Decision-Making Using Service Level Objectives

ALEX HIDALGO

Alex Hidalgo is a Site Reliability Engineer and author of the forthcoming *Implementing Service Level Objectives* (O'Reilly Media, September 2020). During his career he has developed a deep love for sustainable operations, proper observability, and use of SLO data to drive discussions and make decisions. Alex's previous jobs have included IT support, network security, restaurant work, t-shirt design, and hosting game shows at bars. When not sharing his passion for technology with others, you can find him scuba diving or watching college basketball. He lives in Brooklyn with his partner Jen and a rescue dog named Taco. Alex has a BA in philosophy from Virginia Commonwealth University. sometimesitsalex@gmail.com @ahidalgosre

Service level objectives, or SLOs, are quickly becoming the latest industry buzzword. Engineers want them, leadership demands them, and job postings increasingly ask for experience with them. However, SLOs are meaningless unless they are understood as more than just the latest industry jargon. There are true, real-world benefits to adopting an SLO-based approach to reliability. I will explain why they are important and how you can use them most effectively to have discussions that lead to better decisions.

Using service level objectives to measure the reliability of services is getting more attention than ever before. This is partly due to the success of the first two Google-authored site reliability engineering (SRE) books. But it also seems that many people actually resonate with the approach and find it an intuitive concept to follow. While it is possible that many organizations are forcing their teams to adopt SLOs via mandate just to ensure they're on board with the latest buzzwords, it also seems likely that many people are finding true value in the approach.

I found only one study tracking the adoption rates of SLO-based approaches in this book: https://www.oreilly.com/library/view/slo-adoption-and/9781492075370/. Instead, I'll have to rely on the general anecdotal evidence I have in terms of how many companies are rolling out products to help people measure SLOs, how many conference talks are focused on them, and how often I personally find myself engaged with people who want to learn more about the process.

While SLO-based approaches to reliability are certainly useful to many people, I also cannot ignore the fact that the very phrase has become a buzzword that is starting to lose meaning. It's not uncommon for words, phrases, and concepts that gain traction and desirability to have their original meanings forgotten. Service level objectives are no different. They provide many benefits, but some of their most important aims have unfortunately become obfuscated by more readily available ones.

SLO-based approaches to reliability give you many benefits, and there are many reasons why organizations may choose to adopt them. Unfortunately, for many they have just become "a thing you do." This is not to say that every organization looking to adopt such an approach has overlooked the benefits of SLOs, but few manage to use them to their full potential.

Let's explore some of the ways you can use the information that service level objectives provide to make better decisions through data. Making better decisions is at the very heart of the SLO approach, and that's often the part that is overlooked.

But first, let's outline how this approach works in a little more detail.

SLO Components

There are three primary components to an SLO-based approach. The first is *service level indicators*, or SLIs. A good SLI is a measurement that tells you how your service is performing from the perspective of your users. In this case, when I say users, they could be anything from paying customers to coworkers to other services that depend on yours—there doesn't strictly have to be a human attached to the other end. In this article, I'll mostly be talking about human users who interact with web services since they are intuitively accessible concepts that almost all of us interact with on a daily basis; however, the concepts and approaches apply to any service and any type of user, even if those users are just other computer systems.

After SLIs, you have *service level objectives*, which are targets for how you want your SLI to perform. While a service level indicator may tell you how quickly a web page loads, an SLO allows you to do things like set thresholds and target percentages. An example SLI might be "Web pages are fully rendered in the user's browser within 2500 ms." Building off of that, an SLO might read, "The 95th percentile of web page render times will complete within 2500 ms 99.9% of the time." Service level objectives allow you to set reasonable targets. Nothing is ever perfect, and 100% is impossible for just about everything, but by using SLOs, you can ensure that you're striving for a reasonable target and not an unreachable one.

Finally, you have *error budgets*. An error budget is a way of keeping track of how an SLO has performed over time. If you acknowledge that only 99.9% of the 95th percentile of your web page render times have to complete within 2500 ms, you are also acknowledging that 0.1% of them don't have to. Error budgets give you a way to do the math necessary to determine whether your adherence to your SLO target is suitable for your users, not just in the moment but over the last day, week, month, quarter, or year.

SLIs, SLOs, and error budgets are all data—data that allows you to ask important questions that can drive better decision-making.

- Is our SLI adequately measuring what our users need and expect? If not, we need to figure out a new way to measure this.
- Is our SLO target meaningfully capturing the failure rates our users can tolerate? If not, we need to pick a new target or new thresholds.
- What is our error budget status telling us about how our users have actually experienced our service over time? If we've exceeded the error budget, perhaps we drop feature work and focus on reliability instead.

Decisions about User Experience

A meaningful SLI is one that captures the user experience as closely as possible. Following our simple example from above, it is pretty intuitive to think about the fact that the users of a web service need their pages to load—and to load in an amount of time that won't annoy them. But there is so much more to the user experience than just the concepts of availability and latency. A web service is not doing its job just by being able to render pages in a timely manner. If you're only measuring things like availability and latency, the only data SLO-based approaches can provide you are ones that focus on improving your availability and latency.

A web service is often much more than just serving data to people. Imagine that your web service is a retail site. In such a case, you suddenly have many other user journeys to consider. If you want people to be able to purchase items from you, they need to be able to do exactly that and not just have web pages display in their browser.

For example, a standard retail website often has some sort of *shopping cart* feature—one where a user can add a potential purchase to a list of items they might want to check out with later. This shopping cart feature has to do a lot of things in order to be reliable.

The first is that it needs to do what it is supposed to: if a user wants to add an item to their shopping cart, they should be able to do exactly that. Additionally, it needs to be persistent; a shopping cart isn't much good if it only remains consistent with the wishes of a user for a short amount of time.

It also has to be accurate. There is no sense in allowing customers to add to a list of items they might want to purchase if that list doesn't represent the items they have actually chosen.

Finally, how the user interacts with the shopping cart has to work properly. If an item is added or removed, it should actually be added or removed. If the user expects an icon representing how many items they have in their cart to be updated when they add a new item, that icon should actually update in real-time.

These examples all represent different data points that meaningful SLIs can give you—and these data points help you make decisions. If it's simply the case that your site isn't loading well or quickly enough, you might just need to introduce more resources. However, if the shopping cart isn't working well, the problem could be anything from the JavaScript powering user interactions to the service that talks to the database to the data-storage systems that are ultimately responsible for holding the ones and zeroes. By having the data that multiple meaningful SLIs provide you, you can make better decisions about what you should be measuring in the first place and what areas of your system require the most attention.

Decisions about Tolerable Failures

One of the most attractive aspects of measuring services with SLOs is that the entire discipline acknowledges the fact that nothing is ever perfect. All complex systems fail at some point in time, and because of this fact it is fruitless to aim for 100%. Additionally, it turns out that people already know and are okay with this—whether they're consciously aware of it or not.

For example, if you start streaming a video via a video-streaming service, you have a certain expectation for how long it should take for such a video to buffer before it begins playing in real time. However, if it takes much longer than normal to buffer every once in a while, you likely won't care too much. Most people won't abandon a streaming video platform if one in every 100 videos takes 10 seconds of buffering time instead of three seconds. Failure in the sense that the streaming platform had to buffer too long occasionally is just fine—it just can't happen too often. If videos take 10 seconds to start every single time, people might become annoyed and look for other options.

A good service level objective lies somewhere just beyond what you need for users of your service to be happy. If people are okay with one in every 100 streaming attempts buffering for longer than normal, you should set your SLO target at something like one in every 200 streaming attempts. Exactly where you set this target is up to the data you have available to you and the feedback you're able to collect from your users. The important part is that your SLO should be more strict than the level at which users might decide to leave and use a different option. No matter how refined your SLO target is, you're not always going to reach it, and you don't want your business or organization to suffer if you don't.

Acknowledging failure and accounting for it are at the very base of how SLO-based approaches work. Understand that nothing is perfect, but use SLO data to help you decide how close to perfect you should attempt to be—or risk losing users.

Decisions about Work Focus

Once you have a meaningful SLI and a reasonable SLO target, you can produce an error budget. Error budgets are simply just another data point you can use to make decisions. They're the most complicated part of the stack, but once you can find yourself using error budgets to drive your decision-making, you'll truly understand how the entire SLO-based approach works.

Error budgets are the ultimate decision-making tool once you've established SLIs and SLOs. By measuring how you've performed over a time window, you can drive large-scale decisions that could impact anything from the focus of your team for a single sprint to the focus of an entire company for a quarter.

For example, let's say you have a reasonable measurement of how reliable one particular microservice has been. Using your error budget, you can now also see that you haven't been reliable about 10% of the time over the last month. At this point you can use this data to inform a few different discussions that can fuel decisions.

One example is that you simply haven't been performing well enough, and that you believe that your SLI measurement and your SLO target are well-defined. If this is the case, you might choose to pivot one or more members of your team to focusing on reliability work instead of feature work. You could do this for anything, like the length of an on-call shift to a full sprint or even until you've recovered all of your budget. There are no hard-and-fast rules at play here. Error budgets, like everything else, are just data to help you decide what to do.

Another example is that you've completely depleted your error budget but have reason to think this exact situation is unlikely to arise again. An example of this kind of event could be anything like the disruption of power at a datacenter or just a historically bad bug pushed to production. Just because you've depleted your error budget over time doesn't mean you have to take action. Sometimes it absolutely makes sense to do so: perhaps you need to introduce a better testing infrastructure to your deployment pipeline, or maybe you need to install additional circuits or distribute your footprint geographically to avoid further power disruptions.

The point is that it's totally okay to look at how you've performed in terms of reliability over time and say, "This time we can just continue our current work focus." Error-budget statuses are just another data set you should use to make decisions—they shouldn't be rules that need to be followed every single time you examine your status. It doesn't matter if you're looking at the error budget status for a single small microservice that sees very little traffic or your entire service as viewed from your paying customers. Use error budgets as data to help you think about prioritization.

For a larger service, such as an entire customer-facing web service, burning through all of your error budget probably warrants some stricter decision-making. Even if it was due to your ISP that your users experienced an hour of outage last month, it still probably doesn't make sense for you to do things like perform potentially disruptive chaos engineering or experimentation in your production environment until a significant amount of time has passed. Be reasonable about how you make decisions using your error budgets, and certainly feel free to ignore their status from time to time—but never do so at the expense of your users' experience.

Conclusion

There are many benefits to SLO-based approaches that I don't have room to cover here. They can help you better communicate to other teams about how they should think about the reliability of their own services. They can be excellent tools in reporting to management and product teams. They can also be used for many things outside of computer services, such as examining whether your team's ticket load is too high or whether people aren't taking enough vacation time. An SLO-based approach is simply about thinking about people and users first, acknowledging nothing is perfect, and using some math to help you aim for reasonable targets instead.

But one of the most important parts of this approach is that it allows you to make better data-driven decisions. Don't just implement SLOs because they're popular and a buzzword, or because you heard a conference talk about them, or because upper-management has decided that every team must have one.

Implement SLOs because they give you data you can use to have better discussions and make better decisions—decisions that can help make both your team and your users happier.

ML for Operations

Pitfalls, Dead Ends, and Hope

STEVEN ROSS AND TODD UNDERWOOD

Steven Ross is a Technical Lead in site reliability engineering for Google in Pittsburgh, and has worked on machine learning at Google since Pittsburgh Pattern Recognition was acquired by Google in 2011. Before that he worked as a Software Engineer for Dart Communications, Fishtail Design Automation, and then Pittsburgh Pattern Recognition until Google acquired it. Steven has a BS from Carnegie Mellon University (1999) and an MS in electrical and computer engineering from Northwestern University (2000). He is interested in mass-producing machine learning models. stross@google.com

Todd Underwood is a lead Machine Learning for Site Reliability Engineering Director at Google and is a Site Lead for Google's Pittsburgh office. ML SRE teams build and scale internal and external ML services and are critical to almost every product area at Google. Todd was in charge of operations, security, and peering for Renesys's Internet intelligence services that is now part of Oracle's cloud service. He also did research for some early social products that Renesys worked on. Before that Todd was Chief Technology Officer of Oso Grande, an independent Internet service provider (AS2901) in New Mexico. Todd has a BA in philosophy from Columbia University and a MS in computer science from the University of New Mexico. He is interested in how to make computers and people work much, much better together. tmu@goggle.com

Machine learning (ML) is often proposed as the solution to automate this unpleasant work. Many believe that ML will provide near-magical solutions to these problems. This article is for developers and systems engineers with production responsibilities who are lured by the siren song of magical operations that ML seems to sing. Assuming no prior detailed expertise in ML, we provide an overview of how ML works and doesn't, production considerations with using it, and an assessment of considerations for using ML to solve various operations problems.

Even in an age of cloud services, maintaining applications in production is full of hard and tedious work. This is unrewarding labor, or toil, that we collectively would like to automate. The worst of this toil is manual, repetitive, tactical, devoid of enduring value, and scales linearly as a service grows. Think of work such as manually building/testing/deploying binaries, configuring memory limits, and responding to false-positive pages. This toil takes time from activities that are more interesting and produce more enduring value, but it exists because it takes just enough human judgment that it is difficult to find simple, workable heuristics to replace those humans.

We will list a number of ideas that appear plausible but, in fact, are not workable.

What Is ML?

Machine learning is the study of algorithms that learn from data. More specifically, ML is the study of algorithms that enable computer systems to solve some specific problem or perform some task by learning from known examples of data. Using ML requires training a model on data where each element in the data has variables of interest (features) specified for it. This training creates a model that can later be used to make inferences about new data. The generated model is a mathematical function, which determines the predicted value(s) ("dependent variable(s)") based on some input values ("independent variables"). How well the model's inferences fit the historical data is the objective function, generally a function of the difference between predictions and correct inferences for supervised models. In an iterative algorithm, the model parameters are adjusted incrementally on every iteration such that they (hopefully) decrease the objective function.

Main Types of ML

In order to understand how we'll apply ML, it is useful to understand the main types of ML and how they are generally used. Here are broad categories:

Supervised Learning

A supervised learning system is presented with example inputs and their desired outputs labeled by someone or a piece of software that knows the correct answer. The goal is to learn a mapping from inputs to outputs that also works well on new inputs. Supervised learning is the most popular form of ML in production. It generally works well if your data consist of a large volume (millions to trillions) of correctly labeled training examples. It can be effective with many fewer examples, depending on the specific application, but it most commonly does well with lots of input data.

Think of identifying fruit in an image. Given a set of pictures that either contain apples or oranges, humans do an amazing job of picking out the right label ("apple" or "orange") for the right object. But doing this without ML is actually quite challenging because the heuristics are not at all easy. Color won't work since some apples are green and some oranges are green. Shape won't work because it's hard to project at various angles, and some apples are exceedingly round. We could try to figure out the skin/texture but some oranges are smooth and some apples are bumpy.

With ML we simply train a model on a few hundred (or a few thousand) pictures labeled "orange" or "apple." The model builds up a set of combinations of features that predict whether the picture has an apple or an orange in it.

Unsupervised Learning

The goal of unsupervised learning is to cluster pieces of data by some degree of "similarity" without making any particular opinion about what they are, i.e., what label applies to each cluster. So unsupervised learning draws inferences without labels, such as classifying patterns in the data.

One easy-to-understand use case is fraud detection. Unsupervised learning on a set of transactions can identify small clusters of outliers, where some combination of features (card-not-present, account creation time, amount, merchant, expense category, location, time of day) is unusual in some way.

Unsupervised learning is particularly useful as part of a broader strategy of ML, as we'll see below. In particular, in the example above, clustering outlier transactions isn't useful unless we do something with that information.

Semi-Supervised Learning

The goal of semi-supervised learning is to discover characteristics of a data set when only a subset of the data is labeled. Human raters are generally very expensive and slow, so semi-supervised learning tries to use a hybrid of human-labeled data and automatically "guessed" labels based on those human labels. Heuristics are used to generate assumed labels for the data that isn't labeled, based on its relationship to the data that is labeled.

Semi-supervised learning is often used in conjunction with unsupervised learning and supervised learning to generate better results from less effort.

Reinforcement Learning

In reinforcement learning (RL), software is configured to take actions in an environment or a simulation of an environment in order to accomplish some goal or cumulative set of values. The software is often competing with another system (which may be a prior copy of itself or might be a human) without externally provided labeled training data, following the rules.

Google's DeepMind division is well known for using RL to solve various real-world problems. Famously, this has included playing (and winning) against humans in the strategy game Go [1] as well as the video game StarCraft [2]. But it has also included such practical and important work as optimizing datacenter power utilization [3].

ML for Operations: Why Is It Hard?

Given that ML facilitates clustering, categorization, and actions on data, it is enormously appealing as a system to automate operational tasks. ML offers the promise of replacing the human judgment still used in decisions, such as whether a particular new deployment works well enough to continue the roll-out, and whether a given alert is a false positive or foreshadowing a real outage. Several factors make this more difficult than one might think.

ML produces models that encode information by interpreting features in a fashion that is often difficult to explain and debug (especially with deep neural networks, a powerful ML technique). Errors in the training data, bugs in the ML algorithm implementation, or mismatches between the encoding of data between training and inference will often cause serious errors in the resulting predictions that are hard to debug. Below we summarize some common issues.

ML Makes Errors

ML is probabilistic in nature, so it will not always be right. It can classify cats as dogs or even blueberry muffins [4] as dogs a small fraction of the time, especially if the data being analyzed is significantly different from any specific training example. Of course, humans make errors as well, but we are often better able to predict, tolerate, and understand the types of errors that humans make. Systems need to be designed so such occasional gross errors will be tolerable, which sometimes requires sanity tests on the result (especially for numerical predictions).

Large Problem Spaces Require Lots of Training Data

The more possible combinations of feature values that a model needs to deal with, the more training data it requires to be accurate. In other words, where many factors could contribute to a particular labeling or clustering decision, more data is required. But in large feature spaces, there may be a large difference between examples being analyzed and the closest training data, leading to error caused by trying to generalize over a large space. This is one of the most serious issues with using ML in operations, as it is often hard to find sufficient correctly labeled training data, and there are often many relevant variables/features.

Specifically, the problem space of production engineering or operations is much messier than the space of fruit categorization.

In practice, it turns out to be quite difficult to get experts to categorize outages, SLO violations, and causes in a mutually consistent manner. Getting good labels is going to be quite difficult.

Training Data Is Biased Relative to Inference Demand

The data you use to train your model may be too different from the data you're trying to cluster or categorize. If your training data only cover a particular subset of all things the model might need to infer over, all the other slices it wasn't trained on will see higher errors because of their divergence from the training data. Additionally, if the statistical distribution of classifications in the training data differs from the statistical distribution in the real world, the model will make skewed predictions, thinking that things that are more common in the training set are more common in the real world than they really are. For example, if the training data had 10 million dogs and 1000 cats, and dogs and cats are equally likely in the inference examples, it will tend to infer the presence of a dog more often than it should.

Lack of Explainability

Many of the best performing ML systems make judgments that are opaque to their users. In other words, it is often difficult or impossible to know why, in human intelligible terms, an ML model made a particular decision with respect to an example. In some problem domains, this is absolutely not a difficulty. For example, if you have a large number of false positive alerts for a production system and you're simply trying to reduce that, it's not generally a concern to know that an ML model will use unexpected combinations of features to decide which alerts are real. For this specific application, as long as the model is accurate, it is useful. But models with high accuracy due purely to correlation rather than causation do not support decision making. In other situations aspects of provable fairness and lack of bias are critical. Finally, sometimes customers or users are simply uncomfortable with systems that make decisions that cannot be explained to them.

Potential Applications of ML to Operations

Given all of these challenges, it will be useful to examine several potential applications of ML to operations problems and consider which of these is feasible or even possible.

Monitoring

For complex systems, the first problem of production maintenance is deciding which of many thousands of variables to monitor. Candidates might include RAM use by process, latency for particular operations, request rate from end users, timestamp of most recent build, storage usage by customer, number, and type of connections to other microservices, and so on. The possibilities of exactly what to monitor seem unbounded.

Systems and software engineering sometimes suggest using ML to identify the most relevant variables to monitor. The objective would be to correlate particular data series with the kinds of events that we are most interested in predicting—for example, outages, slowness, capacity shortfalls, or other problems.

In order to understand why this is a difficult problem, let us consider how to build an ML model to solve it. In order to use ML to create a dashboard that highlights the best metrics to see any current problems with your system, the best approach will be to treat the problem as a supervised multiclass prediction problem. To address that problem we will need:

- A class to predict for every metric of interest
- Labels for all classes that were helpful for millions of production events of concern
- Training and periodic retraining of your model as you fix bugs and create new ones with failure types shifting over time
- Periodic (potentially on page load) inferring with the model over which metrics should be shown to the user.

There are other complexities, but the biggest issue here is that you need millions of labeled training examples of production events of concern. Without millions of properly categorized examples, simple heuristics, for example that operators select the metrics that appear to be the most relevant, are likely to be as or more effective and at a fraction of the cost to develop and maintain. Simple heuristics also have several advantages over ML, as previously mentioned. We hope you don't have millions of serious problematic events to your production infrastructure to train over. However, if your infrastructure is of a scale and complexity that you think that you will, eventually, have an appropriate amount of data for this kind of application, you should begin accumulating and structuring that data now.

Alerting

Most production systems have some kind of manually configured but automated alerting system. The objective of these systems is to alert a human if and only if there is something wrong with the system that cannot be automatically mitigated by the system itself.

The general idea of an ML-centric approach to alerting is that once you have determined which time series of data are worth *monitoring* (see above) it might be possible to automatically and semi-continuously correlate values and combinations of these. To accomplish this we can start with every alert that we have or could easily have and create a class for each.

We then need to create positive and negative labels. Positive labels are applied to the alerts that were both useful and predictive of some serious problem in the system that actually required human intervention. Negative labels are the opposite: either not useful or not predictive of required human intervention. We need to label many events, those where something bad was happening and those where everything was fine, and continuously add new training examples. To scope the effort, we estimate that we will need at least tens of thousands of positive examples and probably even more (millions, most likely) of negative examples in order to have a pure-ML solution that is able to differentiate real problems from noise more effectively than a simple heuristic. We are not discussing potential hybrid heuristic + ML solutions here since, in many practical setups, this will lead to increased complexity from integrating two systems that need to be kept in sync for the intended outcome, which is unlikely to be worth the extra effort.

Even if we had all these labels (and they're correct) and a good model, which we know to be difficult from the monitoring case above, the on-call will still need to know where to look for the problem. While we may be able to correlate anomalous metrics with a confident alerting signal, covering the majority of alert explanations this way would not be enough. For as long as the fraction of "unexplainable" alerts is perceived by alert recipients as high, the explainability problem makes adoption cumbersome at best. This is the problem of explainability.

Canarying/Validation

Pushing new software to production frequently or continuously as soon as it is effectively tested poses risks that new software will sometimes be broken in ways the tests won't catch. The standard mitigation for this is to use a canary process that gradually rolls out to production combined with monitoring for problems and a rapid rollback if problems are detected. The problem is that monitoring is incomplete, so occasionally bad pushes slip through the canary process unnoticed and cause serious issues.

For this reason, production engineers often suggest using ML to automatically detect bad pushes and alert and/or roll them back.

This is a specialized version of the alerting problem; you need positive labels and negative labels, labeling successful pushes with positive labels and broken pushes with negative labels. Much like with alerting, you will probably need thousands of examples of bad pushes and hundreds of thousands of examples of good pushes to differentiate real problems from noise better than a simple heuristic. The main factor that makes canarying a little less hard than general alerting is that you have a strong signal of a high-risk event when your canary starts (as opposed to continuous monitoring for general alerting) and an obvious mitigation step (roll back), but you still need a large number of correctly labeled examples to do better than heuristics. Note that if you have a bad push that you didn't notice in your labeling, because it was rolled back too fast or got blocked by something else and improperly labeled as a good push, it will mess up your data and confuse your ML model.

False-positive canary failures will halt your release (which is usually a preferable outcome to an outage). To maintain release velocity, these need to be kept to a minimum, but that will lower the sensitivity of your model.

Root Cause Analysis

Outages are difficult to troubleshoot because there are a huge number of possible root causes. Experienced engineers tend to be much faster than inexperienced engineers, showing that there is some knowledge that can be learned.

Production engineers would like to use ML to identify the most likely causes and surface information about them in an ordered fashion to the people debugging problems so that they can concentrate on what is likely. This would require classifying the set of most likely causes, and then labeling and training over enough data to rank this list of causes appropriately.

Because you need a fixed list of classes to train over for this problem, if a new type of problem shows up your model won't be able to predict it until it has trained over enough new examples. If you have a case that isn't on your list, then people may spend excessive time looking through the examples recommended by the model even though they're irrelevant. To minimize this risk, you might want to add lots of classes to handle every different possibility you can think of, but this makes the training problem harder as you need more properly labeled training data for every class of problem you want the model to be able to predict. To be able to differentiate between a list of a hundred causes, you'll probably need tens of thousands of properly labeled training examples. It will be difficult to label these examples with the correct root cause(s) without a huge number of incidents, and there is a strong risk that some of the manually determined root cause labels will be incorrect due to the complexity, making the model inaccurate. An additional complexity is that events (potential causes) sequenced in one order may be harmless (capacity taken down for updates after load has dropped), but sequenced in another order may cause a serious outage (capacity taken down for updates during peak load), and the importance of this sequencing may confuse the ML model.

A manually assembled dashboard with a list of the top *N* most common root causes, how to determine them (some of which might be automated heuristics), and related monitoring will probably be more helpful than an ML model for root cause analysis in most production systems today.

Path Forward

We do not recommend that most organizations use machine learning to manage production operations at this point in the maturity of software services and ML itself. Most systems are not large enough and would do better to focus their engineering effort and compute resources on more straightforward means of improving production operations or expanding the business by improving the product itself. Unless all of your monitoring is well curated, alerting is carefully tuned, new code releases thoroughly tested, and rollouts carefully and correctly canaried, there is no need to expend the effort on ML.

However, in the future as production deployments scale, data collection becomes easier, and ML pipelines are increasingly automated, ML will definitely be useful to a larger fraction of system operators. Here are some ways to get ready:

1. Collect your data. Figure out what data you think you might use to run production infrastructure and collect it.
2. Curate those data. Make sure that the data are part of a system that separates and, where possible, labels the data.
3. Begin to experiment with ML. Identify models that might make sense and, with the full understanding that they will not reach production any time soon, begin the process of prototyping.

Conclusion

While ML is promising for many applications, it is difficult to apply to operations today because it makes errors, it requires a large amount of high-quality training data that is hard to obtain and label correctly, and it's hard to explain the reasons behind its decisions. We've identified some areas where people commonly think ML can help in operations and what makes it difficult to use in those applications. We recommend using standard tools to improve operations first before moving forward with ML, and we suggest collecting and curating your training data as the first step to take before using ML in operations.

References

[1] https://deepmind.com/research/case-studies/alphago-the-story-so-far.

[2] https://www.seas.upenn.edu/~cis520/papers/RL_for_starcraft.pdf.

[3] https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/42542.pdf.

[4] https://www.topbots.com/chihuahua-muffin-searching-best-computer-vision-api/.

COLUMNS

Site Reliability Engineering and the Crisis/Complacency Cycle

LAURA NOLAN

Laura Nolan's background is in site reliability engineering, software engineering, distributed systems, and computer science. She wrote the "Managing Critical State" chapter in the O'Reilly Site Reliability Engineering book and was co-chair of SREcon18 Europe/Middle East/Africa. Laura is a production engineer at Slack. laura.nolan@gmail.com

This column will be published in Summer 2020, but I'm writing it in mid-March. In the past week, in a response to the spread of the new SARS-CoV-2 virus, many nations have closed down schools and imposed restrictions on travel and events. Several major technology companies are encouraging most employees to work from home. Stock markets are falling more quickly than in the first stages of the 2008 crash. Nothing is normal.

My social media feeds clearly show that SARS-CoV-2 is a source of fascination for systems engineers and SREs (site reliability engineers) because it has some characteristics of the kinds of systems problems we deal with in our work. The pandemic response is currently centered around preventing the spread of the infection, effectively an attempt to throttle admissions to intensive care in order to avoid saturating scarce medical resources. It involves gathering metrics (which are lagging and sparse due to shortage of test kits) to make analyses and projections. The mathematical analysis of the spread of the illness is very similar to the characteristics of information propagation in a dissemination gossip protocol [1], which will be familiar to anyone who has worked with Cassandra, Riak, Consul, or even BitTorrent—the major difference being that instead of modifying software parameters to adjust the propagation, we all now need to reduce our social interactions, and perhaps to partition our systems with travel restrictions.

I am not an epidemiologist, and I can't predict how this situation will unfold between now and when you read these words. Will we have endured on an international scale the kind of health crisis that northern Italy is experiencing in March, or will most nations succeed in averting the worst consequences of the pandemic, as South Korea seems to have done? If we do succeed, it's possible that many will consider the robust response to the outbreak to be an overreaction, even in light of the evidence from northern Italy and Wuhan that failure to control outbreaks leads to major public health problems.

The Job Is to Get Ahead of Problems

There is a phenomenon in operations, which I've heard called the "paradox of preparation"—an organization that is effectively managing risks and preventing problems can fail to be recognized as such. Bad outcomes aren't actually occurring, because of this preventative work, so decision-makers may come to believe that the risks are significantly lower than they actually are. Therefore, leaders may conclude that the organization that is preventing the negative events from occurring isn't an efficient use of resources anymore.

This appears to have been the fate of the White House's National Security Council Directorate for Global Health Security and Biodefense, which was set up in 2014 in response to the Ebola outbreaks in Western Africa, then shut down abruptly in 2018. It was tasked with monitoring emerging disease risks and coordinating responses and preparation. According to its former head, Beth Cameron, "The job of a White House pandemics office would have been to get ahead: to accelerate the response, empower experts, anticipate failures, and act quickly and transparently to solve problems" [2]. That is a function very much akin to what a good SRE or resilience engineering team can do within a software engineering organization.

In 2019, before the SARS-CoV-2 virus appeared, the Center for Strategic and International Studies think tank drew attention to the closure of the Directorate.

> When health crises strike—measles, MERS, Zika, dengue, Ebola, pandemic flu—and the American people grow alarmed, the U.S. government springs into action. But all too often, when the crisis fades and fear subsides, urgency morphs into complacency. Investments dry up, attention shifts, and a false sense of security takes hold. The CSIS Commission on Strengthening America's Health Security urges the U.S. government to replace the cycle of crisis and complacency that has long plagued health security preparedness with a doctrine of continuous prevention, protection, and resilience. [3]

This cycle of crisis and complacency is one we see in other kinds of organizations, including software companies—a view that reliability is only worth investing in the wake of problems, and at other times it may be deprioritized and destaffed. The last edition of this column discussed Professor Nancy Leveson's model of operations as a sociotechnical system dedicated to establishing controls over production systems in order to keep them within predefined safety constraints [4]. The crisis/complacency cycle makes it impossible to build a strong sociotechnical system that proactively manages change and emerging risks, because it means that when investment into reliability happens you have to build expertise, standards, processes, and organizations from scratch while already in crisis mode.

Against the Advice of Their Own Experts

This crisis/complacency cycle is not new, nor is it unique to either software or to pandemic prevention. The Boeing 737 Max has been in the news for most of the past year following two fatal crashes which were the consequence of design flaws in the new aircraft type. The entire 737 Max fleet was grounded in response to the accidents.

The airplane's design was certified by the US Federal Aviation Administration (FAA), a body created in 1958 to manage all aspects of safety in aviation. Air travel has become safer every decade since the FAA was set up, driven by improvements in technology and safety culture. Perhaps not coincidentally, the FAA has come under significant budgetary pressure in recent years. Partly as a result of those budgetary constraints and partly because of a shortage of relevant technical expertise, the FAA delegated much of the technical work of validating the design of the 737 Max aircraft against FAA standards to Boeing itself.

The report of the House Committee on Transportation and Infrastructure paints a clear picture of enormous pressure from Boeing's management to get the aircraft to the market as quickly as possible, at the lowest feasible cost and without any need for existing 737 pilots to take further training—regardless of any safety concerns [5]. Budgets for testing were cut, and multiple suggestions by engineers to incorporate extra alerts and indicators were rejected. Though it isn't in Boeing's commercial interest to develop an unsafe aircraft, the company's management consistently made decisions that compromised safety, contrary to the advice of their own technical experts. That they did this against the backdrop of the safest period in the history of commercial flight strongly suggests the same cycle of crisis and complacency was at work in Boeing and the FAA that led to the shutdown of the White House's pandemics office in 2018.

Disconnects between Management and Engineers

On January 28, 1986, the Space Shuttle Challenger exploded during liftoff. The accident was triggered by the failure of an O-ring seal in unusually cold weather conditions. The disaster occurred after 24 successful space shuttle launches, and these successes helped to create complacency about safety at NASA. The incident has been studied extensively, most notably by Diane Vaughan, who coined the term "normalization of deviance" to describe the process by which previously unacceptable results and practices can gradually become the norm over time [6]. Despite that phenomenon, the Rogers Commission Report on the disaster found that engineers had raised safety concerns over the design with management.

Richard Feynman, the noted physicist, was a member of the commission that investigated the Challenger accident. Feynman was particularly struck by the difference in perception of risk between the engineers who worked on the shuttle and NASA's management. The engineers mostly believed that the shuttle had a risk of catastrophic failure between 1 in 50 and 1 in 200. NASA's management claimed that the risk was 1 in 100,000. Feynman's assessment was that the engineers' estimate of the risk was far closer to the truth than management's number, which seemed based largely on wishful thinking and misunderstandings [7].

This kind of disconnect seems also to have existed at Boeing in recent years. In 2001, Boeing's executives moved from Seattle, where its engineers are located, to Chicago, and non-engineers moved into many executive roles.

> [T]he ability [was lost] to comfortably interact with an engineer who in turn feels comfortable telling you their reservations, versus calling a manager [more than] 1,500 miles away who you know has a reputation for wanting to take your pension away. It's a very different dynamic. As a recipe for disempowering engineers in particular, you couldn't come up with a better format. [8]

"Captain Hindsight Suited Up": Outcome Bias

Many of us in the software industry still remember the cautionary tale of Knight Capital, a financial firm that went bust in 2012 as a result of a bug in their trading software. As Knight Capital was an SEC (Securities and Exchange Commission) regulated company, there was an investigation and a report, which recommended that the company should have halted trading as soon as they realized there was something amiss [9].

On July 9, 2015, the New York Stock Exchange discovered a problem in their systems. They halted trading, just as the SEC said that Knight Capital ought to have done. However, as John Allspaw put it, the "clone army of Captain Hindsights suited up, ready to go" decided that the shutdown hadn't been essential and criticized the NYSE for unnecessarily halting over a "glitch" [10].

This is outcome bias, a cognitive bias that leads us to judge decisions based on their results. We can't predict the consequences of decisions perfectly at the time we make them. Many tough decisions have to be made with imperfect information—risks we can't fully quantify, information that's incomplete or missing. Sometimes, you need to make a sacrifice decision to avoid a risk of greater harm. This is likely better than simply reacting according to prevailing conditions of the crisis/complacency cycle. This closely describes the situation that the political leaders of most of the world find themselves in March 2020 with respect to SARS-CoV-2. By the time you read this, outcome bias will likely have declared their actions as overkill (if successful) or insufficient.

Risk and Rot in Sociotechnical Systems

We work in organizations made up of people, all subject to outcome bias and prone to underestimate or overestimate risks, depending on to what extent normalization of deviance has set in on our team. Executives can become far removed from the reality of life at the front line, and their appreciation of probabilities of adverse events can be strongly affected by recent outcomes.

One of the major functions of an SRE or operations team is to proactively manage risks. This kind of work covers a broad spectrum, from keeping systems patched, rotating certs and tokens, and validating backups, through to less routine things like writing runbooks and recovery tools, running disaster tests, performing production readiness reviews for new systems, and doing thorough reviews of near-miss production incidents. These are also the kinds of work that can fall by the wayside all too easily when a team is overloaded or understaffed. The eventual outcome is likely to be a crisis and the start of a new cycle of investment.

An important part of our job, therefore, is to make the value of our work visible in order to avoid the organizational rot that makes us underestimate risk and underinvest in reliability. We live in a data-driven world, but of course, we can't track the incidents that don't happen because of good preventative work. However, at times when we aren't in crisis mode, there are many other things that we can do to show how our work contributes to increasing reliability.

We can create internal SLOs for the routine jobs we do to manage risks, and set up dashboards to show whether you're meeting those SLOs or not. Write production-readiness standards that you'd like your services to meet—covering areas such as change management, monitoring and alerting, load balancing and request management, failover, and capacity planning. Track how your services meet those standards (or don't). Set up chaos engineering and game days to test how your services deal with failure, and track those results as you would postmortem action items. Load test your systems to understand how they scale, and address bottlenecks you will encounter in the next year or two. Take near-misses and surprises seriously, and track them, along with action items. All of these things help to prevent a slide into normalization of deviance as well as giving visibility into our work.

As engineers, we have a responsibility to clearly communicate about risks in our systems and the proactive work we do to reduce them. But "the fish rots from the head down": engineering leaders ultimately make critical decisions and therefore they must be acutely aware of outcome bias and the risk of disconnects in understanding of risk between front-line engineers and themselves. Most importantly, they must be mindful of the crisis/complacency cycle and maintain an appropriate continuous investment in resilience and reliability in order to avoid crisis.

References

[1] K. Birman, "The Promise, and Limitations, of Gossip Protocols": http://www.cs.cornell.edu/Projects/Quicksilver/public_pdfs/2007PromiseAndLimitations.pdf.

[2] B. Cameron, "I ran the White House pandemic office. Trump closed it," *The Washington Post,* March 13, 2020.

[3] J. S. Morrison, K. Ayotte, and J. Gerberding, "Ending the Cycle of Crisis and Complacency in U.S. Global Health Security," November 20, 2019, Center for Strategic International Studies: https://www.csis.org/analysis/ending-cycle-crisis-and-complacency-us-global-health-security.

[4] L. Nolan, "Constraints and Controls: The Sociotechnical Model of Site Reliability Engineering," *;login:*, vol. 45, no. 1 (Spring 2020), pp. 44–48.

[5] The House Committee on Transportation and Infrastructure, "Boeing 737 MAX Aircraft: Costs, Consequences, and Lessons from Its Design, Development, and Certification," March 2020: https://transportation.house.gov/imo/media/doc/TI%20Preliminary%20Investigative%20Findings%20Boeing%20737%20MAX%20March%202020.pdf.

[6] D. Vaughan, *The Challenger Launch Decision: Risky Technology, Culture, and Deviance at NASA* (University of Chicago Press, 1996).

[7] Presidential Commission on the Space Shuttle Challenger Accident, Report, 1986: https://science.ksc.nasa.gov/shuttle/missions/51-l/docs/rogers-commission/table-of-contents.html.

[8] J. Useem, "The Long-Forgotten Flight That Sent Boeing Off Course," *The Atlantic*, November 20, 2019: https://www.theatlantic.com/ideas/archive/2019/11/how-boeing-lost-its-bearings/602188/.

[9] "In the Matter of Knight Capital Americas LLC," SEC Release No. 70694, October 16, 2013: https://www.sec.gov/litigation/admin/2013/34-70694.pdf.

[10] J. Allspaw, "Hindsight and Sacrifice Decisions," March 3, 2019: https://www.adaptivecapacitylabs.com/blog/2019/03/03/hindsight-and-sacrifice-decisions/.