FAIL at Scale

RELIABILITY IN THE FACE OF RAPID CHANGE

BY BEN MAURER, FACEBOOK

Failure is part of engineering any large-scale system. One of Facebook's cultural values is embracing failure. This can be seen in the posters hung around the walls of our Menlo Park headquarters: "What Would You Do If You Weren't Afraid?" and "Fortune Favors the Bold."

To keep Facebook reliable in the face of rapid change we study common patterns in failures and build abstractions to address them. These abstractions ensure that best practices are applied across our entire infrastructure. To guide our work in building reliability abstractions we must understand our failures. We do this by building tools to diagnose issues and by creating a culture of reviewing incidents in a way that pushes us to make improvements that prevent future failures.

IN THE WORDS OF BEN MAURER, "FACEBOOK IN 30 SECONDS"

WHY DO FAILURES HAPPEN?

While every failure has a unique story, many failures boil down to a small number of fundamental root causes.

Individual machine failures

Often an individual machine will run into an isolated failure that doesn't affect the rest of the infrastructure. For example, maybe a machine's hard drive has failed, or a service on a particular machine has experienced a bug in code, such as memory corruption or a deadlock.

The key to avoiding individual machine failure is automation. Automation works best by combining known failure patterns (such as a hard drive with S.M.A.R.T. errors) with a search for symptoms of an unknown problem (for example, by swapping out servers with unusually slow response times). When automation finds symptoms of an unknown problem, manual investigation can help develop better tools to detect and fix future problems.

Legitimate workload changes

Sometimes Facebook users change their behavior in a way that poses challenges for our infrastructure. During major world events, for example, unique types of workloads may stress our infrastructure in unusual ways. When Barack Obama won the 2008 U. S. Presidential election, his Facebook page experienced record levels of activity. Climactic plays in major sporting events such as the Super Bowl or World Cup result in an extremely high number of posts. Load testing, including "dark launches" where a feature is activated but not visible to the user, helps ensure that new features are able to handle load.

Statistics gathered during such events often provide a unique perspective on a system's design. Oftentimes, major events cause changes in user behavior (for example, by creating focused activity around a particular object). Data about these changes often points to design decisions that will allow smoother operation in subsequent events.

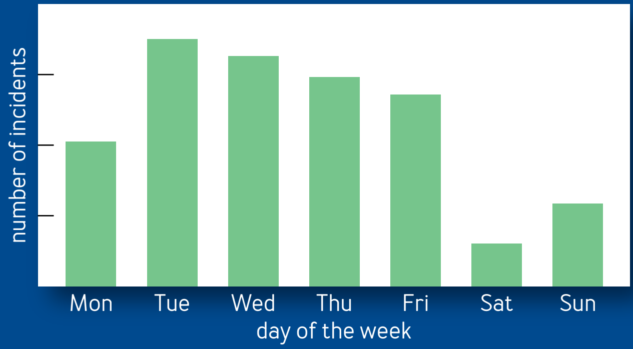
Human error

Given that Facebook encourages engineers to "Move Fast and Break Things"—another one of the posters that adorns the offices—one might expect that many errors are caused by humans. Our data suggests that human error is a factor in our failures. Figure 1 includes data from an analysis of the timing of events severe enough to be considered an SLA (service-level agreement) violation. Each violation indicates an instance where our internal reliability goals were not met and caused an alert to be generated. Because our goals are strict most of these incidents are minor and not noticeable to users of the site. Figure 1a shows how incidents happened substantially less on Saturday and Sunday even though traffic to the site remains consistent throughout the week. Figure 1b shows a six-month period during which there were only two weeks with no incidents: the week of Christmas and the week when employees are expected to write peer reviews for each other.

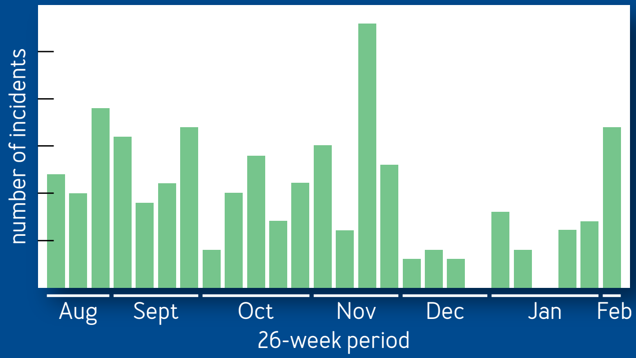
These two data points seem to suggest that when Facebook employees are not actively making changes to infrastructure because they are busy with other things (weekends, holidays, or even performance reviews), the site experiences higher levels of reliability. We believe this is not a result of carelessness on the part of people making

FIGURE 1: **SLA VIOLATION EVENTS**

a. Incidents across the week



b. Number of incidents by week



changes but rather evidence that our infrastructure is largely self-healing in the face of non-human causes of errors such as machine failure.

THREE EASY WAYS TO CAUSE AN INCIDENT

While failures have different root causes, we have found three common pathologies that amplify failures and cause them to become widespread. For each pathology, we have developed preventative measures that mitigate widespread failure.

Rapidly deployed configuration changes

Configuration systems tend to be designed to replicate changes quickly on a global scale. Rapid configuration change is a powerful tool that can let engineers quickly manage the launch of new products or adjust settings. However, rapid configuration change means rapid failure when bad configurations are deployed. We use a number of practices to prevent configuration changes from causing failure.

➡ **Make everybody use a common configuration system.** Using a common configuration system ensures that procedures and tools apply to all types of configuration. At Facebook we have found that teams are sometimes tempted to handle configurations in a one-off way. Avoiding these temptations and managing configurations in a unified way has made the configuration system a leveraged way to make the site more reliable.

➡ **Statically validate configuration changes.** Many configuration systems allow loosely typed configuration, such as JSON structures. These types of configurations make it easy for an engineer to mistype the name of a field, use a string where an integer was required, or make other simple errors. These kinds of straightforward errors are best caught using static validation. A structured format (for example, at Facebook we use Thrift)[4] can provide the most basic validation. It is not unreasonable, however, to write programmatic validation to validate more detailed requirements.

➡ **Run a canary.** First deploying your configuration to a small scope of your service can prevent a change from being disastrous. A canary can take multiple forms. The most obvious is an A/B test, such as launching a new configuration to only 1 percent of users. Multiple A/B tests can be run concurrently, and you can use data over time to track metrics.

For reliability purposes, however, A/B tests do not satisfy all of our needs. A change that is deployed to a small number of users, but causes implicated servers to crash or run out of memory, will obviously create impact that goes beyond the limited users in the test. A/B tests are also time consuming. Engineers often wish to push out minor changes without the use of an A/B test. For this reason, Facebook infrastructure automatically tests out new configurations on a small set of servers. For example, if we wish to deploy a new A/B test to 1 percent of users, we will first deploy the test to 1 percent of the users that hit a small number of servers. We monitor these servers for a short amount of time to ensure that they do not crash or have other highly visible problems. This mechanism provides a basic "sanity check" on all changes to ensure that they do not cause widespread failure.

➡ **Hold on to good configurations.** Facebook's configuration system is designed to retain good configurations in the face of failures when updating those configurations. Developers tend naturally to create configuration systems that will crash when they receive updated configurations that are invalid. We prefer systems that retain old configurations in these types of situations and raise alerts to the system operator that the configuration failed to update. Running with a stale configuration is generally preferable to returning errors to users.

➡ **Make it easy to revert.** Sometimes, despite all best efforts, a bad configuration is deployed. Quickly finding and reverting the change is key to resolving this type of issue. Our configuration system is backed by version control, making it easy to revert changes.

Hard dependencies on core services

Developers tend to assume that core services—such as configuration management, service discovery, or storage systems—never fail. Even brief failures in these core services, however, can turn into large-scale incidents.

➡ **Cache data from core services.** Hard dependencies on these types of services are often not necessary. The data these services return can be cached in a way that allows for the majority of services to continue operating during a brief outage of one of these systems.

➡ **Provide hardened APIs to use core services.** Core services are best complemented by common libraries that follow best practices when using these core services. For example, the libraries might provide good APIs for managing the cache or good failure handling.

➡ **Run fire drills.** You might think you are able to survive an outage of a core service, but you never know until you try. For these types of outages we have had to develop systems for fire drills ranging from fault injection applied to a single server to manually triggered outages of entire data centers.

While small amounts of additional latency can be easily handled by the Facebook infrastructure, large amounts of latency lead to cascading failures.

Increased latency and resource exhaustion

Some failures result in services having increased latency to clients. This increase in latency could be small (for example, think of a human configuration error that results in increased CPU usage that is still within the service's capacity), or it could be nearly infinite (a service where the threads serving responses have deadlocked). While small amounts of additional latency can be easily handled by Facebook's infrastructure, large amounts of latency lead to cascading failures. Almost all services have a limit to the number of outstanding requests. This limit could be due to a limited number of threads in a thread-per-request service, or it could be due to limited memory in an event-based service. If a service experiences large amounts of extra latency, then the services that call it will exhaust their resources. This failure can be propagated through many layers of services, causing widespread failure.

Resource exhaustion is a particularly damaging mode of failure because it allows the failure of a service used by a subset of requests to cause the failure of all requests. For example, imagine that a service calls a new experimental service that is only launched to 1% of users. Normally requests to this experimental service take 1 millisecond, but due to a failure in the new service the requests take 1 second. Requests for the 1% of users using this new service might consume so many threads that requests for the other 99% of users are unable to run.

We have found a number of techniques that can avoid this type of buildup with a low false positive rate.

➡ **Controlled Delay.** In analyzing past incidents involving latency, we found that many of our worst incidents involved large numbers of requests sitting in queues awaiting processing. The services in question had a resource limit (such as a number of active threads or memory) and would buffer requests in order to keep usage below the limit. Because the services were unable to keep up with the rate of incoming requests, the queue would grow larger and larger until it hit an application-defined limit. To address this situation, we wanted to limit the size of the queue without impacting reliability during normal operations. We studied the research on bufferbloat as our problems seemed similar—the need to queue for reliability without causing excessive latency during congestion. We experimented with a variant of the CoDel (controlled delay) algorithm:

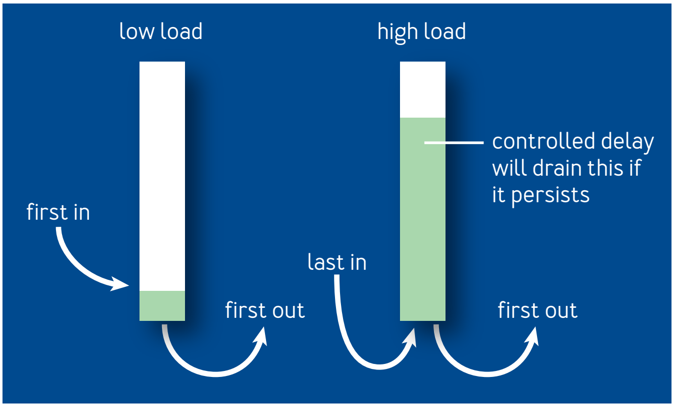
```
onNewRequest(req, queue):

  if (queue.lastEmptyTime() < (now - N seconds)) {
    timeout = M ms
  } else {
    timeout = N seconds;
  }
  queue.enqueue(req, timeout)
```

In this algorithm, if the queue has not been empty for the last **N** milliseconds, then the amount of time spent in the queue is limited to **M** milliseconds. If the service has been able to empty the queue within the last **N** milliseconds, then the time spent in the queue is limited to `N` milliseconds. This algorithm prevents a standing queue (because the `lastEmptyTime` will be in the distant past, causing an `M`-ms queuing timeout) while allowing short bursts of queuing for reliability purposes. While it might seem counterintuitive to have requests with such short timeouts, this process allows requests to be quickly discarded rather than build up when the system is not able to keep up with the rate of incoming requests. A short timeout ensures that the server always accepts just a little bit more work than it can actually handle so it never goes idle.

An attractive property of this algorithm is that the values of `M` and `N` tend not to need tuning. Other methods of solving the problem of standing queues, such as setting a limit on the number of items in the queue or setting a timeout for the queue, have required tuning on a per-service basis. We have found that a value of 5 milliseconds for M and 100 ms for N tends to work well across a wide set of use cases. Facebook's open source Wangle library[5] provides an implementation of this algorithm which is used by our Thrift[4] framework.

➡ **Adaptive LIFO (last-in, first-out).** Most services process queues in FIFO (first-in first-out) order. During periods of high queuing, however, the first-in request has often been sitting around for so long that the user may have aborted the action that generated the request. Processing the first-in request first expends resources on a request that is less likely to benefit a user than a request that has just arrived. Our services process requests using adaptive LIFO. During normal operating conditions, requests are processed

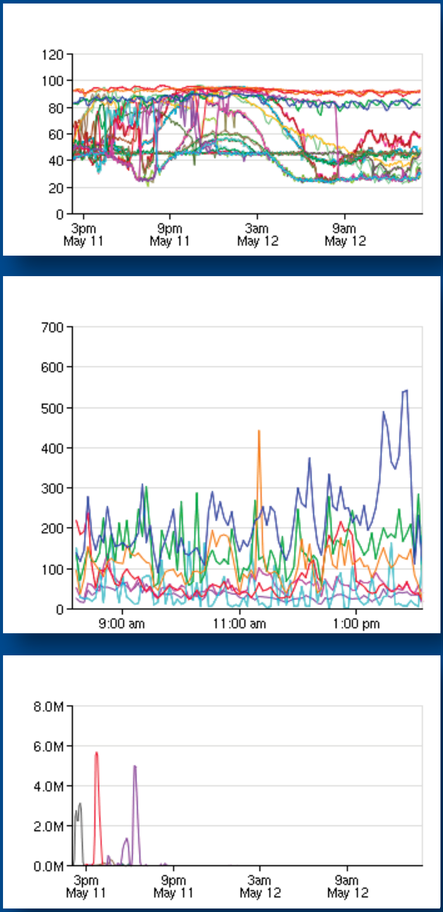
FIGURE 2: LIFO (LEFT) AND ADAPTIVE LIFO WITH CODEL (RIGHT)



in FIFO order, but when a queue is starting to form, the server switches to LIFO mode. Adaptive LIFO and CoDel play nicely together, as shown in figure 2. CoDel sets short timeouts, preventing long queues from building up, and adaptive LIFO places new requests at the front of the queue, maximizing the chance that they will meet the deadline set by CoDel. HHVM3, Facebook's PHP runtime, includes an implementation of the Adaptive LIFO algorithm.

➡ **Concurrency Control.** Both CoDel and adaptive LIFO operate on the server side. The server is often the best place to implement latency-preventing measures—a server tends to serve a large number of clients and often has more information than its clients possess. Some failures are so severe, however, that server-side controls are not able to kick in. For this reason, we have implemented a stopgap measure in clients. Each client keeps track of the number of outstanding outbound requests on a per-service basis. When new requests are sent, if the number of outstanding requests

FIGURE 3: **DIFFICULT-TO-READ CHARTS FROM A TYPICAL DASHBOARD**



to that service exceeds a configurable number, the request is immediately marked as an error. This mechanism prevents a single service from monopolizing all its client's resources.

TOOLS THAT HELP DIAGNOSE FAILURES

Despite the best preventative measures, some failures will always occur. During outages the right tools can quickly lead to the root cause, minimizing the duration of the failure.

High-Density Dashboards with Cubism

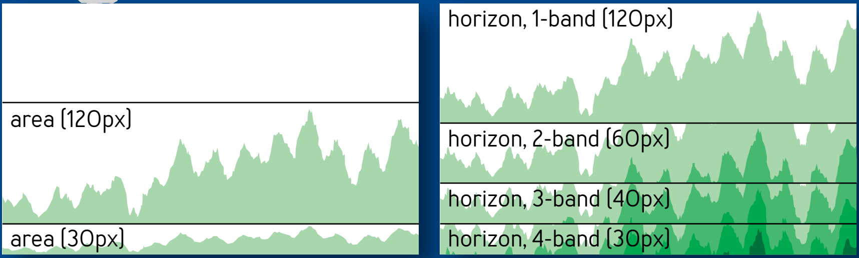
When handling an incident, it is important to have quick access to information. Good dashboards allow engineers quickly to assess the types of metrics that might be abnormal and then use this information to hypothesize a root cause. We found, however, that our dashboards grew so large that it was difficult to navigate them quickly, and that charts shown on those dashboards had too many lines to read at a glance, as in figure 3.

To address this, we built our top-level dashboards using Cubism,[2] a framework for creating horizon charts—line charts that use color to encode information more densely, allowing for easy comparison of multiple similar data series. For example, we use Cubism to compare metrics between different data centers. Our tooling around Cubism allows for easy keyboard navigation so engineers can view multiple metrics quickly. Figure 4 shows the same data set at various heights using area charts and horizon charts. In the area chart version, the 30-pixel version is hard to read. On the other hand, the horizon chart makes it extremely easy to find the peak value, even at a height of 30 pixels.

What just changed?

Since one of the top causes of failure is human error, one of

FIGURE 4: **AREA CHARTS (LEFT) AND HORIZON CHARTS (RIGHT)**



the most effective ways of debugging failures is to look for what humans have changed recently. We collect information about recent changes ranging from configuration changes to deployments of new software in a tool called OpsStream. However, we have found over time that this data source has become extremely noisy. With thousands of engineers making changes, there are often too many to evaluate during an incident.

To solve this problem, our tools attempt to correlate failures with relevant changes. For example, when an exception is thrown, in addition to outputting the stack trace, we output any configuration settings read by that request that have had their values changed recently. Often, the cause of an issue that generates many stack traces is one of these configuration values. We can then quickly respond to the issue—for example, by reverting the configuration and involving the engineer who made the change.

LEARNING FROM FAILURE

After failures happen, our incident-review process helps us learn from these incidents.

The goal of the incident-review process is not to assign blame. Nobody has been fired because an incident he or she caused came under review. The goal of the review is to understand what happened, remediate situations that allowed the incident to happen, and put safety mechanisms in place to reduce the impact of future incidents.

A methodology for reviewing incidents

Facebook has developed a methodology called DERP (for detection, escalation, remediation, and prevention) to aid in productive incident reviews.

- **Detection.** How was the issue detected—alarms, dashboards, user reports?
- **Escalation.** Did the right people get involved quickly? Could these people have been brought in via alarms rather than manually?
- **Remediation.** What steps were taken to fix the issue? Can these steps be automated?
- **Prevention.** What improvements could remove the risk of this type of failure happening again? How could you have failed gracefully, or failed faster to reduce the impact of this failure?

DERP helps analyze every step of the incident at hand. With the aid of this analysis, even if you cannot prevent this type of incident from happening again, you will at least be able to recover faster the next time.

MOVE FAST BY BREAKING FEWER THINGS

A "move-fast" mentality does not have to be at odds with reliability. To make these philosophies compatible, Facebook's infrastructure provides safety valves: our configuration system protects against rapid deployment of bad configurations; our core services provide clients with hardened APIs to protect against failure; and our core libraries prevent resource exhaustion in the face of latency. To deal with the inevitable issues that slip through the cracks, we build easy-to-use dashboards and tools to help find recent changes that might cause the issues under investigation. Most importantly, after an incident we use lessons learned to make our infrastructure more reliable.

References

1. CoDel (controlled delay) algorithm; http://queue.acm.org/detail.cfm?id=2209336.
2. Cubism; https://square.github.io/cubism/.
3. HipHop Virtual Machine (HHVM); https://github.com/facebook/hhvm/blob/43c20856239cedf842b2560fd768038f52b501db/hphp/util/job-queue.h#L75.
4. Thrift framework; https://github.com/facebook/fbthrift.
5. Wangle library; https://github.com/facebook/wangle/blob/master/wangle/concurrent/Codel.cpp.

LOVE IT, HATE IT? LET US KNOW feedback@queue.acm.org

Ben Maurer *is the tech lead of the Web Foundation team at Facebook, which is responsible for the overall performance*

and reliability of Facebook's user-facing products. Ben joined Facebook in 2010 as a member of the infrastructure team. Before Facebook, he co-founded reCAPTCHA with Luis von Ahn. Recently, Ben worked with the U.S. Digital Service to improve the use of technology within the federal government.

AD