longer sends it requests."

### 20.2.2 A Robust Approach to Unhealthy Tasks: Lame Duck State

From a client perspective, a given backend task can be in one of 3 states: Healthy (accepting and processing requests); Refusing Connections (entirely unresponsive); Lame Duck (accepting and processing requests, but explicitly asking clients to stop sending requests).

"When a task enters lame duck state, it broadcasts that fact to all its active clients."

"With Google's RPC implementation, inactive clients (i.e., clients with no active TCP connections) still send periodic UDP health checks"

"it simplifies clean shutdown, which avoids serving errors to all the unlucky requests that happened to be active on backend tasks that are shutting down."

"[This] facilitates code pushes, maintenance activities, or machine failures that may require restarting all related tasks."

## 20.3 Limiting the Connections Pool with Subsetting

"*Subsetting*: limiting the pool of potential backend tasks with which a client interacts."

"Each client in our RPC system maintains a pool of long-lived connections to its backends that it uses to send new requests. These connections are typically established early on as the client is starting and usually remain open, with requests flowing through them, until the client's death. An alternative model would be to establish and tear down a connection for each request, but this model has significant resource and latency costs."

"Subsetting avoids the situation in which a single client connects to a very large number of backend tasks or a single backend task receives connections from a very large number of client tasks."

### 20.3.1 Picking the Right Subset

"The "right" subset size for a system depends heaviliy on the typical behavior of your service."

Use a large subset size if:

- The number of clients is significantly smaller than the number of backends.
- There are frequent load imbalances within the client jobs. Because a burst of requests will be concentrated in the client's assigned subset, you need a larger subset size to ensure the load is spread evenly across the larger set of available backend tasks.

"The selection algorithm for clients should assign backends uniformly to optimize resource provisioning. For example, if subsetting overloads one backend by 10%, the whole set of backends needs to be overprovisioned by 10%. The algorithm should also handle restarts and failures gracefully and robustly by continuing to load backends as uniformly as possible while minimizing churn."

"The algorithm should also handle resizes in the number of clients and/or number of backends, with minimal connection churn and without knowing these numbers in advance. This functionality is particularly important (and tricky) when the entire set of client or backend tasks are restarted one at a time (e.g., to push a new version)."

### 20.3.2 A Subset Selection Algorithm: Random Subsetting

This spreads load very unevenly.

"We concluded that for random subsetting to spread the load relatively evenly across all available tasks, we would need subset sizes as large as 75%. A subset that large is simply impractical"

### 20.3.3 A Subset Selection Algorithm: Deterministic Subsetting

```
def Subset(backends, client_id, subset_size):
    subset_count = len(backends) / subset_size

    # Group clients into rounds: each round uses the same shuffled list:
    round =client_id / subset_count
    random.seed(round)
    random.shuffle(backends)

    # The subset id corresponding to the current client
    subset_id = client_id % subset_count

    start = subset_id * subset_size
    return backends[start:start + subset_size]
```

## 20.4 Load Balancing Policies

### 20.4.1 Simple Round Robin

"we've found that Round Robin can result in a spread of up to 2x in CPU consumption from the least to the most loaded task. Such a spread is extremely wasteful and occurs for a number of reasons"

#### 20.4.1.1 Small subsetting

In the case of vastly different processes (different request rates and loads), and especially if you're using small subset sizes, backends in the subsets of the clients generating the most traffic will naturally tend to be more loaded.

#### 20.4.1.2 Varying query costs

"It can become necessary to adjust the service interfaces to functionally cap the amount of work done per request"

"To keep interfaces (and their implementations) simple, services are often defined to allow the most expensive requests to consume 100, 1,000, or even 10,000 times more resources than the cheapest requests. However, varying resource requirements per-request naturally mean that some backend tasks will be unlucky and occasionally receive more expensive requests than others"

#### 20.4.1.3 Machine diversity

"Dealing with machine diversity–*without* requiring strict homogeneity–was a challenge for many years at Google. In theory, the solution to working with heterogeneous resource capacity in a fleet is simple: scale the CPU reservations depending on the processor/machine type. However, in practice, rolling out this solution required significant effort because it required our job scheduler to account for resource equivalencies based on average machine performance across a sampling of services."

#### 20.4.1.4 Unpredictable performance factors

The performance of backend tasks may differ vastly due to several *unpredictable* aspects that cannot be accounted for statically

- Antagonistic neighbors
  > "For example, if the latency of outgoing requests from a backend task grows (because of competition for network resources with an antagonistic neighbor), the number of active requests will also grow, which may trigger increased garbage collection."
- Task restarts
  > "When a task gets restarted, it often requires significantly more resources for a few minutes"
  > Think of dynamic code optimization (JIT compilation in Java)
  > "we keep servers in lame duck state and pre-warm them (triggering these optimizations) for a period of time after they start, until their performance is nominal."

### 20.4.2 Least-Loaded Round Robin

"use Round Robin *among the set of tasks with a minimal number of active requests*"

"one very dangerous pitfall of the Least-Loaded Round Robin approach: if a task is seriously unhealthy, it might start serving 100% errors. Depending on the nature of those errors, they may have very low latency: it's frequently significantly faster to just return an "I'm unhealthy!" error than to actually process a request. As a result, clients might start sending a very large amount of traffic to the unhealthy task, erroneously thinking that the task is available, as opposed to fast-failing them! We say that the unhealthy task is now *sinkholing* traffic. Fortunately, this pitfall can be solved relatively easily by modifying the policy to count recent errors as if they were active requests."

Two main limitations

- The count of active requests may not be a very good proxy for the capability of a given backend
- The count of active requests in each client doesn't include requests from other clients to the same backends (aka, each client task has only a very limited view into the state of its backend tasks)

### 20.4.3 Weighted Round Robin

"each client task keeps a "capability" score for each backend in its subset"

"In each response (including responses to health checks), backends include the current observed rates of queries and errors per second, in addition to the utilization (typically, CPU usage). Clients adjust the capability scores periodically to pick backend tasks based upon their current number of successful requests handled and at what utilization cost; failed requests result in a penalty that affects future decisions."

# 21 Handling Overload

## 21.1 The Pitfalls of "Queries per Second"

"We often speak about the *cost* of a request to refer to a normalized measure of how much CPU tie it has consumed (over different CPU architectures, with consideration of performance differences.

In a majority of cases (although certainly not in all), we've found that simply using CPU consumption as the signal for provisioning works well"

## 21.2 Per-Customer Limits

"When global overload *does* occur, it's vital that the service only delivers error responses to misbehaving customers, while other customers remain unaffected"

## 21.3 Client-Side Throttling

"When a customer is out of quota, a backend task should reject requests quickly with the expectation that returning a "customer is out of quota" error consumes significantly fewer resources than actually processing the request and serving back a correct response. However, this logic doesn't hold true for all services."

"even in the case where rejecting requests saves significant resources, those requests *still* consume some resources. If the amount of rejected requests is significant, these numbers add up quickly. In such cases, the backend can become overloaded even though the vast majority of its CPU is spent just rejecting requests!"

"Whena client detects that a significant portion of its recent requests have been rejected due to "out of quota" errors, it starts self-regulating and caps the amount of outgoing traffic it generates. Requests above the cap fail locally without even reaching the network."

Adaptive throttling:

- Each client keeps the last two minutes of history
- requests: the number of requests attempted by the application layer (whether accepted by the backend, rejected by the backend, or rejected client-side)
- accepts: the number of requests accepted by the backend
- Under normal conditions the values are the same, but as the number of rejects diverges from the number of accepts, eventually self-regulation kicks in.
- Client-side rejection probability: $max(0, \frac{requests-(K\times accepts)}{requests+1})$
- Reducing the multiplier $K$ will let more requests pass through in overload situations

"client-side throttling may not work well with clients that only very sporadically send requests to their backends."

## 21.4 Criticality

"*Criticality* is another notion that we've found very useful in the context of global quotas and throttling."

- CRITICAL_PLUS: requests whose failure would result in serious user-visible impact
- CRITICAL: the default. Failure would have user-visible effects of lower severity
- SHEDDABLE_PLUS: traffic which is okay with partial unavailability. the default for batch jobs, where retries are fine.
- SHEDDABLE: traffic which is okay with occasional full unavailability and regular partial unavailability

"We've made criticality a first-class notion of our RPC system and we've worked hard to integrate it into many of our control mechanisms so it can be taken into account when reacting to overload situations."

- the per-customer limits can be set per criticality level
- "When a task is itself overloaded, it will reject requests of lower criticalities sooner"

- "The adaptive throttling system also keeps separate stats for each criticality"

  "The criticality of a request is orthogonal to its latency requirements and thus to the underlying network quality of service (QoS) used."

  "We've also significantly extended our RPC system to propagate criticality automatically"

## 21.5 Utilization Signals

"Our implementation of task-level overload protection is based on the notion of *utilization*. In many cases, the utilization is just a measurement of the CPU rate (i.e., the current CPU rate divided by the total CPUs reserved for the task)."

"To find the executor load average, we count the number of active threads in the process. In this case, "active" refers to threads that are currently running or ready to run and waiting for a free processor. We smooth this value with exponential decay and begin rejecting requests as the number of active threads grows beyond the number of processors availabile to the task. That means that an incoming request that has a very large fan-out (i.e., one that schedules a burst of a very large number of short-lived operations) will cause the load to spike very briefly, but the smoothing will mostly swallow that spike. However, if the operations are not short-lived (i.e., the load increases and remains high for a significant amount of time) the task will start rejecting requests."

## 21.6 Handling Overload Errors

"If a large subset of backend tasks in the datacenter are overloaded, requests should not be retried and errors should bubble up all the way to the caller (e.g., returning an error to the end user). It's much more typical that only a small portion of tasks become overloaded, in which case the preferred response is to retry the request immediately."

### 21.6.1 Deciding to Retry

"When a client receives a "task overloaded" error response, it needs to decide whether to retry the request."

*per-request retry budget*: if a request has already failed 3 times, let it bubble up to the caller

*per-client retry budget*: "Each client keeps track of the ratio of requests that correspond to retries. A request will only be retried as long as this ratio is below 10%"

"A third approach has clients include a counter of how many times the request has already been tried in the request metadata." "Backends keep histograms of these values in recent history. When a backend needs to reject a request, it consults these histograms to determine the likelihood that other backend tasks are also overloaded. If these histograms reveal a significant amount of retries (indicating that other backend tasks are likely also overloaded), they return an "overloaded: don't retry" error response instead of the standard "task overloaded" error that triggers retries."

## 21.7 Load from Connections

"our RPC protocol requires inactive clients to perform periodic health checks. After a connection has been idle for a configurable amount of time, the client drops its TCP connection and switches to UDP for health checking. Unfortunately, this behavior is problematic when you have a very large number of client tasks that issue a very low rate of requests: health checking on the connections can require more resources than actually serving the requests."

"Handling bursts of new connection requests is a second (but related) problem. We've seen bursts of this type happen in the case of very large batch jobs that create a very large number of worker client tasks all at once."

Mitigation

- "Expose the load to the cross-datacenter balancing algorithm"
- "Mandate that batch client jobs use a separate set of *batch proxy* backend tasks that do nothing but forward requests to the underlying backends and hand their responses back to the clients in a controlled way."

  > "Effectively, the batch proxy acts like a fuse"

### 21.8 Conclusions

"a backend task provisioned to serve a certain traffic rate should continue to serve traffic at that rate without any significant impact on latency, regardless of how much excess traffic is thrown at the task. As a corollary, the backend task should not fall over and crash under the load. These statements should hold true up to a certain rate of traffic–somewhere above 2x or even 10x what the task is provisioned to process."

"It's a common mistake to assume that an overloaded backend should turn down and stop accepting all traffic."

"A well-behaved backend, supported by robust load balancing policies, should accept only the requests that it can process and reject the rest gracefully."

## 22 Addressing Cascading Failures

### 22.1 Causes of Cascading Failures and Designing to Avoid Them

#### 22.1.1 Server Overload

#### 22.1.2 Resource Exhaustion

##### 22.1.2.1 CPU

"If there is insufficient CPU to handle the request load, typically all requests become slower."

Typical secondary effects

- Increased number of in-flight requests (leading toward exhaustion of other resources)
- Excessively long queue lengths
- Thread exhaustion
- CPU or request starvation

  > Internal watchdogs ("often implemented as a thread that wakes up periodically to see whether work has been done since the last time it checked. If not, it assumes that the server is stuck and kills it.") may crash the server

- Missed RPC deadlines

  > "As a server becomes overloaded, its responses to RPCs from its clients arrive later, which may exceed any deadlines those clients set. The work the server did to respond is then wasted, and clients may retry the RPCs, leading to even more overload"

- Reduced CPU caching benefits

#### 22.1.2.2 Memory

Secondary effects

- Dying tasks (evicted by managing software for exceeding hard limits)
- Increased rate of garbage collection, resulting in increased CPU usage ("GC death spiral")
- Reduction in cache hit rates

#### 22.1.2.3 Threads

#### 22.1.2.4 File descriptors

#### 22.1.2.5 Dependencies among resources

### 22.1.3 Service Unavailability

## 22.2 Preventing Server Overload

- Load test the server's capacity limits, and test the failure mode for overload
- Serve degraded results to the user (results that are lower-quality and cheaper-to-compute)
- Instrument the server to reject requests when overloaded (fail early and cheaply)
- Instrument higher-level systems to reject requests, rather than overloading servers
- Perform capacity planning

### 22.2.1 Queue Management

"Most thread-per-request servers use a queue in front of a thread pool to handle requests. Requests come in, they sit on a queue, and then threads pick requests off the queue and perform the actual work (whatever actions are required by the server). Usually, if the queue is full, the server will reject new requests."

### 22.2.2 Load Shedding and Graceful Degradation

"*Load shedding* drops some proportion of load by dropping traffic as the server approaches overload conditions. The goal is to keep the server from running out of RAM, failing health checks, serving with extremely high latency, or any of the other symptoms associated with overload, while still doing as much useful work as it can."

"Changing the queuing method from the standard *first-in, first-out* (FIFO) to *last-in, first-out* or using the *controlled delay* (CoDel) algorithm or similar approaches can reduce load by removing requests that are unlikely to be worth processing. If a web user's search is slow because an RPC has been queued for 10 seconds, there's a good chance the user has given up and refreshed their browser, issuring another request: there's no point in responding to the first one, since it will be ignored!"

"*Graceeful degradation* takes the concept of load shedding one step further by reducing the amount of work that needs to be performed."

"Graceful degradation shouldn't trigger very often–usually in cases of a capacity planning failure or unexpected load shift. Keep the system simple and understandable, particularly if it isn't used often."

"Remember that the code path you never use is the code path that (often) doesn't work. In steady-state operation, graceful degradation mode won't be used, implying that you'll have much less operational experience with this mode and any of its quirks, which *increases* the level of risk. You can make sure that graceful degradation stays working by regularly running a small subset of servers near overload in order to exercise this code path."

"Monitor and alert when too many servers enter these modes."

#### 22.2.3 Retries

"If the backend spends a significant amount of resources processing requests that will ultimately fail due to overload, then the retries themselves may be keeping the backend in an overloaded mode."

"The backend servers themselves may not be stable. Retries an amplify the effects"

When issuing automatic retries

- "Always use randomized exponential backoff when scheduling retries."
- "Limit retries per request. Don't retry a given request indefinitely."
- "Consider having a server-wide retry budget. For example, only allow 60 retries per minute in a process, and if the retry budget is exceeded, don't retry; just fail the request. This strategy can contain the retry effect and be the difference between a capacity planning failure that leads to some dropped queries and a global cascading failure."
- "Think about the service holistically and decide if you really need to perform retries at a given level. In particular, avoid amplifying retries by issuing retries at multiple levels: a single request at the highest layer may produce a number of attempts as large as the *product* of the number of attempts at each layer to the lowest layer. If the database can't service requests because it's overloaded, and the backend, frontend, and JavaScript layers all issue 3 retries (4 attempts), then a single user action may create 64 attempts (4**3) on the database. This behavior is undesirable when teh database is returning those errors because it's overloaded."
- "Use clear response codes and consider how different failure modes should be handled. For example, separate retriable and nonretriable error conditions. Don't retry permanent errors or malformed requests in a client, because neither will ever succeed. Return a specific status when overloaded so that clients and other layers back off and do not retry."

#### 22.2.4 Latency and Deadlines

"When a frontend sends an RPC to a backend server, the frontend consumes resources waiting for a reply. RPC deadlines define how long a request can wait before the frontend gives up, limiting the time that the backend may consume the frontend's resources."

##### 22.2.4.1 Picking a deadline

"High deadlines can result in resource consumption in higher levels of the stack when lower levels of the stack are having problems. Short deadlines can cause some more expensive requests to fail consistently."

##### 22.2.4.2 Missing deadlines

"A common theme in many cascading outages is that servers spend resources handling requests that will exceed their deadlines on the client. As a result, resources are spent while no progress is made: you don't get credit for late assignments with RPCs."

"If handling a request is performed over multiple stages (e.g., there are a few callbacks and RPC calls), the server should check the deadline left at each stage before attempting to perform any more work on the request."

#### 22.2.4.3 Deadline propagation

"Rather than inventing a deadline when sending RPCs to backends, servers should employ deadline propagation and cancellation propagation."

"With deadline propagation, a deadline is set high in the stack (e.g., in the frontend). The tree of RPCs emanating from an initial request will all have the same absolute deadline."

"consider setting an upper bound for outgoing deadlines. You may want to limit how long the server waits for outgoing RPCs to noncritical backends, or for RPCs to backends that typically complete in a short duration."

"Propagating cancellations avoids the potential RPC leakage that occurs if an initial request has a long deadline, but RPCs between deeper layers of the stack have short deadlines and time out."

#### 22.2.4.4 Bimodal latency

## 22.3 Slow startup and Cold Caching

If caching has a significant effect on teh service, try one or some of these strategies:

- "Overprovision the service. It's important to note the distinction between a latency cache versus a capacity cache: when a latency cache is employed, the service can sustain its expected load with an empty cache, but a service using a capacity cache cannot sustain its expected load under an empty cache. Service owners should be vigilant about adding caches to their service, and make sure that any new caches are either latency caches or are sufficiently well engineered to safely function as capacity caches. Sometimes caches are added to a service to improve performance, but actually wind up being hard dependencies."
- Employ cascading failure prevention techniques
- "When adding load to a cluster, slowly increase the load. The initially small request rate warms up the cache; once the cache is warm, more traffic can be added. It's a good idea to ensure that all clusters carry nominal load and that the caches are kept warm."

## 22.4 Always Go Downward in the Stack

intra-layer communication can be problematic for several reasons:

- "The communication is susceptible to a distributed deadlock."
- Bootstrapping the system may become more complex

"It's usually better to avoid inra-layer communication–i.e., possible cycles in the communication path–in the user request path. Instead, have the client do the communication. For example, if a frontend talks to a backend but guesses the wrong backend, the backend should not proxy to the correct backend. Instead, the backend should tell the frontend to retry its request on the correct backend."

## 22.5 Triggering Conditions for Cascading Failures

### 22.5.1 Process Death

### 22.5.2 Process Updates

### 22.5.3 New Rollouts

### 22.5.4 Organic Growth

### 22.5.5 Planned Changes, Drains, or Turndowns

#### 22.5.5.1 Request profile changes

#### 22.5.5.2 Resource limits

## 22.6 Testing for Cascading Failures

### 22.6.1 Test Until Failure and Beyond

"Because of caching effects, gradually ramping up load may yield different results than immediately increasing to expected load levels. Therefore, consider testing both gradual and impulse load patterns."

"You should also test and understand how the component behaves as it returns to nominal load after having been pushed well beyond that load."

"If you're load testing a stateful service or a service that employs caching, yoru load test should track state between multiple interactions and check correctness at high load, which is often where subtle concurrency bugs hit."

### 22.6.2 Test Popular Clients

### 22.6.3 Test Noncritical Backends

"Test your noncritical backends, and make sure their unavailability does not interfere with the critical components of your service."

"text how the frontend behaves if the noncritical backend never responds (for example, if it is blackholing requests)."

## 22.7 Immediate Steps to Address Cascading Failures

### 22.7.1 Increase Resources

### 22.7.2 Stop Health Check Failures/Deaths

"Some cluster scheduling systems, such as Borg, check the health of tasks in a job and restart tasks that are unhealthy. This practice may create a failure mode in which health-checking itself makes the service unhealthy."

"Process health-checking ("is this binary responding *at all*?") and service health checking ("is this binary able to respond *to this class of requests* right now?") are two conceptually distinct operations. Process health checking is relevant to the cluster scheduler, whereas service health checking is relevant to the load balancer. Clearly distinguishing between the two types of health checks can help avoid this scenario."

#### 22.7.3 Restart Servers

"If servers are somehow wedges and not making progress, restarting them may help."

"Make sure that you identify the source of the cascading failure before you restart your servers. Make sure that taking this action won't simply shift around load. Canary this change, and make it slowly."

#### 22.7.4 Drop Traffic

#### 22.7.5 Enter Degraded Modes

#### 22.7.6 Eliminate Batch Load

#### 22.7.7 Eliminate Bad Traffic

### 22.8 Closing Remarks

## 23 Managing Critical State: Distributed Consensus for Reliability

"Systems and software engineers are usually familiar with the traditional ACID datastore semantics (Atomicity, Consistency, Isolation, and Durability), but a growing number of distributed datastore technolgoies provide a different set of semantics known as BASE (Basically Available, Soft state, and Eventual consistency)."

### 23.1 Motivating the User of Consensus: Distributed Systems Coordination Failure

#### 23.1.1 Case Study 1: The Split-Brain Problem

"Each pair of servers has one leader and one follower. The servers monitor each other via heartbeats. If one server cannot contact its partners, it issues a STONITH (Shoot The Other Node In The Head) command to its partner node to shut the node down, and then takes mastership of its files."

#### 23.1.2 Case Study 2: Failover Requires Human Intervention

#### 23.1.3 Cast Study 3: Faulty Group-Membership Algorithms

### 23.2 How Distributed Consensus Works

"When dealing with distributed software systems, we are interested in *asynchronous distributed consensus*, which applies to environments with potentially unbounded delays in message passing. (*Synchronous consensus* applies to real-time systems, in which dedicated hardware means that messages will always be passed with specific timing guarantees.)"

"Distributed consensus algorithms may be *crash-fail* (which assumes that crashed nodes never return to the system) or *crash-recover*. Crash-recover algorithms are much more useful"

"Algorithms may deal with Byzantine or non-Byzantine failures. *Byzantine failure* occurs when a process passes incorrect messages due to a bug or malicious activity, and are comparatively costly to handle, and less often encountered."

### 23.2.1 Paxos Overview: An Example Protocol

## 23.3 System Architecture Patterns for Distributed Consensus

### 23.3.1 Reliable Replicated State Machines

"A *replicated state machine* (RSM) is a system that executes the same set of operations, in the same order, on several processes. RSMs are the fundamental building block of useful distributed systems components and services such as data or configuration storage, locking, and leader election."

### 23.3.2 Reliable Replicated Datastores and Configuration Stores

### 23.3.3 Highly Available Processing Using Leader Election

### 23.3.4 Distributed Coordination and Locking Services

"A *barrier* in a distributed computation is a primitive that blocks a group of processes from proceeding until some condition is met (for example, until all parts of one phase of a computation are completed). Use of a barrier effectively splits a distributed computation into logical phases."

"*Locks* are another useful coordination primitive that can be implemented as an RSM. Consider a distributed system in which worker processes atomically consume some input files and write results. Distributed locks can be used to prevent multiple workers from processing the same input file. In practice, it is essential to use renewable leases with timeouts instead of indefinite locks, because doing so prevents locks from being held indefinitely by processes that crash. Distributed locking is beyond the scope of this chapter, but bear in mind that distributed locks are a low-level systems primitive that should be used with care. Most applications should use a higher-level system that provides distributed transactions."

### 23.3.5 Reliable Distributed Queuing and Messaging

"*Atomic broadcast* is a distributed systems primitive in which messages are received reliably and in the same order by all participants."

"Chandra and Toueg demonstrate the equivalence of atomic broadcast and consensus."

"The *queuing-as-work-distribution* pattern, which uses the queue as a load balancing device [...] can be considered to be point-to-point messaging. [...] Publish-subscribe systems can be used for many types of applications that require clients to subscribe to receive notifications of some type of event. Publish-subscribe systems can also be used to implement coherent distributed caches."

## 23.4 Distributed Consensus Performance

"*Workloads* can vary in many ways and understanding how they can vary is critical to discussing performance. In the case of a consensus system, workload may vary in terms of:"

- "Throughput: the number of proposals being made per unit of time at peak load"
- "The type of requests: proportion of operations that change state"
- "The consistency semantics required for read operations"
- "Request sizes, if size of data payload can vary"

"Deployment strategies vary too. For example:"

- "Is the deployment local area or wide area?"
- "What kinds of quorum are used, and where are the majority of processes?"
- "Does the system use sharding, pipelining, and batching?"

### 23.4.1 Multi-Paxos: Detailed Message Flow

"The Multi-Paxos protocol uses a *strong leader process*: unless a leader has not yet been elected or some failure occurs, it requires only one round trip from the proposer to a quorum of acceptors to reach consensus. Using a strong leader process is optimal in terms of the number of messages to be passed, and is typical of many consensus protocols."

"For systems that use a leader process, the leader election process must be tuned carefully to balance the system unavailability that occurs when no leader is present with the risk of dueling proposers. It's important to implement the right timeouts and backoff strategies. If multiple processes detect that there is no leader and all attempt to become leader at the same time, then none of the processes is likely to succeed (again, dueling proposers). Introducing randomness is the best approach. Raft, for example, has a well-thought-out method of approaching the leader election process."

### 23.4.2 Scaling Read-Heavy Workloads

### 23.4.3 Quorum Leases

### 23.4.4 Distributed Consensus Performance and Network Latency

"for systems with a very high number of clients, it may not be practical for all clients to keep a persistent connection to the consensus clusters open [...]. A solution is to use a pool of regional proxies [...] which hold persistent TCP/IP connections to the consensus group in order to avoid the setup overhead over long distances. Proxies may also be a good way to encapsulate sharding and load balancing strategies, as well as discovery of cluster members and leaders."

### 23.4.5 Reasoning About Performance: Fast Paxos

### 23.4.6 Stable Leaders

### 23.4.7 Batching

### 23.4.8 Disk Access

## 23.5 Deploying Distributed Consensus-Based Systems

### 23.5.1 Number of Replicas

"In general, consensus-based systems operate using *majority quorums*, i.e. a group of $2f + 1$ replicas may tolerate $f$ failures (if Byzantine fault tolerance, in which the system is resistant to replicas returning incorrect results, then $3f+1$ replicas may tolerate $f$ failures. For non-Byzantine failures, the minimum number of replicas that can be deployed is three–if two are deployed, then there is no tolerance for failure of any process. Three replicas may tolerate one failure. Most system downtime is a result of planned maintenance: three replicas allow a system to operate normally when one replica is down for maintenance (assuming that the remaining two replicas can handle system load at an acceptable performance)."

"If an unplanned failure occurs during a maintenance window, then the consensus system becomes unavailable. Unavailability of the consensus system is usually unacceptable, and so five replicas should be run, allowing the system to operate with up to two failures."

"The *replicated log* is not always a first-class citizen in distributed consensus theory, but it is a very important aspect of production systems. Raft describes a method for managing the consistency of replicated logs explicitly defining how any gaps in a replica's log are filled."

"There is a relationahips betwen performance and the number of replicas in a system that do not need to form part of a quorum: a minority of slower replicas may lag behind, allowing the quorum of better-performing replicas to run faster (as long as the leader performs well). If replica performance varies significantly, then every failure may reduce the performance of the system overall because slow outliers will be required to form a quorum. The more failures or lagging replicas a system can tolerate, the better the system's performance overall is likely to be."

### 23.5.2 Location of Replicas

"A *failure domain* is the set of components of a system that can become unavailabile as a result of a single failure. Example failure domains include the following:"

- "A physical machine"
- "A rack in a datacenter served by a single power supply"
- "Several racks in a datacenter that are served by one piece of networking equipment"
- "A datacenter that could be rendered unavailable by a fiber optic cable cut"
- "A set of dataceners in a single geographic area that could all be affected by a single natural disaster such as a hurricane"

"It doesn't always make sense to continually increase the size of the failure domain whose loss the system can withstand. For instance, if all the clients using a consensus system are running within a particular failure domain (say, the New York area) and deploying a distributed consensus-based system across a wider geographical area would allow it to remain serving during outages in that failure domain (say, Hurricane Sandy), is it worth it? Probably not"

"You should take disaster recovery into account when deciding where to locate your replicas: in a system that stores critical data, the consensus replicas are also essentially online copies of the system data. However, when critical data is at stake, it's important to back up regular snapshots elsewhere, even in the case of solid consensus-based systems that are deployed in several diverse failure domains. There are two failure domains that you can never escape: the software itself, and human error on the part of the system's administrators."

### 23.5.3 Capacity and Load Balancing

"If clients are dense in a particular geographic region, it is bsed to locate replicas close to clients."

"leader replicas will use more computational resources, particularly outgoing network capacity. This is because the leader sends proposal messages that include the proposed data, but replicas send smaller messages, usually just containing agreement with a particular consensus transaction ID. Organizations that run highly sharded consensus systems with a very large number of processes may find it necessary to ensure that leader processes for the different shards are balanced relatively evently across different datacenters. Doing so prevents the system as a whole from being bottlenecked on outgoing network capacity for just one datacenter, and makes for much greater overall system capacity."

"Another downside of deploying consensus groups in multiple datacenters is the very extreme change in the system that can occur if the datacenter hosting the leaders suffers a widespread failure (power, networking equipment failure, or fiber cut, for instance)."

"However, this type of deployment could easily be an unintended result of automatic processes in the system that have bearing on how leaders are chosen. For instance:"

- "Clients will experience better latency for any operations handled via the leader if the leader is located closest to them."
- "An algorithm might try to locate leaders on machines with the best performance. A pitfall of this approach is that if one of the three datacenters houess faster machines, then a disproportionate amount of traffic will be sent to that datacenter, resulting in extreme traffic changes should that datacenter go offline."
- "A leader election algorithm might favor processes that have been running longer. Longer-running processes are quite likely to be correlated with location if software releases are performed on a per-datacenter basis."

#### 23.5.3.1 Quorum composition

"Geography can complicate [the approach of spreading replicas to have similar RTTs to all replicas]. This is particularly true for intracontinental versus transpacific and transatlantic traffic. Consider a system that spans North America and Europe: it is impossible to locate replicas equidistant from each other because there will always be a longer lag for transatlantic traffic than for intracontinental traffic. No matter what, transactions from one region will need to make a transatlantic round trip in order to reach consensus."

"systems designers might choose to site five replicas, with two replicas roughly centrally in the US, one on the east cast, and two in Europe. Such a distribution would mean that in the average case, consensus could be achieved in North America without waiting for replies form Europe, or that from Europe, consensus can be achieved by exchanging messages only with the east coast replica. The east cost replica acts as a linchpin of sorts, where two possible quorums overlap. [...] loss of this replica means that system latency is likely to change drastically"

"This scenario is a key weakness of the simple majority quorum when applied to groups composed of replicas with very different RTTs between members. In such cases, a hierarchical quorum approach may be useful."

### 23.6 Monitoring Distributed Consensus Systems

"Experience has shown us that there are certain specific aspects of distributed consensus systems that warrant special attention. These are:"

- The number of members running in each consensus group, and the status of each process (healthy or not healthy)
- Persistently lagging replicas
- Whether or not a leader exists
- Number of leader changes
- Consensus transaction number
- Number of proposals seen; number of proposals agreed upon
- Throughput and latency

### 23.7 Conclusion

## 24 Distributed Periodic Scheduling with Cron

### 24.1 Cron

#### 24.1.1 Introduction

#### 24.1.2 Reliability Perspective

"Cron's failure domina is essentially just one machine. If the machine is not running, neither the cron scheduler nor

the jobs it launches can run."

"The only state that needs to persiste across crond restarts (including machine reboots) is the crontab configuration itself. The cron launches are fire-and-forget, and crond makes no attempt to track these launches."

"anacron attempts to launch jobs that would have been launched when the system was down. Relaunch attempts are limited to jobs that run daily or less frequently."

## 24.2 Cron Jobs and Idempotency

"some cron jobs are idempotent [and some should not be launched more than once]"

"failure to launch is acceptable for some cron jobs but not for others. For example, a garbage collection cron job scheduled to run every five minutes may be able to skip one launch, but a payroll cron job scheduled to run once a month should not be skipped."

"This large variety of cron jobs makes reasoning about failure modes difficult: in a system like the cron service, tehre is no single answer that fits every situation. In general, we favor skipping launches rather than risking double launches, as much as the infrastructure allows. This is because recovering from a skipped launch is more tenable than recovering from a double launch. Cron job owners can (and should!) monitor their cron jobs [...]. In case of a skipped launch, cron job owners can take action that appropriately matches the nature of the cron job. However, undoing a double launch, such as the previously mentioned newsletter example, may be difficult or even entirely impossible. Therefore, we prefer to "fail closed" to avoid systematically creating bad state."

## 24.3 Cron at Large Scale

### 24.3.1 Extended Infrastructure

"To increase cron's reliability, we decouple processes from machines. [...] Launching a job in a datacenter then effectively turns into sending one or more RPCs to the datacenter scheduler."

"Discovering a dead machine entails health check timeouts, while rescheduling your service onto a different machine requires time to install software and start up the new process."

"Because moving a process to a different machine can mean loss of any local state stored on the old machine (unless live migration is employed), and the rescheduling time may exceed the smallest scheduling interval of one minute, we need procedures in place to mitigate both data loss and excessive time requirements. To retain local state of teh old machine, you might simply persist the state on a distributed filesystem such as GFS, and use this filesystem during startup to identify jobs that failed to launch due to rescheduling. However, this solution falls short in terms of timeliness expectations: if you run a job every five minutes, a one- to two-minute delay cause by the total overhead of cron system rescheduling is potentially unacceptably substantial. In this case, hot spares, which would be able to jump quickly in and resume operation, can significantly shorten this time window."

### 24.3.2 Extended Requirements

"Deployment at datacenter scale commonly means deployment into containers that enforce isolation. Isolation is necessary because the base expectation is that independent processes running in the same datacenter should not negatively impact each other. In order to enforce that expectation, you should know the quantity of resources you need to acquire up front for any given process you want to run–both for the cron system and the jobs it launches. A cron job may be delayed if the datacenter does not have resources available to match the demands of the cron job. Resource requirements, in addition to user demand for monitoring of cron job launches, means that we need to track the full state of our cron job launches, from the scheduled launch to termination."

"Decoupling process launches from specific machines exposes the cron system to partial launch failure. The versatility of cron job configurations also means that launching a new cron job in a datacenter may need multiple RPCs, such

that sometimes we encounter a scenario in which some RPCs succeeded but others did not [...]. The cron recovery procedure must also account for this scenario."

## 24.4 Building Cron at Google

### 24.4.1 Tracking the State of Cron Jobs

To track the state of jobs, there are roughly two options: store data externally; or, use a system that stores a small volume of state as part of the cron service itself.

Google chose the second option for these reasons:

- "Distributed filesystems such as GFS or HDFS often cater to the use case of very large files, whereas the information we need to store about cron jobs is very small. Small writes on a distributed filesystem are very expensive and come with a high latency"
- "Base services for which outages have wide impact (such as cron) should have very few dependencies. [...] However, cron should be able to operate independently of downstream systems that cater to a large number of internal users."

### 24.4.2 The Use of Paxos

"We deploy multiple replicas of the cron service and use the Paxos distributed consensus algorithm to ensure they have consistent state."

"distributed cron uses a single leader job, which is the only replica that can modify the shared state, as well as the only replica that can launch cron jobs."

"As soon as the new leader is elected, we follow a leader election protocol specific to the cron service, which is responsible for taking over all the work left unfinished by the previous leader. The leader specific to the cron service is the same as the Paxos leader, but the cron service needs to take additional action upon promotion."

"The most important state we keep in Paxos is information regarding which cron jobs are launched. We synchronously inform a quorum of replicas of the beginning and end of each scheduled launch for each cron job."

### 24.4.3 The Roles of the Leader and the Follower

#### 24.4.3.1 The leader

"Upon reaching the scheduled launch time, the leader replica announces that it is about to start this particular cron job's launch, and calculates the new scheduled launch time, just like a regular crond implementation would."

"Simply identifying the cron job is not enough: we should also uniquely identify the particular launch using the start time; otherwise, ambiguity in cron job launch tracking may occur. (Such ambiguity is especially unlikely in the case of high-frequency cron jobs, such as those running every minute.)"

"It is important that Paxos communication remain synchronous, and that the actual cron job launch does not proceed until it receives confirmation that the Paxos quorum has received the launch notification. The cron service needs to understand whether each cron job has launched in order to decide the next scourse of action in case of leader failover."

#### 24.4.3.2 The follower

#### 24.4.3.3 Resolving partial failures

Every cron job launch has two synchronization points: when we are about to perform the launch; and when we have finished the launch

"Even if the launch consists of a single RPC, how do we know if the RPC was actually sent?"

One of these conditions must be met:

- "All operations on external systems, which we may need to continue upon re-election, must be idempotent"
- "We must be able to look up the state of all operations on external systems in order to unambiguously determine whether they completed or not."

"Most infrastructure that launches logical jobs in datacenters (Mesos, for example) provides naming for those datacenter jobs, making it possible to look up the state of jobs, stop the jobs, or perform other maintenance. A reasonable solution to the idempotency problem is to construct job names ahead of time (thereby avoiding causing any mutating operations on the datacenter scheduler), and then distribute the names to all replicas of your cron service. If the cron service leader dies during launch, the new leader simply looks up the state of all the precomputed names and launches the missing names."

"Recall that we track the scheduled launch time when keeping the internal state between the replicas. Similarly, we need to disambiguate our interaction with the datacenter scheduler, also by using the scheduled launch time."

### 24.4.4 Storing the State

"Paxos is essentially a continuous log of state changes, appended to synchronously as state changes occur."

"In case of lost logs, we only lose the state since the last snapshot. Snapshots are in fact our most critical state–if we lose our snapshots, we essentially have to start from zero again because we've lost our internal state. Losing logs, on the other hand, just causes a bounded loss of state and sends the cron system back in time to the point when the latest snapshot was taken."

"We store Paxos logs on local disk of the machine where cron service replicas are scheduled. Having three replicas in default operation implies that we have three copies of the logs. We store the snapshots on local disk as well. However, because they are critical, we also back them up onto a distributed filesystem, thus protecting against failures affecting all three machines."

"We consciously decided that losing logs, which represent a small amount of the most recent state changes, is an acceptable risk. Storing logs on a distributed filesystem can entail a substantial performance penalty caued by frequent small writes. The simultaneous loss of all three machines is unlikely, and if simultaneous loss does occur, we automatically restore from the snapshot. We thereby lose only a small amount of logs: those taken since the last snapshot, which we perform on configurable intervals."

"In addition to the logs and snapshots stored on local disk and snapshot backups on the distributed filesystem, a freshly started replica can fetch the state snapshot and all logs from an already running replica over the network. This ability makes replica startup independent of any state on the local machine."

## 24.5 Running Large Cron

"Beware the large and well-known problem of distributed systems: the thundering herd. Based on user configuration, the cron service can cause substantial spikes in datacenter usage. When people think of a "daily cron job," they commonly configure this job to run at midnight."

To solve this problem, google expanded the launch schedule specification for their cron to include the "?" metacharacter (whereas "*" indicates the cron should run at every value, "?" indicates that it can run at any value and the cron system can choose).

### 24.6 Summary

# 25 Data Processing Pipelines

## 25.1 Origin of the Pipeline Design Pattern

"The classic approach to data processing is to write a program that reads in data, transforms it in some desired way, and outputs new data. Typically the program is scheduled to run under the control of a periodic scheduling program such as cron. This design pattern is called a *data pipeline*."

## 25.2 Initial Effect of Big Data on the Simple Pipeline Pattern

"Programs are typically organized into a chained series, with the output of one program becoming the input to the next. [...] Programs organized this way are called *multiphase pipelines*"

"The number of programs chained together in a series is a measurement known as the *depth* of a pipeline."

## 25.3 Challenges with the Periodic Pipeline Pattern

"when a periodic pipeline is first installed with worker sizing, periodicity, chunking technique, and other parameters carefully tuned, performance is initially reliable. However, organic growth and change inevitably begin to stress the system, and problems arise."

## 25.4 Trouble Caused By Uneven Work Distribution

"The key breakthrough of Big Data is the widespread application of "embarrassingly parallel" algorithms to cut a large workload into chunks small enough to fit onto individual machines. Sometimes chunks require an uneven amount of resources relative to one another, and it is seldom initially obvious why particular chunks require different amounts of resources."

## 25.5 Drawbacks of Periodic Pipelines in Distributed Environments

"batch work is not sensitive to latency in the same way that Internet-facing web services are."

"In light of the risk trade-offs, running a well-tuned periodic pipeline successfully is a delicate balance between high resource cost and risk of preemptions."

"Delays of up to a few hours might well be acceptable for pipelines that run daily. However, as the scheduled exectuion frequency increases, the minimum time between executions can quickly reach the minimum average delay point, placing a lower bound on the latency that a periodic pipeline can expect to attain. Reducing the job execution interval below this effective lower bound simply results in undesirable behavior rather than increased progress."

"a distinction between batch scheduling resources versus production priority resources has to be made to rationalize resource acquisition costs."

### 25.5.1 Monitoring Problems in Periodic Pipelines

"For pipelines of sufficient execution duration, having real-time information on runtime performance metrics can be as important, if not even more important, than knowing overall metrics. This is because real-time data is important to providing operational support, including emergency response. In practice, teh standard monitoring model involves collecting metrics during job execution, and reporting metrics only upon completion."

"Continuous pipelines do not share these problems because their tasks are constantly running and their telemetry is routinely designed so that real-time metrics are available."

### 25.5.2 "Thundering Herd" Problems

"Given a large enough periodic pipeline, for each cycle, potentially thousands of workers immediately start work. If there are too many workers or if the workers are misconfigured or invoked by faulty retry logic, the servers on which they run will be overwhelmed, as will the underlying shared cluster services, and any networking infrastructure that was being used will also be overwhelmed."

### 25.5.3 Moir Load Pattern

"A related problem we call "Moir load pattern" occurs when two or more pipelines run simultaneously and their execution sequences occasionally overlap, causing them to simultaneously consume a common shared resource."

## 25.6 Introduction to Google Workflow

"it's important to ascertain several details at the outset of designing a system involving a proposed data pipeline. Be sure to scope expected growth trajectory (Jeff Dean's lecture on "Software Engineering Advice from Building Large-Scale Distributed Systems" is an excellent resource), demand for design modifications, expected additional resources, and expected latency requirements from the business."

"Workflow uses the leader-follower (workers) distributed systems design pattern and the system prevalence design pattern."

### 25.6.1 Workflow as Model-View-Controller Pattern

## 25.7 Stages of Execution in Workflow

### 25.7.1 Workflow Correctness Guarantees

## 25.8 Ensuring Business Continuity

## 25.9 Summary and Concluding Remarks

# 26 Data Integrity: What You Read Is What You Wrote

## 26.1 Data Integrity's Strict Requirements

"An SLO of 99.99% uptime leaves room for only an hour of downtime in a whole year. This SLO sets a rather high bar, which likely exceeds the expectations of most Internet and Enterprise users."

"In contrast, an SLO of 99.99% good bytes in a 2 GB artifact would render documents, executables, and databases corrupt (up to 200 KB garbled)."

"*the secret to superior data integrity is proactive detection and rapid repair and recovery*

### 26.1.1 Choosing a Strategy for Superior Data Integrity

### 26.1.2 Backups Versus Archives

'No one really *wants* to make backups; what people *really* want are *restores*.'

'The most important difference between backups and archives is that backups *can* be loaded back into an application, while archives *cannot*.'

'*Archives* safekeep data for long periods of time to meet auditing, discovery, and compliance needs. Data recovery for such purposes generally doesn't need to complete within uptime requirements of a service.'

'when disaster strikes, data must be recovered from *real backups* quickly, preferably well within the uptime needs of a service.'

'when formulating a backup strategy, consider how quickly you need to be able to recover from a problem, and how much recent data you can afford to lose.'

### 26.1.3 Requirements of the Cloud Environment in Perspective

## 26.2 Google SRE Objectives in Maintaining Data Integrity and Availability

### 26.2.1 Data Integrity Is the Means; Data Availability Is the Goal

'From the user's point of view, data integrity without expected and regular data availability is effectively the same as having no data at all.'

### 26.2.2 Delivering a Recovery System, Rather Than a Backup System

'Backups aren't a high priority for anyone–they're an ongoing drain on time and resources, and yield no immediate visible benefit. [...] The fundamental problem with [a] lackadaisical strategy is that the dangers it entails may be low risk, but they are also high impact.'

'Instead of focusing on the thankless job of taking a backup, it's much more useful, not to mention easier, to motivate participation in taking backups by concentrating on a task with a visible payoff: the *restore*! *Backups are a tax*, one paid on an ongoing basis for the municipal service of guaranteed data availability. Instead of emphasizing the tas, draw attention to the service the tax funds: data availability. We don't make teams "practice" their backups, instead:'

- 'Teams define service level objectives (SLOs) for data availability in a variety of failure modes.'
- 'A team practices and demonstrates their ability to meet those SLOs'

### 26.2.3 Types of Failures That Lead to Data Loss

'A study of 19 data recovery efforts at Google found that the most common user-visible data loss scenarios involved data deletion or loss of referential integrity caused by software bugs. [...] Therefore, the safeguards Google employs should be well suited to prevent or recover from these types of loss.'

'The application may also need to recover each affected artifact ot a unique point in time. This data recovery scenario is called "point-in-time recovery" outside Google, and "time travel" inside Google.'

'Many projects compromise by adopting a tiered backup strategy without point-in-time recovery. For instance, the APIs beneath your application may support a variety of data recovery mechanisms. Expensive local "snapshots" may provide limited protection from application bugs and offere quick restoration functionality, so you might retain a few days of such local "snapshots," taken several hours apart.'

#### 26.2.4 Challengages of Maintaining Data Integrity Deep and Wide

'*replication and redundancy are not recoverability*'

##### 26.2.4.1 Scaling issues: Fulls, incrementals, and the competing forces of backups and restores

*Replication provides many benefits, including locality of data and protection from a site-specific disaster, but it can't protect you from many sources of data loss. Datastores that automatically sync multiple replicas guarantee that a corrupt database row or errant delete are pushed to all of your copies, likely before you can isolate the problem.*

'To address this concern, you might make nonserving copies of your data in some other fomrat, such as frequent database exports to a native file. This additional measure adds protection from the types of errors replication doesn't protect against–user errors and application-layer bugs–but does nothing to guard against losses introduced at a lower layer. This measure also introduces a risk of bugs during data conversion (in both directions) and during storage of the native file, in addition to possible mismatches in semantics between the two formats. Imagine a zero-day attack at some low level of your stack, such as the filesystem or device driver. Any copies that rely on the compromised software component, including the database exports that were written to the same filesystem that backs your database, are vulnerable.'

'diversity is key: protecting against a failure at layer X requires storing data on diverse components at that layer. Media isolation protects against media flaws: a bug or attack in a disk device driver is unlkely to affect tape drives. If we could, we'd make backup copies of our valuable data on clay tablets.'

'The forces of data freshness and restore completion compete against comprehensive protection. The further down the stack you push a snapshot of your data, the longer it takes to make a copy, which means that the frequency of copies decreases.'

##### 26.2.4.2 Retention

'Retention–how long you keep copies of your data around–is yet another factor to consider in your data recovery plans.'

'it might takes days for a more gradual loss of data to attract the right person's attention. Restoring the lost data in [this] scenario requires snapshots taken further back in time. When reaching back this far, you'll likely want to merge the restored data with the current state.'

### 26.3 How Google SRE Faces the Challenges of Data Integrity

#### 26.3.1 The 24 Combinations of Data Integrity Failure Modes

'there is no silver bullet that guards against the many combinations of failure modes. [...] Defense in depth comprises multiple layers, with each successive layer of defense conferring protection from progressively less common data loss scenarios. [The figure] illustrates an object's journey from soft deletion to destruction, and the data recovery strategies that should be employed along this journey to ensure defense in depth.'

### 26.3.2 First Layer: Soft Deletion

'When velocity is high and privacy matters, bugs in applications account for the vast majority of data loss and corruption events.'

'Any product that upholds the privacy of its users must allow the users to delete selected subsets and/or all of their data. Such products incur a support burden due to accidental deletion. Giving users the ability to undelete their data (for example, via a trash folder) reduces but cannot completely eliminate this support burden.'

'Soft deletion can dramatically reduce or even completely eliminate this support burden. Soft deletion means that deleted data is immediately marked as such, rendering it unusable by all but the application's administrative code paths. Administrative code paths may include legal discovery, hijacked account recovery, enterprise administration, user support, and problem troubleshooting and its related features. Conduct soft deletion when a user empties his or her trash, and provide a user support tool that enables authorized administrators to undelete any items accidentally deleted by users.'

'Another common source of unwanted data deletion occurs as a result of account hijacking.'

'In Google's experience, the majority of account hijacking and data integrity issues are reported or detected within 60 days. Therefore, the case for soft deleting data for longer than 60 days may not be strong.'

'Google has also found that the most devastating acute data deletion cases are caused by application developers unfamiliar with existing code but working on deletion-related code, especially batch processing pipelines [...]. It's advantageous to design your interfaces to hinder developers unfamiliar with your code from circumventing soft deletion features with new code.'

'it can be useful to introduce an additional layer of soft deletion, which we will refer to as "lazy deletion." You can think of lazy deletion as behind the scenes purging, controlled by the storage system (whereas soft deletion is controlled by and expressed to the client application or service). In a lazy deletion scenario, data that is deleted by a cloud application becomes immediately inaccessible to the application, but is preserved by the cloud service provider for up to a few weeks before destruction. Lazy deletion isn't advisable in all defense in depth strategies: a long lazy deletion period is costly in systems with much short-lived data, and impractical in systems that must guarantee destruction of deleted data within a reasonable time frame (i.e., those that offer privacy guarantees).'

'What about *revision history*? Some products provide the ability to revert items to previous states. When such a feature is available to users, it is a form of trash.'

### 26.3.3 Second Layer: Backups and Their Related Recovery Methods

'Backups and data recovery are the second line of defense after soft deletion. The most important principle in this layer is that backups don't matter; what matters is recovery. The factors supporting successful recovery should drive your backup decisions, not the other way around.'

'How much recent data can you afford to lose during a recovery effort? The less data you can afford to lose, the more serious you should be about an incremental backup strategy.'

'Even if money isn't a limitation, frequent full backups are expenive in other ways. Most notably, they impose a compute burden on the live datastores of your service while it's serving users, driving your service closer to its scalability and performance limits. To ease this burden, you can take full backups during off-peak hours, and then a series of incremental backups when your service is busier.'

'How quickly do you need to recover? The faster your users need to be rescued, the more local your backups should be.'

'How far back should your backups reach? Your backup strategy becomes more costly the further back you reach.'

'In Google's experience, low-grade data mutation or deletion bugs within application code demand the furthest reaches back in time, as some of those bugs were noticed months after the first data loss began. Such cases suggest that you'd like the ability to reach back in time as far as possible.'

'On the flip side, in a high-velocity development environment, changes to code and schema may render older backups expensive or impossible to use. Furthermore, it is challenging to recover different subsets of data to different restore points,b ecause doing so would involve multiple backups. Yet, that is exactly the sort of recovery effort demanded by low-grade data corruption or deletion scenarios.'

#### 26.3.4 Overarching Layer: Replication

'As the volume of data increases, replication of every storage instance isn't always feasible. In such cases, it makes sense to stagger successive backups across different sites, each of which may fail independently, and to write your backups using a redundancy method such as RAID, Reed-Solomon erasure codes, or GFS-style replication.'

#### 26.3.5 1T vs 1E: Not "Just" a Bigger Backup

'If we shard our data well, it's possible to run $N$ tasks in parallel [...]. Doing so requires some forethought and planning in the schema design and the physical deployment of our data in order to: Balance the data correctly; Ensure the independence of each shard; Avoid contention among the concurrent sibling tasks'

#### 26.3.6 Third Layer: Early Detection

##### 26.3.6.1 Challenges faced by cloud developers

##### 26.3.6.2 Out-of-band data validation

'Out-of-band validators can be expenive at scale. A significant portion of Gmail's compute resource footprint supports a collection of daily validators. To compound this expense, these validators also lower server-side cache hit rates, reducing server-side responsiveness experienced by users. To mitigate this hit to responsiveness, Gmail provides a variet of knobs for rate-limiting its validators and periodically refactors the validators to reduce disk contention.'

Effective out-of-band data validation demands all of the following:

- Validation job management
- Monitoring, alerting, and dashboards
- Rate-limiting features
- Troubleshooting tools
- Production playbooks
- Data validation APIs that make validators easy to add and refactor

### 26.4 Knowing that Data Recovery Will Work

The aspects of your recovery plan you should confirm are myriad:

- Are your backups valid and complete, or are they empty?
- Do you have sufficient machine resources to run all of the setup, restore, and post-processing tasks that comprise your recovery?
- Does the recovery process complete in a reasonable wall time?
- Are you able to monitor the state of your recovery process as it progresses?
- Are you free of critical dependencies outside of your control, such as access to an offsite media storage vault that isn't available 24/7?

## 26.5 Case Studies

### 26.5.1 Gmail–February, 2011: Restore from GTape

#### 26.5.1.1 Sunday, February 27, 2011, late in the evening

### 26.5.2 Google Music–March 2012: Runaway Deletion Detection

#### 26.5.2.1 Tuesday, March 6th, 2012, mid-afternoon

#### 26.5.2.2 Discovering the problem

#### 26.5.2.3 Assessing the damage

#### 26.5.2.4 Resolving the issue

#### 26.5.2.5 Addressing the root cause

'cloud computing engineers are often reluctant to set up production alerts on data deletion rates due to natural variation of per-user data deletion rates with time. However, since the intent of such an alert is to detect global rather than local deletion rate anomalies, it would be more useful to alert when the global data deletion rate, aggregated across all users, crosses an extreme threshold (such as 10x the observed 95th percentile), as opposed to less useful per-user deletion rate alerts.'

## 26.6 General Principles of SRE as Applied to Data Integrity

### 26.6.1 Beginner's Mind

'Large-scale, complex services have inherent bugs that can't be fully grokked. Never think you understand enough of a complex system to stay it won't fail in a certain way. Trust but verify, and apply defense in depth.'

### 26.6.2 Trust but Verify

'It's a given that regardless of your engineering quality or rigor of testing, the API will have defects. Check the correctness of the most critical elements of your data using out-of-band data validators, even if API semantics suggest that you need not do so. Perfect algorithms may not have perfect implementations.'

### 26.6.3 Hope Is Not a Strategy

'System components that aren't continually exercised fail when you need them most. Prove that data recovery works with regular exercise, or data recovery won't work. Humans lack discipline to continually exercise system components, so automation is your friend.'

### 26.6.4 Defense in Depth

'The best data integrity strategies are multitiered–multiple strategies that fall back to one another and address a broad swath of scenarios together at reasonable cost.'

## 26.7 Conclusion

'Rather than focusing on the means to the end, Google SRE finds it useful to borrow a page from test-driven development by proving that our systems can maintain data availability with a predicted maximum down time. The means and mechanismas that we use to achieve this end goal are necessary evils. By keeping our eyes on the goal, we avoid falling into the trap in which "The operation was a success, but the system died."'

'As you get better recovering from any breakage in reasonable time $N$, find ways to whittle down that time through more rapid and finder-grained loss detection, with the goal of approaching $N = 0$. You can then switch from planning recovery to planning prevention, with the aim of achieving the holy grail of *all the data, all the time.*'

# 27 Reliable Product Launches at Scale

## 27.1 Launch Coordination Engineering

Consulting Launch Coordination Engineering (LCE) teams faciliate smooth product launches by:

- Auditing products and services for compliance with Google's reliability standards and best practices, and providing specific actions to improve reliability.
- Acting as a liaison between the multiple teams involved in a launch
- Driving the technical aspects of a launch by making sure that tasks maintain momentum
- Acting as gatekeepers and signing off on launches determined to be "safe"
- Educating developers on best practices and on how to integrate with Google's services, equipping them with internal documentation and training resources to speed up their learning

### 27.1.1 The Role of the Launch Coordination Engineer

'LCEs are held to the same technical requirements as any other SRE, and are also expected to have strong communication and leadership skills–an LCE brings disparate parties together to work toward a common goal, mediates occasional conflicts, and guides, coaches, and educates fellow engineers.'

## 27.2 Setting Up a Launch Process

### 27.2.1 The Launch Checklist

### 27.2.2 Driving Convergence and Simplification

'In a large organization, engineers may not be aware of available infrastructure for common tasks (such as rate limiting). Lacking proper guidance, they're likely to reimplement existing solutions. Converging on a set of common infrastructure libraries avoids this scenario, and provides obvious benefits to the company: it cuts down on duplicate effort, makes knowledge more easily transferable between serices, and results in a higher level of engineering and service quality due to the concentrated attention given to infrastructure.'

'Almost all groups at Google participate in a common launch process, which makes the launch checklist a vehicle for driving convergence on common infrastructure. Rather than implementing a custom solution, LCE can recommend existing infrastructure as building blocks–infrastructure that is already hardened through years of experience and that can help mitigate capacity, performance, or scalability risks.'

'LCEs are also in a unique position to identify opportunities for simplification. While working on a launch, they witness the stumbling blocks firsthand: which parts of a launch are causing the most struggle, which steps take a disproportionate amount of time, which problems get solved independently over and over again in similar ways, where common infrastructure is lacking, or where duplication exists in common infrastructure.'

#### 27.2.3 Launching the Unexpected

### 27.3 Developing a Launch Checklist

#### 27.3.1 Architecture and Dependencies

'An architecture review allows you to determine if the service is using shared infrastructure correctly and identifies the owners of shared infrastructure as additional stakeholders in the launch.'

Example checklist questions

- What is your request flow from user to frontend to backend?
- Are there different types of requests with different latency requirements?

#### 27.3.2 Integration

'Many companies' services run in an internal ecosystem that entails guidelines on how to set up machines, configure new services, set up monitoring, integrate with load balancing, set up DNS addresses, and so forth. These internal systems usually grow over time, and often have their own idiosyncracies and pitfalls to navigate.'

Example action items

- Set up a new DNS name for your service
- Set up load balancers to talk to your service
- Set up monitoring for your new service

#### 27.3.3 Capacity Planning

'The type of workload or traffic mix from a launch spike could be substantially different from steady state, throwing off load test results.'

Example checklist questions

- Is this launch tied to a press release, advertisement, blog post, or other form of promotion?
- How much traffic and rate of growth do you expect during and after the launch?
- Have you obtained all the compute resources needed to support your traffic?

### 27.3.4 Failure Modes

'In this portion of the checklist, examine each component and dependency and identify the impact of its failure. Can the service deal with individual machine failures? Datacenter outages? Network failures? How do we deal with bad input data? Are we prepared for the possibility of a denial-of-service (DoS) attack? Can the service continue serving in a degraded mode if one of its dependencies fails? How do we deal with unavailability of a dependency upon startup of the service? During runtime?'

Example checklist questions

- Do you have any single points of failure in your design?
- How do you mitigate unavailability of your dependencies?

### 27.3.5 Client Behavior

'abusive client behavior can very easily threaten the stability of a service. (There is also teh topic of protecting a service from abusive traffic such as scrapers and denial-of-service attacks–which is different from designing safe behavior for first-party clients.)'

### 27.3.6 Processes and Automation

### 27.3.7 Development Process

### 27.3.8 External Dependencies

Example checklist questions

- What third-party code, data, services, or events does the service or the launch depend upon?
- Do any partners depend on your service? If so, do they need to be notified of your launch?
- What happens if you or the vendor can't meet a hard launch deadline?

### 27.3.9 Rollout Planning

## 27.4 Selected Techniques for Reliable Launches

### 27.4.1 Gradual and Staged Rollouts

'Almost all updates to Google's services proceed gradually, according to a defined process, with appropriate verification steps interspersed.[...] The first stages of a rollout are usually called "canaries" [...]. Our canary servers detect dangerous effects from teh behavior of the new software under real user traffic.'

'The concept of gradual rollouts even applies to software that does not run on Google's servers. New versions of an Android app can be rolled out in a gradual manner, in which the updated version is offered to a subset of the installs for upgrade.'

### 27.4.2 Feature Flag Frameworks

Such frameworks usually meet the following requirements:

- Roll out many changes in parallel, each to a few servers, users, entities, or datacenters.
- Gradually increase to a larger but limited group of users, usually between 1 and 10 percent
- Direct traffic through different servers depending on users, sessions, objects, and/or locations
- Automatically handle failure of the new code paths by design, without affecting users
- Independently revert each such change immediately in the event of serious bugs or side effects
- Measure the extent to which each change improves the user experience

'A configuration mechanism may specify an identifier associated withe the new code paths and the scope of the change (e.g., cooki hash mode range), whitelists, and blacklists.'

'Stateful services tend to limit feature flags to a subset of unique logged-in user identifiers or to the actual product entities accessed, such as the ID of documents, spreadsheets, or storage objects. Rather than rewrite HTTP payloads, stateful services are more likely to proxy or reroute requests to different servers depending on the change, conferring the ability to test improved business logic and more complex new features.'

### 27.4.3 Dealing with Abusive Client Behavior

### 27.4.4 Overload Behavior and Load Tests

'Many services are much slower when they are not loaded, usually due to the effect of various kinds of caches such as CPU caches, JIT caches, and service-specific data caches.'

'At some point, many services reach a point of nonlinearity as they approach overload.'

'it is very hard to predict from first principles how a service will react to overload. Therefore, load tests are an invaluable tool, both for reliability reasons and capacity planning, and load testing is required for most launches.'

## 27.5 Development of LCE

'the SRE organization was growing quickly, and inexperienced SREs were sometimes overly cautious and averse to change. Google ran a risk that the resulting negotiations between these two parties would reduce the velocity of product/feature launches.'

'[The first team of LCEs, created in 2004] were responsible for accelerating the launches of new products and features, while at the same time applying SRE expertise to ensure that Google shipped reliable products with high availability and low latency.'

'LCEs were responsible for making sure launches were executed quickly without the services falling over, and that if a launch did fail, it didn't take down other products. LCEs were also responsible for keeping stakeholders informed of the nature and likelihood of such failures whenever corners were cut in order to accelerate time to market. Their consulting sessions were formalized as Production Reviews.'

#### 27.5.1 Evolution of the LCE Checklist

#### 27.5.2 Problems LCE Didn't Solve

##### 27.5.2.1 Scalability changes

##### 27.5.2.2 Growing operational load

'When running a service after it launches, operational load, the amount of manual and repetitive engineering needed to keep a system functioning, tends to grow over time unless efforts are made to control such load. [...] SRE has an internally advertised goal of keeping oeprational work below a maximum of 50%. Staying below this maximum requires constant tracking of sources of operational work, as well as directed effort to remove these sources.'

##### 27.5.2.3 Infrastructure churn

### 27.6 Conclusion

## 28 Accelerating SREs to On-Call and Beyond

### 28.1 You've Hired Your Next SRE(s), Now What?

'It is essential, then, but not sufficient, to think of SRE education through the lens of, "What does a newbie need to learn to go on-call?"'

| Recommended patterns | Anti-pa |
|---|---|
| Designing concrete, sequential learning experiences for students to follow | Delugin |
| Encouraging reverse engineering, statistical thinking, and working from fundamental principles | Trainin |
| Celebrating the analysis of failure by suggesting postmortems for students to read | Treatin |
| Creating contained but realistic breakages for students to fix using real monitoring and tooling | Having |
| Role-playing theoretical disasters as a group, to intermingle a team's problem-solving approaches | Creatin |
| Enabling students to shadow their on-call rotation early, comparing notes with the on-caller | Pushin |
| Pairing students with expert SREs to revise targeted sections of the on-call training plan | Treatin |
| Carving out nontrivial project work for students to undertake, allowing them to gain partial ownership in the stack | Awardi |

### 28.2 Initial Learning Experiences: The Case for Structure Over Chaos

#### 28.2.1 Learning Paths That Are Cumulative and Orderly

'The Google Search SRE team structures this learning through a document called the "on-call learning checklist."'

'Depending on how the access permissions are configured for your service, you can also consider implementing a tiered access mode. The first tier of access would allow your student read-only access to the inner workings of the components, and a later tier would permit them to mutate the production state. [...] The Search SRE team calls these attained levels "powerups" on the route to on-call, as trainees are eventually added into the highest level of systems access.'

#### 28.2.2 Targeted Project Work, Not Menial Work

## 28.3 Creating Stellar Reverse Engineers and Improvisational Thinkers

#### 28.3.1 Reverse Engineers: Figuring Out How Things Work

'Teach your SREs about the diagnostic and debugging surfaces of your applications and have them practice drawing inferences from the information these surfaces reveal, so that such behavior becomes reflexive when dealing with future outages.'

#### 28.3.2 Statistical and Comparative Thinkers: Stewards of the Scientific Method Under Pressure

'tracking down system breakages is often akin to playing a game of "which one of these things is not like the other?" where "things" might entail kernel version, CPU architecture, binary version(s) in your stack, regional traffic mix, or a hundred other factors. Architecturally, it's the team's responsibility to ensure all of these factors can be controlled for and individually analyzed and compared.'

#### 28.3.3 Improv Artists: When the Unexpected Happens

#### 28.3.4 Tying This Together: Reverse Engineering a Production Service

## 28.4 Five Practices for Aspiring On-Callers

#### 28.4.1 A Hunger for Failure: Reading and Sharing Postmortems

'Without radical editing, subtle changes can be made to our best postmortems to make them "teachable" postmortems.'

'Even the best postmortems aren't helpful if they languish in the bottom of a virtual filing cabinet. [...] your team should collect and curate valuable postmortems to serve as educational resources for future newbies.'

#### 28.4.2 Disaster Role Playing

"Wheel of Misfortune" / "Walk the Plank"

#### 28.4.3 Break Real Things, Fix Real Things

#### 28.4.4 Documentation as Apprenticeship

#### 28.4.5 Shadow On-Call Early and Often

'After the student has made their way through all system fundamentals (by completing, for example, an on-call learning checklist), consider configuring your alerting system to copy incoming pages to your newbie, at first only during business hours.'

'Some teams will also include a final step: having the experienced on-caller "reverse shadow" the student. The newbie will become primary on-call and own all incoming escalations, but the experienced on-caller will lurk in the shadows, independently diagnosing the situation without modifying any state. The experienced SRE will be available to provide active support, help, validation, and hints as necessary.'

### 28.5 On-Call and Beyond: Rites of Passage, and Practicing Continuing Education

### 28.6 Closing Thoughts

# 29 Dealing with Interrupts

## 29.1 Managing Operational Load

'The primary on-call engineer might also manage support communications, escalation to product developers, and so on. In order to both minimize the interruption a page causes to a team and avoid the bystander effect, Google on-call shifts are managed by a single engineer.'

Pages, other interrupts (person-originated?), Tickets, Ongoing operational responsibilities

## 29.2 Factors in Determining How Interrupts are Handled

Google SRE teams have found the following metrics useful to decide how to handle interrupts:

- Interrupt SLO or expected response time
- The number of interrupts that are usually backlogged
- The severity of the interrupts
- The frequency of the interrupts
- The number of people available to handle a certain kind of interrupt (e.g., some teams require a certain amount of ticket work before going on-call)

## 29.3 Imperfect Machines

### 29.3.1 Cognitive Flow State

#### 29.3.1.1 Cognitive flow state: Creative and engaged

#### 29.3.1.2 Cognitive flow state: Angry Birds

### 29.3.2 Do One Thing Well

#### 29.3.2.1 Distractibility

#### 29.3.2.2 Polarizing time

'viewing an engineer as an interruptible unit of work, whose context switches are free, is suboptimal if you want people to be happy and productive. Assign a cost to context switches.'

### 29.3.3 Seriously, Tell Me What to Do

#### 29.3.3.1 General suggestions

#### 29.3.3.2 On-call

'The primary on-call engineer should focus solely on on-call work. If the pager is quiet for your service, tickets or other interrupt-based work that can be abandoned fairly quickly should be part of on-call duties. When an engineer is on-call for a week, that week should be written off as far as project work is concerned. If a project is too important to let slip by a week, that person shouldn't be on-call. Escalate in order to assign someone else to the on-call shift. *A person should never be expected to be on-call and also make progress on projects (or anything else with a high context switching cost).*'

'*You never run out of cleanup work.* Your ticket count might be at zero, but there is always documentation that needs updating, configs that need cleanup, etc.'."

#### 29.3.3.3 Tickets

"If you currently assign tickets randomly to victims on your team, *stop.*"

"Tickets should be a full-time role, for an amount of time that's manageable for a person. If you happen to be in the unenviable position of having more tickets than can be closed by the primary and secondary on-call engineers combined, then structure your ticket rotation to have two people handling tickets at any given time."

#### 29.3.3.4 Ongoing responsibilities

"define roles that let anyone on the team take up the mantle"

#### 29.3.3.5 Be on interrupts, or don't be

### 29.3.4 Reducing Interrupts

#### 29.3.4.1 Actually analyze tickets

"Even some basic introspection into the root causes of interrupts can provide good solutions for reducing the overall rate."

"Your team should conduct a regular scrub for tickets and pages, in which you examine classes of interrupts to see if you can identify a root cause. If you think the root cause is fixable in a reasonable amount of time, then *silence the interrupts until the root cause is expected to be fixed.* Doing so provides relief for the person handling interrupts and creatse a handy deadline enforcement for the person fixing the root cause."

#### 29.3.4.2 Rexpect yourself, as well as your customers

"If tickets are particularly annoying or onerous to resolve, you can effectively use policy to mitigate the burden."

Remember:

- Your team sets the level of service provided by your service.
- It's OK to push back some of the effort onto your customers.

"A policy fix can be temporary or permanent, depending on what makes sense. Such a fix should strike a good balance between respect for the customer and respect for yourself."

"If there are particular steps for an interrupt that are time-consuming or tricky, but don't require your privileges to accomplish, consider using policy to push the request back to the requestor. [...] Remember that if the customer wants a certain task to be accomplished, they should be prepared to spend some effort getting what they want."

"Your guiding principle in constructing a strategy for dealing with customer requests is that the request should be meaningful, be rational, and provide all the information and legwork you need in order to fulfill the request. In return, your response should be helpful and timely."

# 30 Embedding an SRE to Recover from Operational Overload

## 30.1 Phase 1: Learn the Service and Get Context

"more tickets should not require more SREs: the goal of the SRE model is to only introduce more humans as more complexity is added to the system. Instead, try to draw attention to how healthy work habits reduce the time spent on tickets."

Ops Mode Versus Nonlinear Scaling: "The term *ops mode* refers to a certain method of keeping a service running. Various work items increase with the size of the service. For example, a service needs a way to increase the number of configured virtual machines (VMs) as it grows. A team in ops mode responds by having a greater number of administrators managing those VMs. SRE instead focuses on writing software or eliminating scalability concerns so that the number of people required to run a service doesn't increase as a function of load on the service."

"Remember that your job is to make the service work, not to shield the development team from alerts."

### 30.1.1 Identify the Largest Sources of Stress

"SRE teams sometimes fall into ops mode because they focus on how to quickly address emergencies instead of how to reduce the number of emergencies."

### 30.1.2 Identify Kindling

"To give acceptable on-call support for a component, you should at least know the consequences when it breaks and the urgency needed to fix problems."

## 30.2 Phase 2: Sharing Context

### 30.2.1 Write a Good Postmortem for the Team

### 30.2.2 Sort Fires According to Type

"Some fires shouldn't exist. They cause what is commonly called ops work or toil."

## 30.3 Phase 3: Driving Change

### 30.3.1 Start with the Basics

"An SLO is probably the single most important lever for moving a team from reactive ops work to a healthy, long-term SRE focus. If this agreement is missing, no other advice in this chapter will be helpful. *If you find yourself on a team without SLOs, first read Chapter 4, then get the tech leads and management in a room and start arbitrating.*"

### 30.3.2 Get Help Clearing Kindling

### 30.3.3 Explain Your Reasoning

"*After you leave, the team should be able to predict what your comment on a design or changelist would be.*"

### 30.3.4 Ask Leading Questions

"Leading questions are not loaded questions. When talking with the SRE team, try to ask questions in a way that encourages people to think about the basic principles. It's particularly valuable for *you* to model this behavior because, by definition, a team in ops mode rejects this sort of reasoning from its own constituents. Once you've spent some time explaining your reasoning for various policy questions, this practice reinforces the team's understanding of SRE philosophy."

## 30.4 Conclusion

# 31 Communication and Collaboration in SRE

"there is a tremendous diversity in what SRE does, and how we do it. We have infrastructural teams, service teams, and horizontal product teams."

## 31.1 Communications: Production Meetings

"Production meetings are a special kind of meeting where an SRE team carefully articulates to itself–and to its invitees–the state of the service(s) in their charge, so as to increase general awareness among everyone who cares, and to improve the operation of the service(s). [...] we talk in detail about the operational performance of the service, and relate that operational performance to design, configuration, or implementation, and make recommendations for how to fix the problems. Connecting the performance of the service with design decisions in a regular meeting is an immensely powerful feedback loop."

### 31.1.1 Agenda

- Upcoming production changes: intent is giving near-term horizon visibility
- Metrics: "Keeping track of how your latency figures, CPU utilization figures, etc., change over time is incredibly valuable for developing a feel for the performance envelope of a system."
- Outages
- Paging events
- Nonpaging events

> An issue that probably should have paged, but didn't (e.g. "this would have told us this outage was coming")

> An issue that is not pageable but requires attention (e.g., low-impact data corruption; slowness in some non-user-facing dimension of the system; amount of reactive operational work)

> An issue that is not pageable and does not require attention

- Prior action items

#### 31.1.2 Attendance

## 31.2 Collaboration within SRE

"Because our *raison d'tre* is bringing value through technical mastery, and technical mastery tends to be hard, we therefore try to find a way to have mastery over some related subset of systems or infrastructures, in order to decrease cognitive load. Specialization is one way of accomplishing this objective; i.e., team X works only on product Y. Specialization is good, because it leads to higher chances of improved technical mastery, but it's also bad, because it leads to siloization and ignorance of the broader picture. We try to have a crisp team charter to define what a team will-and more importantly, won't–support"

#### 31.2.1 Team Composition

"Formally, SRE teams have the roles of "tech lead" (TL), "manager" (SRM), and "project manager" (also known as PM, TPM, PgM)."

"the tech lead is responsible for technical direction within the team, and can lead in a variety of ways–everything from carefully commenting on everyone's code, to holding quarterly direction presentations, to building consensus in the team. In Google, TLs can do almost all of a manager's job, because our managers are highly technical, but the manager has two special responsibilities that a TL doesn't have: the performance management function, and being a general catchall for everything that isn't handled by someone else."

#### 31.2.2 Techniques for Working Effectively

"For collaborations outside the building, effectively working across time zones implies either great written communication, or lots of travel to supply the in-person experience that is deferrable but ultimately necessary for a high-quality relationship."

## 31.3 Case Study of Collaboration in SRE: Viceroy

"for various reasonings, monitoring dashboard frameworks were a particularly fertile ground for duplication of work. (In this particular case, the road to hell was indeed paved with JavaScript.)"

"The incenetives that led to the serious litter problem of many smoldering, abandoned hulks of monitoring frameworks lying around were pretty simple: each team was rewarded for developing its own solution, working outside of the team boundary was hard, and the infrastructure that tended to be provided SRE-wide was typically closer to a toolkit than a product. This environment encouraged individual engineers to use the toolkit ot make another burning wreck rather than fix the problem for the largest number of people possible (an effort that would therefore take much longer)."

#### 31.3.1 The Coming of the Viceroy

#### 31.3.2 Challenges

"the extended team of contributors was fairly dynamic. [...] Thus, the developer contributor pool, which was inherently larger than the core Viceroy team, was characterized by a significant amount of churn."

"Adding new people to the project required training each contributor on the overall design and structure of the system, which took some time. On the other hand, when an SRE contributed to the core functionality of Viceroy and later returned to their own team, they were a local expert on teh system. That unanticipated dissemination of local Viceroy experts drove more usage and adoption."

"casual contributions were both useful and costly. The primary cost was the dilution of ownership: once features were delivered and the person left, the features became unsupported over time, and were generally dropped."

"the Viceroy team found it difficult to completely own a component that had significant (determining) contributions for distributed sites. Even with the best will in the world, people generally default to the path of least resistance and discuss issues or make decisions locally without involving remote owners, which can lead to conflict."

#### 31.3.3 Recommendations

"Writing things down is one of the major techniques you have to offset physical and/or logical distance–use it."

"Standards are important. Coding style guidelines are a good start, but they're usually quite tactical and therefore only a starting point for establishing team norms. Every time there is a debate about which choice to make on an issue, argue it out fully with the team but with a strict time limit. Then pick a solution, document it, and move on. If you can't agree, you need to pick some arbitrator that everyone respects, and again just move forward."

### 31.4 Collaboration Outside SRE

### 31.5 Case Study: Migrating DFP to F1

### 31.6 Conclusion

## 32 The Evolving SRE Engagement Model

### 32.1 SRE Engagement: What, How, and Why

### 32.2 The PRR Model

Production Readiness Review (PRR)

"A PRR is considered a prerequisite for an SRE team to accept responsibility for managing the production aspects of a service."

### 32.3 The SRE Engagement Model

"SRE seeks production responsibility for important services for which it can make concrete contributions to reliability."

- System architecture and interservice dependencies
- Instrumentation, metrics, and monitoring
- Emergency response
- Capacity planning
- Change management
- Performance: availability, latency, and efficiency

#### 32.3.1 Alternative Support

##### 32.3.1.1 Documentation

##### 32.3.1.2 Consultation

### 32.4 Production Readiness Reviews: Simple PRR Model

Objectives

- Verify that a service meets accepted standards of production setup and operational readiness, and that service owners are prepared to work with SRE and take advantage of SRE expertise.
- Improve the reliability of the service in production, and minimize the number and severity of incidents that might be expected. A PRR targets all aspects of production that SRE cares about.

> "After sufficient improvements are made and the service is deemed ready for SRE support, an SRE team assumes its production responsibilities"

#### 32.4.1 Engagement

#### 32.4.2 Analysis

#### 32.4.3 Improvements and Refactoring

#### 32.4.4 Training

#### 32.4.5 Onboarding

#### 32.4.6 Continuous Improvement

### 32.5 Evolving the Simple PRR Model: Early Engagement

#### 32.5.1 Candidates for Early Engagement

#### 32.5.2 Benefits of the Early Engagement Model

##### 32.5.2.1 Design phase

##### 32.5.2.2 Build and implementation

##### 32.5.2.3 Launch

> "SRE might help implement a "dark launch" setup, in which part of the traffic from existing users is sent to a new service in addition to being sent to the live production service. The responses from the new service are "dark" since they are thrown away and not actually shown to users. Practices such as dark launches allow the team to gain operational insight, resolve issues without impacting existing users, and reduce the risk fo encountering issues after launch."

#### 32.5.2.4 Post-launch

#### 32.5.2.5 Disengaging from a service

"Sometimes a service doesn't warrant full-fledged SRE team management–this determination might be made post-launch, or SRE might engage with a service but never officially take it over. This is a positive outcome, because the service has been engineered to be reliable and low maintenance, and can therefore remain with the development team."

## 32.6 Evolving Services Development: Frameworks and SRE Platform

### 32.6.1 Lessons Learned

"Onboarding each service required two or three SREs and tpyically lasted two or three quarters. The lead times for a PRR were relatively high (quarters away)."

"Due to differing software practices across services, each production feature was implemented differently."

"A review of common service issues and outages revealed certain patterns, but there was no way to easily replicate fixes and improvements across services. Typical examples included service overload situations and data hot-spotting."

### 32.6.2 External Factors Affecting SRE

"Because each service has a base fixed operational cost, even simple servcies demand more staffing. Microservices also implay an expectation of lower lead time for deployment, which was not possible with the previous PRR model (which had a lead time of months)."

### 32.6.3 Toward a Structural Solution: Frameworks

- Codified best practices: "commit what works well in production into code, so services can simply use this code and become "production ready" by design."
- Reusable solutions
- A common production platform with a common control surface
- Easier automation and smarter systems

"SRE builds framework modules to implement canonical solutions for the concerned production area. As a result, development teams can focus on the business logic, because the framework already takes care of correct infrastructure use."

"A framework essentially is a prescriptive implementation for using a set of software components and a canonical way of combining these components."

"Frameworks provide multiple upfront gains in consistency and efficiency. They free developers from having to glue together and configure individual components in an ad hoc service-specific manner, in ever-so-slightly incompatible ways, that then have to be manually reviewed by SREs."

#### 32.6.4 New Service and Management Benefits

##### 32.6.4.1 Significantly lower operational overhead

"A production platform built on top of frameworks with stronger conventions significantly reduced operational overhead"

"supports strong conformance tests for coding structure, dependencies, tests, coding style guidelines, and so on. This functionality also improves user data privacy, testing, and security conformance."

"built-in service deployment, monitoring, and automation for all services."

##### 32.6.4.2 Universal support by design

"services that don't receive full SRE support can be built to use production features that are developed and maintained by SREs. This practice effectively breaks the SRE staffing barrier. Enabling SRE-supported production standards and tools for all teams improves the overall service quality across Google."

##### 32.6.4.3 Faster, lower overhead engagements

"Less cognitive burden for the SRE teams managing services built using frameworks."

##### 32.6.4.4 A new engagement model based on shared responsibility

"The original SRE engagement model presented only two options: either full SRE support, or approximately no SRE engagement."

### 32.7 Conclusion

# 33 Lessons Learned from Other Industries

## 33.1 Meet Our Industry Veterans

## 33.2 Preparedness and Disaster Testing

#### 33.2.1 Relentless Organizational Focus on Safety

#### 33.2.2 Attention to Detail

#### 33.2.3 Swing Capacity

#### 33.2.4 Simulations and Live Drills

#### 33.2.5 Training and Certification

#### 33.2.6 Focus on Detailed Requirements Gathering and Design

#### 33.2.7 Defense in Depth and Breadth

## 33.3 Postmortem Culture

"Corrective and preventative action (CAPA) is a well-known concept for improving reliability that focuses on the systematic investigation of root causes of identified issues or risks in order to prevent recurrence."

"Lifeguarding has a deeply embedded culture of post-incident analysis and action planning. Mike Doherty quips, "If a lifeguard's feet go in the water, there will be paperwork!""

## 33.4 Automating Away Repetitive Work and Operational Overhead

"Dan Sheridan discussed automation as deployed in the UK nuclear industry. Here, a rule of thumb dictates that if a plant is required to respond to a given situation in less than 30 minutes, that response must be automated."

## 33.5 Structured and Rational Decision Making

"Many industries heavily focus on playbooks and procedures rather than open-ended problem solving. Every humanly conceivable scenario is captured in a checklist or in "the binder." When something goes wrong, this resource is the authoritative source for how to react. This prescriptive approach works for industries that evolve and develop relatively slowly, because the scenarios of what could go wrong are not constantly evolving due to system updates or changes. This approach is also common in industries in which the skill level of the workers may be limited, and the best way to make sure that people will respond appropriately in an emergency is to provide a simple, clear set of instructions."

## 33.6 Conclusions

"A main takeaway of our cross-industry survey was that in many parts of its software business, Google has a higher appetite for velocity than players in most other industries. The ability to move or change quickly must be weighed against the differing implications of a failure."

"many of our software businesses such as Search make conscious decisions as to how reliable "reliable enough" really is."

"Google has that flexibility in most of our software products and services, which operate in an environment in which lives are not directly at risk if something goes wrong. Therefore, we're able to use tools such as error budgets as a means to "fund" a culture of innovation and calculated risk tasking."

# 34 Conclusion

"an analogy between how SRE thinks about complex systems and how the aircraft industry has approached plane flight is useful in conceptualizing how SRE has evolved and matured over time."

"How has every other element of the flight experience–safety, capacity, speed, and reliability–scaled up so beautifully, while there are still only two pilots? [...] The interfaces to the plane's operating systems are well thought out and approachable enough that learning how to pilot them in normal conditions is not an insurmountable task. yet these interfaces also provdie enough flexibility, and the people operating them are sufficiently trained, that responses to emergencies are robust and quick. The cockpit was designed by people who understand complex systems and how to present them to humans in a way that's both consumable and scalable. The systems underlying the cockpit have all the same properties as discussed in this book: availability, performance optimization, change management, monitoring and alerting, capacity planning, and emergency response."

# 35 Appendix A: Availability Table

# 36 Appendix B: A Collection of Best Practices for Production Services

## 36.1 Fail Sanely

"Sanitize and validate configuration inputs, and responte to implausible inputs by *both* continuing to operate in the previous state *and* alerting to the receipt of bad input."

"it's generally safer for systems to continue functioning with their previous configuration and await a human's approval before using the new, perhaps invalid, data."

## 36.2 Progressive Rollouts

"Nonemergency rollouts *must* proceed in stages. Both configuration and binary changes introduce risk, and you mitigate this risk by applying the change to small fractions of traffic and capacity at one time. The size of your service or rollout, as well as your risk profile, will inform the percentages of production capacity to which the rollout is pushed, and the appropriate time frame between stages. It's also a good idea to perform different stages in different geographies, in order to detect problems related to diurnal traffic cycles and geographical traffic mix differences."

"Rollouts should be supervised. To ensure that nothing unexpected is occurring during the rollout, it must be monitored either by the engineer performing the rollout stage or–preferably–a demonstrably reliable monitoring system. If unexpected behavior is detected, roll back first and diagnose afterward in order to minimize Mean Time to Recovery."

## 36.3 Define SLOs Like a User

## 36.4 Error Budgets

## 36.5 Monitoring

Monitoring may have only three output types:

- *Pages*: A human must do something *now*
- *Tickets*: A human must do something within a few days
- *Logging*: No one need look at this output immediately, but it's available for later analysis if needed

## 36.6 Postmortems

## 36.7 Capacity Planning

"Provision to handle a simultaneous planned and unplanned outage, without making the user experience unacceptable; this results in an "N + 2" configuration, where peak traffic can be handled by N instances (possibly in degraded mode) while the largest 2 instances are unavailable."

"Don't mistake day-one load for steady-state load."

## 36.8 Overloads and Failure

"when load is high enough that even degraded responses are too expensive for all queries, practice graceful load shedding, using well-behaved queuing and dynamic timeouts; see Chapter 21. Other techniques include answering requests after a significant delay ("tarpitting") and choosing a consistent subset of clients to receive errors, preserving a good user experience for the remainder."

"Retries can amplify low error rates into higher levels of traffic, leading to cascading failures (see Chapter 22). Respond to cascading failures by dropping a fraction of traffic (including retries!) upstream of the system once total load exceeds total capacity."

"Every client that makes an RPC must implement exponential backoff (with jitter) for retries, to dampen error amplification."

## 36.9 SRE Teams

"We've found that at least eight people need to be part of the on-call team, in order to avoid fatigue and allow sustainable staffing and low turnover. Preferably, those on-call should be in two well-separated geographic locations (e.g., California and Ireland) to provide a better quality of life by avoiding nighttime pages; in this case, six people at each site is the minimum team size."

## 37 Appendix C: Example Incident State Document

## 38 Appendix D: Example Postmortem

## 39 Appendix E: Launch Coordination Checklist

## 40 Appendix F: Example Production Meeting Minutes