Chapter 29 - Dealing with Interrupts

Dealing with Interrupts

Written by Dave O'Connor
Edited by Diane Bates

"Operational load," when applied to complex systems, is the work that must be done to maintain the system in a functional state. For example, if you own a car, you (or someone you pay) always end up servicing it, putting gas in it, or doing other regular maintenance to keep it performing its function.

Any complex system is as imperfect as its creators. In managing the operational load created by these systems, remember that its creators are also imperfect machines.

Operational load, when applied to managing complex systems, takes many forms, some more obvious than others. The terminology may change, but operational load falls into three general categories: pages, tickets, and ongoing operational activities.

Pages concern production alerts and their fallout, and are triggered in response to production emergencies. They can sometimes be monotonous and recurring, requiring little thought. They can also be engaging and involve tactical in-depth thought. Pages always have an expected response time (SLO), which is sometimes measured in minutes.

Tickets concern customer requests that require you to take an action. Like pages, tickets can be either simple and boring, or require real thought. A simple ticket might request a code review for a config the team owns. A more complex ticket might entail a special or unusual request for help with a design or capacity plan. Tickets may also have an SLO, but response time is more likely measured in hours, days, or weeks.

Ongoing operational responsibilities (also known as "Kicking the can down the road" and "toil"; see Eliminating Toil) include activities like team-owned code or flag rollouts, or responses to ad hoc, time-sensitive questions from customers. While they may not have a defined SLO, these tasks can interrupt you.

Some types of operational load are easily anticipated or planned for, but much of the load is unplanned, or can interrupt someone at a nonspecific time, requiring that person to determine if the issue can wait.

Managing Operational Load

Google has several methods of managing each type of operational load at the team level.

Pages are most commonly managed by a dedicated primary on-call engineer. This is a single person who responds to pages and manages the resulting incidents or outages. The primary on-call engineer might also manage user 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 manned by a single engineer. The on-call engineer might escalate pages to another team member if a problem isn't well understood.

Typically, a secondary on-call engineer acts as a backup for the primary. The secondary engineer's duties vary. In some rotations, the secondary's only duty is to contact the primary if pages fall through. In this case, the secondary might be on another team. The secondary engineer may or may not consider themselves *on interrupts*, depending on duties.

Tickets are managed in a few different ways, depending on the SRE team: a primary on-call engineer might work on tickets while on-call, a secondary engineer might work on tickets while on-call, or a team can have a dedicated ticket person who is *not* on-call. Tickets might be randomly autodistributed among team members, or team members might be expected to service tickets ad hoc.

Ongoing operational responsibilities are also managed in varying ways. Sometimes, the on-call engineer does the work (pushes, flag flips, etc.). Alternately, each responsibility can be assigned to team members ad hoc, or an on-call engineer might pick up a lasting responsibility (i.e., a multiweek rollout or ticket) that lasts beyond their shift week.

Factors in Determining How Interrupts Are Handled

To take a step back from the mechanics of how operational load is managed, there are a number of metrics that factor into how each of these interrupts are handled. Some SRE teams at Google have found the following metrics to be useful in deciding how to manage 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)

You might notice that all of these metrics are suited to meeting the lowest possible response time, without factoring in more human costs. Trying to take stock of the human and productivity cost is difficult.

Imperfect Machines

Humans are imperfect machines. They get bored, they have processors (and sometimes UIs) that aren't very well understood, and they aren't very efficient. Recognizing the human element as "Working as Intended" and trying to work around or ameliorate how humans work could fill a much larger space than provided here; for the moment, some basic ideas might be useful in determining how interrupts should work.

Cognitive Flow State

The concept of *flow state*[143] is widely accepted and can be empirically acknowledged by pretty much everyone who works in Software Engineering, Sysadmin, SRE, or most other disciplines that require focused periods of concentration. Being in "the zone" can increase productivity, but can also increase artistic and scientific creativity. Achieving this state encourages people to actually master and improve the task or project they're working on. Being interrupted can kick you right out of this state, if the interrupt is disruptive enough. You want to maximize the amount of time spent in this state.

Cognitive flow can also apply to less creative pursuits where the skill level required is lower, and the essential elements of flow are still fulfilled (clear goals, immediate feedback, a sense of control, and the associated time distortion); examples include housework or driving.

You can get in the zone by working on low-skill, low-difficulty problems, such as playing a repetitive video game. You can just as easily get there by doing high-skill, high-difficulty problems, such as those an engineer might face. The methods of arriving at a cognitive flow state differ, but the outcome is essentially the same.

Cognitive flow state: Creative and engaged

This is the zone: someone works on a problem for a while, is aware of and comfortable with the parameters of the problem, and feels like they can fix it or solve it. The person works intently on the problem, losing track of time and ignoring interrupts as much as possible. Maximizing the amount of time a person can spend in this state is very desirable—they're going to produce creative results and do good

work by volume. They'll be happier at the job they're doing.

Unfortunately, many people in SRE-type roles spend much of their time either trying and failing to get into this mode and getting frustrated when they cannot, or never even attempting to reach this mode, instead languishing in the interrupted state.

Cognitive flow state: Angry Birds

People enjoy performing tasks they know how to do. In fact, executing such tasks is one of the clearest paths to cognitive flow. Some SREs are on-call when they reach a state of cognitive flow. It can be very fulfilling to chase down the causes of problems, work with others, and improve the overall health of the system in such a tangible way. Conversely, for most stressed-out on-call engineers, stress is caused either by pager volume, or because they're treating on-call as an interrupt. They're trying to code or work on projects while simultaneously being on-call or on full-time interrupts. These engineers exist in a state of constant interruption, or *interruptability*. This working environment is extremely stressful.

On the other hand, when a person is concentrating full-time on interrupts, *interrupts stop being interrupts*. At a very visceral level, making incremental improvements to the system, whacking tickets, and fixing problems and outages becomes a clear set of goals, boundaries, and clear feedback: you close X bugs, or you stop getting paged. All that's left is distractions. *When you're doing interrupts, your projects are a distraction*. Even though interrupts may be a satisfying use of time in the short term, in a mixed project/on-call environment, people are ultimately happier with a balance between these two types of work. The ideal balance varies from engineer to engineer. It's important to be aware that some engineers may not actually know what balance best motivates them (or might think they know, but you may disagree).

Do One Thing Well

You might be wondering about the practical implications of what you've read thus far.

The following suggestions, based on what's worked for various SRE teams that I've managed at Google, are mainly for the benefit of team managers or influencers. This document is agnostic to personal habits—people are free to manage their own time as they see fit. The concentration here is on directing the structure of how the team itself manages interrupts, so that people aren't set up for failure because of team function or structure.

Distractibility

The ways in which an engineer may be distracted and therefore prevented from achieving a state of cognitive flow are numerous. For example: consider a random SRE named Fred. Fred comes into work on Monday morning. Fred isn't on-call or on interrupts today, so Fred would clearly like to work on his projects. He grabs a coffee, sticks on his "do not disturb" headphones, and sits at his desk. Zone time, right?

Except, at any time, any of the following things might happen:

- Fred's team uses an automated ticket system to randomly assign tickets to the team. A ticket gets assigned to him, due today.
- Fred's colleague is on-call and receives a page about a component that Fred is expert in, and interrupts him to ask about it.
- A user of Fred's service raises the priority of a ticket that's been assigned to him since last week, when he was on-call.
- A flag rollout that's rolling out over 3–4 weeks and is assigned to Fred goes wrong, forcing Fred to drop everything to examine the rollout, roll back the change, and so forth.
- A user of Fred's service contacts Fred to ask a question, because Fred is such a helpful chap.

- And so on.

The end result is that even though Fred has the entire calendar day free to work on projects, he remains extremely distractible. Some of these distractions he can manage himself by closing email, turning off IM, or taking other similar measures. Some distractions are caused by policy, or by assumptions around interrupts and ongoing responsibilities.

You can claim that some level of distraction is inevitable and by design. This assumption is correct: people do hang onto bugs for which they're the primary contact, and people also build up other responsibilities and obligations. However, there are ways that a team can manage interrupt response so that more people (on average) can come into work in the morning and *feel undistractible*.

Polarizing time

In order to limit your distractibility, you should try to minimize context switches. Some interrupts are inevitable. However, 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. A 20-minute interruption while working on a project entails two context switches; realistically, this interruption results in a loss of a couple hours of truly productive work. To avoid constant occurrences of productivity loss, aim for polarized time between work styles, with each work period lasting as long as possible. Ideally, this time period is a week, but a day or even a half-day may be more practical. This strategy also fits in with the complementary concept of *make time* [Gra09].

Polarizing time means that when a person comes into work each day, they should know if they're doing *just* project work or *just* interrupts. Polarizing their time in this way means they get to concentrate for longer periods of time on the task at hand. They don't get stressed out because they're being roped into tasks that drag them away from the work they're supposed to be doing.

Seriously, Tell Me What to Do

If the general model presented in this chapter doesn't work for you, here are some specific suggestions of components you can implement piecemeal.

General suggestions

For any given class of interrupt, if the volume of interrupts is too high for one person, *add another person*. This concept most obviously applies to tickets, but can potentially apply to pages, too—the on-call can start bumping things to their secondary, or downgrading pages to tickets.

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).*

Secondary duties depend on how onerous those duties are. If the function of the secondary is to back up the primary in the case of a fallthrough, then maybe you can safely assume that the secondary can also accomplish project work. If someone other than the secondary is assigned to handling tickets, consider merging the roles. If the secondary is expected to actually help the primary in the case of high pager volume, they should do interrupt work, too.

(Aside: *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. Your future on-call engineers will thank you, and it means they're less likely to interrupt you during your precious make time).

Tickets

If you currently assign tickets randomly to victims on your team, *stop*. Doing so is extremely disrespectful of your team's time, and works completely counter to the principle of not being interruptible as much as possible.

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. Don't spread the load across the entire team. People are not machines, and you're just causing context switches that impact valuable flow time.

Ongoing responsibilities

As much as possible, define roles that let anyone on the team take up the mantle. If there's a well-defined procedure for performing and verifying pushes or flag flips, then there's no reason a person has to shepherd that change for its entire lifetime, even after they stop being on-call or on interrupts. Define a *push manager* role who can juggle pushes for the duration of their time on-call or on interrupts. Formalize the handover process—it's a small price to pay for uninterrupted make time for the people not on-call.

Be on interrupts, or don't be

Sometimes when a person isn't on interrupts, the team receives an interrupt that the person is uniquely qualified to handle. While ideally this scenario should never happen, it sometimes does. You should work to make such occurrences rare.

Sometimes people work on tickets when they're not assigned to handle tickets because it's an easy way to look busy. Such behavior isn't helpful. It means the person is less effective than they should be. They skew the numbers in terms of how manageable the ticket load is. If one person is assigned to tickets, but two or three other people also take a stab at the ticket queue, you might still have an unmanageable ticket queue even though you don't realize it.

Reducing Interrupts

Your team's interrupt load may be unmanageable if it requires too many team members to simultaneously staff interrupts at any given time. There are a number of techniques you can use to reduce your ticket load overall.

Actually analyze tickets

Lots of ticket rotations or on-call rotations function like a gauntlet. This is especially true of rotations on larger teams. If you're only on interrupts every couple of months, it's easy to run the gauntlet,[144] heave a sigh of relief, and then return to your regular duties. Your successor then does the same, and the root causes of tickets are never investigated. Rather than achieving forward movement, your team is bogged down by a succession of people getting annoyed by the same issues.

There should be a handoff for tickets, as well as for on-call work. A handoff process maintains shared state between ticket handlers as responsibility switches over. Even some basic introspection into the root causes of interrupts can provide good solutions for reducing the overall rate. *Lots* of teams conduct on-call

handoffs and page reviews. *Very few* teams do the same for tickets.

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 creates a handy deadline enforcement for the person fixing the root cause.

Respect yourself, as well as your customers

This maxim applies more to user interrupts than automated interrupts, although the principles stand for both scenarios. 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.

If your team is responsible for handling tickets or interrupts for customers, you can often use policy to make your work load more manageable. 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. Policy can be as powerful a tool as code.

For example, if you support a particularly flaky tool that doesn't have many developer resources, and a small number of needy customers use it, consider other options. Think about the value of the time you spend doing interrupts for this system, and if you're spending this time wisely. At some point, if you can't get the attention you need to fix the root cause of the problems causing interrupts, perhaps the component you're supporting isn't that important. You should consider giving the pager back, deprecating it, replacing it, or another strategy in this vein that might make sense.

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. For example, if people need to donate compute resources, prepare a code or config change or some similar step, and then instruct the customer to execute that step and send it for your review. Remember that if the customer wants a certain task to be accomplished, they should be prepared to spend some effort getting what they want.

A caveat to the preceding solutions is that you need to find a balance between respect for the customer and for yourself. 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.

[143]See Wikipedia: Flow (psychology), *http://en.wikipedia.org/wiki/Flow_(psychology)*.

[144]See *http://en.wikipedia.org/wiki/Running_the_gauntlet*.

previous

Chapter 28 - Accelerating SREs to On-Call and Beyond

next

Chapter 30 - Embedding an SRE to Recover from Operational Overload