reddit

Why can't we stop asking "what does it mean to be SRE"?

Laura de Vesine

silverrose@reddit.com

Seriously, we keep asking this

A short tour of "who are we" talks





Early SREcons had talks and panels about "what is SRE" – SREcon 2014 (the first one) had Ben Treynor-Sloss talking about what this SRE thing is and how it works, and in 2016 both the Europe and Americas programs had panels titled "What is SRE". That makes sense – obviously we wanted to establish the field and what's in scope for the conference as it gets started.

(Images from Usenix)

Over

2018



What Makes a Good SRE: Findings from the SRE Survey

Dawn Parzych

"This leads to challenges defining the role and understanding exactly what it is."



But in 2018 we were still saying that it's hard to define and understand exactly what SRE is: https://www.usenix.org/conference/srecon18europe/presentation/parzych

Image taken from speaker photo on Usenix talk page

And Over

2021



What To Do When SRE is Just a New Job Title?

Benjamin Bütikofer

"Did the System Administrator and Operations jobs disappear or have their job titles simply changed?"

In 2021 we were asking if SRE was really a "thing" or if we'd just changed the job title (and giving tips for how to make a team a "real" SRE team): https://www.usenix.org/conference/srecon21/presentation/buetikofer

Image taken from speaker photo on Usenix talk page

And Over

2022

What SRE Could Be: Systems Reliability Engineering

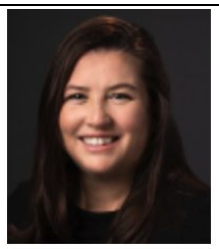
Laura Nolan

"SRE is still in its infancy, but it isn't too young to be experiencing a profound identity crisis."

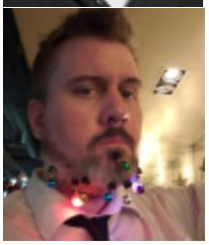
SRE in Enterprise

Steve McGhee and James Brookbank

"We review challenges that enterprises are having with adopting SRE today..."







I think in 2022 we had a real existential crisis about SRE, and so we talked about who we are, who we could be, and what we do a lot – we talked about the idea that we were having an identity crisis, how it was challenging to adopt in enterprises

https://www.usenix.org/conference/srecon22emea/presentation/nolan-sre
https://www.usenix.org/conference/srecon22emea/presentation/mcghee

Images taken from speaker photos on Usenix talk pages

And Over

2022 Still

Unified Theory of SRE

Emil Stolarsky

"We've never truly grappled with what SRE would look like at a startup..."

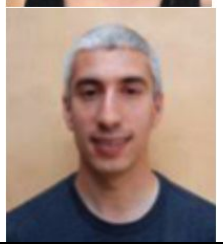
Knowledge and Power: A Sociotechnical Systems Discussion on the Future of SRE

Dr. Laura Maguire and Lorin Hochstein

"...meditations on the 'practice' and values of SRE and provocative promising new directions."







How it was challenging to adopt in startups, and where we go from here

https://www.usenix.org/conference/srecon22emea/presentation/stolarsky
https://www.usenix.org/conference/srecon22emea/presentation/maguire

Images taken from speaker photos on Usenix talk pages

And Over Again

2023 and 2024

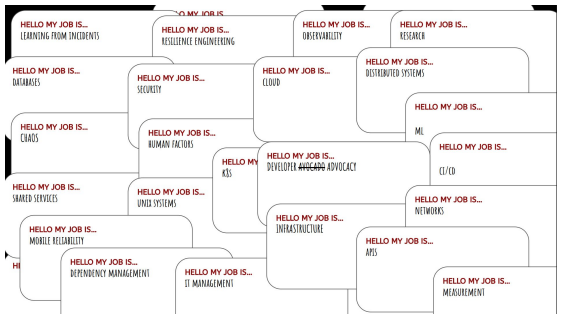
From Sysadmins to (almost) Flying Unicorns

Guillaume Hérail and Gilberto Müller

"...the definition of [SRE] team usually varies from one team to another."

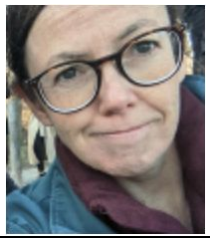
What Can You See from Here?

Tanya Reilly









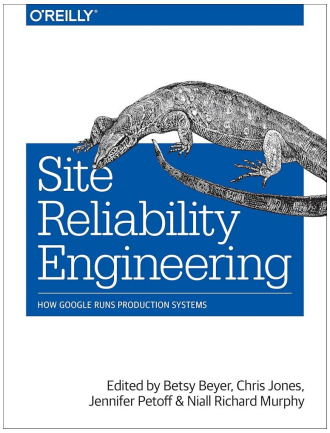
And in 2023 and 2024 – we're still talking about "what is this job"?

https://www.usenix.org/conference/srecon23emea/presentation/herail
https://www.usenix.org/conference/srecon24americas/presentation/reilly

Images taken from speaker photos on Usenix talk pages
Nametag slide copied from Tanya Reilly's talk, ibid.

A Never-ending Existential Crisis

Part of being SRE is engaging in self (re)definition





Now, you might argue with whether every single one of these is asking the same thing – I'd agree, they're really not. They all have unique things to say about exactly who we are, and how to think about that and what lessons to draw from it. But they're also all engaged in this never-ending exercise of self-definition. I would argue that this theme of talks (and there's plenty more writing and talks elsewhere out there; I was focused entirely on SREcon!) tells us that part of what it means to be SRE is engaging in a continual self definition (or maybe redefinition).

Why is that? If you'd asked me a year or two ago, I'd have told you that it was a bunch of things (several of which are referenced in those talks!): the Google SRE book tried to do too much, we're a newer field trying to find our feet, many of us have this constant awareness that we have different (often lesser) status vs. developers that we work with even though we have the same skills and are often even on the same ladder, ….

I don't think any of this is wrong, as an explanation. But over the summer I read a book…

Image credit: O'Reilly SRE book cover and AI generated

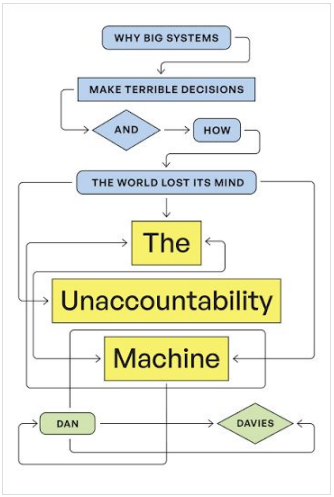
Or Maybe it's all Adaptive

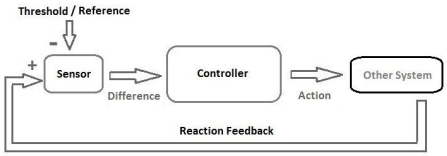


Cybernetics: a cross between management theory and systems theory

Also known as a good model of what SREs do

...And a good reason to continuously ask "who are we"





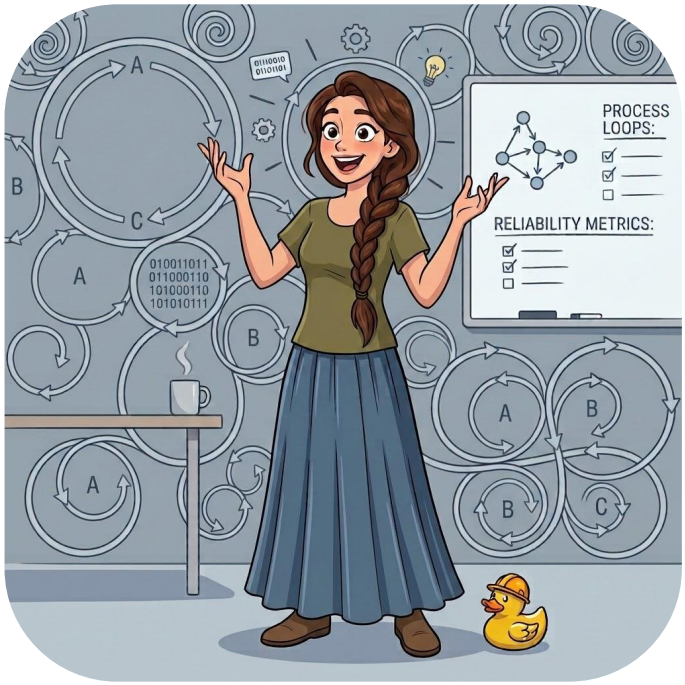


And a light bulb went off in my head. The book is called "The Unaccountability Machine". Maybe you've already read it; maybe you haven't. Honestly I only recommend it if you enjoy slightly offbeat books about the intersection of politics and systems theory so uh… I guess to this audience I heartily recommend it.

Aside from the politics though, there's a big section of the book introducing the field of "cybernetics", which I think it's fair to characterize as something of a cross of management theory and systems theory. Actually, cybernetics (though you may never have heard of it) was foundational to some of the concepts that we use to analyze systems today. But some parts of it are deeply obscure these days, the name included. I've done a bit of an informal survey, so I *think* a bunch of this is going to be new and interesting for you as well – I'd actually never run into a full characterization of cybernetics before. I found myself *really excited* by how well SRE maps to the cybernetic systems model, from the work that we do with the systems we run, to the way we organize and work on our teams (and why it makes our jobs a little oddly shaped), to who we are as a whole field – and how continuously asking "who are we" fits in the system to make us better at all three.

Images:
cover image for "The Unaccountability Machine" by Dan Davies
Cybernetic loop by Baango via Wikipedia (cc 4.0)
Lightbulb by vectorportal.com (cc 4.0)

Laura Explains (Some of) Cybernetics !

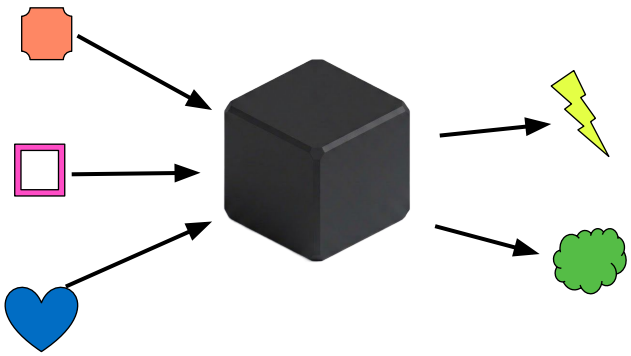


So with that, let me take you on a whirlwind tour of the cybernetics systems model, how I think it can explain some of the stranger parts of what we do as totally sensible, and some Hot Takes about AI solving incidents.

Image: AI generated

The Purpose of a System is What it Does

- Cybernetics treats systems as a black box
- We observe Inputs and outputs
- Don't assume that anything else is relevant



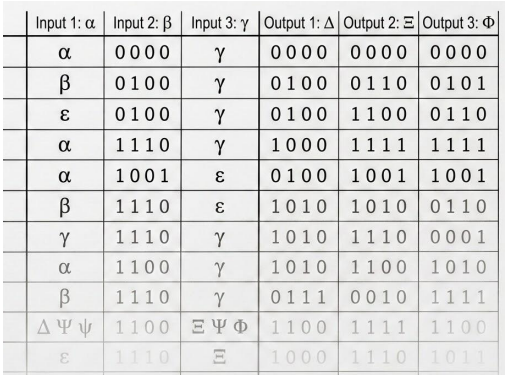
Firstly, cybernetics assumes that all systems are black boxes. We don't need to know their internals to think about and understand them – we have their inputs and outputs, and that is a description of what the system is and does *and* what it's "for". Cybernetics phrases this as "the purpose of the system is what it does".

As SREs, we interact with "black box" systems all the time – while sometimes we read the source code for breaking services, more often we look at their exterior behavior. The "black box" of our system isn't *just* the direct inputs and outputs – it's everything that you can observe from the outside about the system. So if I talk about the behavior of a piece of software as "scales to serve 100 RPS", that's still treating the system as a black box. Similarly, its error behaviors, how it responds to configuration changes, things I know about its latency – those are all elements of my black-box software system. Most of the time as an SRE interacting with software I operate, unless I'm reading the code or looking at internal traces, I'd say I'm dealing with a system as a black box.

Image AI and author-generated.

It's all about "Variety"

- Cybernetics thinks about this system surface as "variety"
  - Literally how many outputs and internal states can the system have



Input 1: α	Input 2: β	Input 3: γ	Output 1: Δ	Output 2: Ξ	Output 3: Φ
α	0000	γ	0000	0000	0000
β	0100	γ	0100	0110	0101
ε	0100	γ	0100	1100	0110
α	1110	γ	1000	1111	1111
α	1001	ε	0100	1001	1001
β	1110	ε	1010	1010	0110
γ	1110	γ	1010	1110	0001
α	1100	γ	1010	1100	1010
β	1110	γ	0111	0010	1111
Δ Ψ ψ	1100	Ξ Ψ Φ	1100	1111	1100
ε	1110	Ξ	1000	1110	1011

– expand here – dig into variety more

So, we have a system, it's a black box to us – queries go in, responses and errors come out. We want to control this system so that fewer errors come out (or so that more results come out, or whatever our operational goal with the system is). We can do this without necessarily knowing anything about *how* the internals work – all we need is to be able to observe that if we run more copies of the service, it can serve more traffic, for instance.

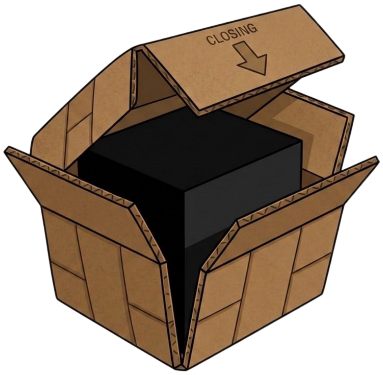
The possible outputs of a service (both its direct outputs and its errors and timing and such) are the system's "variety". You can conceptualize this as a (very long) truth table representing all of the possible input series to the service and its outputs. However many rows you need to write out that table in full is the system's variety. To write out the truth table of our software systems we'd need to include inputs over time, specific timing of arrival of inputs, configuration inputs as well as actual RPCs… I think you can immediately see that the systems we work on, while being a black box, are *quite* complex. As operators of the system, a lot of our goal is to *reduce* its variety – to make sure that we reach more of the cases where the overall output is a good result, and fewer of the ones with errors or high latency or "bad" results (ideally none of them, of course).

Image AI-generated

Requisite Variety

- We want to put boxes around our boxes to reduce the number of outcomes
- Our enclosing box needs at least as many actions as the inner box has states



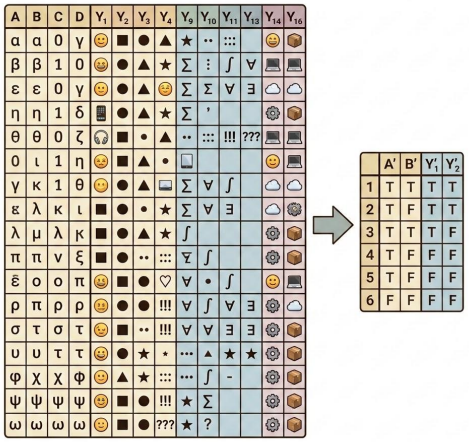


Since the system is a black box, we can't reduce its possible outputs (and its failures) by changing the system itself. Instead put another box around it, a control system, to reduce the possible outputs. That control system, in our case, is made up of both more software *and* humans. Part of why that is is that it needs to follow a thing called the *law of requisite variety.* There's math to show this, but the tl;dr is that our control system needs to have at least as much variety, at least as many states, as the system it's controlling, in order to actually constrain the internal black box system.

I think this idea of requisite variety helps us explain a few interesting things I see over and over in SRE. First, that we (routinely) wind up with very complex configurations for our autoscaling, system rollouts, traffic management, monitoring graphs, and so on. That's not a sign that we don't have enough to do: it's a sign that the systems we are operating (and controlling) with that infrastructure are themselves complex, and we need to absorb that complexity. Second, it explains why we're still oncall. For all the many things we can do with our computers, it's hard to produce software with *enough* variety to account for everything that might happen to our black-box service as a pre-planned action. What we're doing is relying on humans (via pagers) to generate the variety – the set of states – needed to constrain the service in its more unusual corners. I think this also helps us understand why many runbooks aren't actually very useful. When what we're doing with our oncall humans is specifically asking them to *absorb* the "extra" system states, the ones we *didn't* plan ahead for, we're going to need those humans to be very flexible and responsive to do that, and runbooks are static and inflexible, exactly the wrong thing for the situation.

Images AI generated

Another way we reduce variety



	A′	B′	Y'_1	Y'_2
1	T	T	T	T
2	T	F	T	T
3	T	T	T	F
4	T	F	F	F
5	T	F	F	F
6	F	F	F	F

- SLOs let us care about categories of outcome instead of every nuance
- OKRs and scoring are ways of reducing variety for the human part of the system

The other way we can reduce the variety of a system is to *ignore* a lot of the system's outputs – or more particularly, to compress them into categories where many of our possible output states are treated as much simpler – say just "good" or "bad" rather than all the richness of the actual output state. And this of course is one of the things we do with SLOs (I'll be coming back to several others in a minute). Instead of having to describe, understand, and control all the many possible outputs of the system, we can say that we care about staying to a rate of errors less than a percentage of requests, or under a particular latency. This lets our controlling system be *less* complex than if we were trying to control every individual system state at full granularity.

Our organizational systems also have variety reducers – we give people job roles and expectations with them, and score them into a set of specific boxes; we commit to OKRs (which are in descriptive language but still particular forms and have a target score) to try and reduce the output states of our teams. Variety reducers are critical to making it possible to steer a complex system without falling over under your own weight, and you can find them all over the place if you start looking.

Image AI generated

Systems are made of feedback loops

- Viable systems keep working in the face of change
  - Scale up, different traffic, new features
- Viable systems are made of feedback loops
  - Actions, control systems, and controls on controls
- We stack our control loops to handle changes in the environment



Alright, we have this black box system. We've reduced the variety we care about from it, and now we still want it to produce the good outcomes in its operational space. And we want it to keep doing this not just right this second, but *over time*. Our ideal system becomes more stable (or at least no less stable) over time, even as the system itself is changed with new updates, our users do new and different things, the underlying hardware fails, whatever might happen in the environment of the system. A system that is stable over time and in response to a variable environment is a "viable system" in cybernetics.

Viable systems are made of multiple layers of feedback loop with different jobs to keep them working well and able to respond to changes. We work with feedback loops all the time – a simple one is "measure the CPU use of each pod in the system. if it's above x%, add more pods. if it's below y%, reduce the number of pods. Measure again". This keeps the state of the underlying system at "use between x and y% of CPU per pod" – reducing the system's variety with a control loop.

In our scaling example, we can see the enclosing feedback loops when we ask… how did we set those thresholds? How do we change them as the system and our environment change? We put another control system around them! In this case, that control system looks like:

- incident reviews (do we need to set the threshold lower to scale sooner/faster and prevent outages?)
- cost and capacity reviews
- broader policies by the business about what utilization we're aiming for

And it's Turtles all the way Down

- My service (with my team operating it) is a viable system
- The larger system it's in is a viable system
- My *team* is a viable system
- The *company* my team exists in is a viable system
- ...



Viable systems and their control loops aren't one thing. They're made up of systems inside systems inside systems. any viable system contains, and is contained in, a viable system.

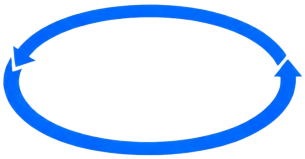
Think about it this way – an individual service (along with its control systems and humans, which we'll get to in a minute) is a viable system. It works and keeps working, and it has inputs and outputs, and it's basically stable and functional over time, assuming we're doing our jobs. But that service is also part of a larger system, which solves some larger problem – whatever our larger software actually *does*. And that larger system, whether it's my internal team webpage or google search (again along with its controllers and humans) is *also* a viable system. It's all recursion all the time.

Okay, with that in mind, let's talk through the pieces you need to actually have a viable system, stable over time in response to a changing environment. Cybernetics calls these "systems", which is accurate but I find a little confusing trying to keep distinct, so I'm going to use the term "layers".

Image AI generated

Layer 1: Operations

- The part of the system that takes inputs and generates outputs
- What the system does; it's reason and purpose



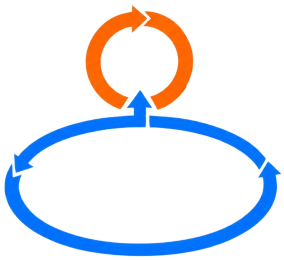
The operational layer of a system is its actual job; the thing it does. In our software systems, this is the actual service or *set* of services that work together to produce a user-facing response. It's not "the software", but rather, our software as it is running in production, as a black box. So layer one/operations is the binary and the configuration, running on the hardware and network, that gets requests and returns responses (or errors) at whatever speed it does, presumably causing the actual effect that the end user wants (we hope).

You can picture this as a simple, continual process of taking inputs and producing outputs. If we don't take the time to look closer, it's easy to think that this is the whole of the system, and the only part of it that you need – after all, this is the stuff we actually care about the system producing.

Control loops image AI generated (and really annoying to convince it to do)

Layer 2: Coordination

- How we keep systems from interfering with each other
- Lower-level systems isolation: process isolation, timeouts, …

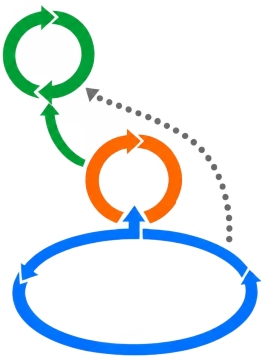


Right on top of the operations layer, we have the coordination layer. These are the things that keep multiple layer one systems from interfering with themselves and each other. When multiple parts of the operations layer might otherwise interfere with each other, the coordination layer keeps things working steadily. In a production system, this is the platform that schedules jobs and ensures resource isolation. It's also timeouts, retry limits, exponential backoff, service discovery… everything we build into our system to keep our services both able to talk to each other and not able to break each other, and continuing to serve our users instead. Coordination is the way processes work together *directly*, whether designed or not. If you have a system that routinely, say, floods downstream systems with requests when it's recovering a backlog… that's the coordination behavior you have. You might want to change it (seems like a bug), but right now, that's how system coordination is happening.

Control loops image AI generated (and really annoying to convince it to do)

Layer 3: Integration and Control

- Do the right operations, and distribute resources for it
- Request priority; monitoring
- SLOs
- Autoscaling



So, now we have a service that gets requests and returns responses (at a particular latency and with some set of errors), and a set of mechanisms that allow it to talk to other services safely. How do we… make sure that it's working? Like, all I've told you is that the service exists and is doing *something*, but not whether it's *working*. Cybernetics calls this the "integration and control" layer. This layer sets the current goal, and ensures that our system is actually accomplishing that. Generally that happens by prioritizing (limited) resources and deciding how they are spent.

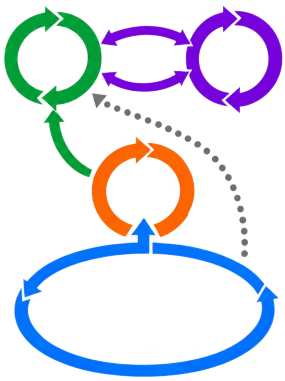
Within a software system, these are mechanisms that explicitly prioritize resources with some intention. Service and request tiering (that cause more-critical requests to be served more promptly/reliably) are a kind of layer 3 control system.

Even more than request prioritization though… this is what monitoring (and having an SLO) are all about. The SLO defines what "working" means for our system, and our monitoring tells us if it's actually accomplishing that. If it's not, then we apply corrective action. *Some* corrective action is automatic – autoscaling, for example. But a lot of the corrective actions that our systems reach for when they're not doing the right thing is that… we get paged, and then apply manual intervention to get things back on track – this is "management by exception" (and cybernetics considers it a good thing): this layer should often only get involved when something is *unusual* or wrong.

Control loops image AI generated (and really annoying to convince it to do)

Layer 4: Intelligence

- Observing and planning over time
  - Becomes a human process, not a software-only one
- Capacity planning
- Incident trends



So of course, the systems that we work on aren't just frozen in time. The environment around them – what our users want and how they are asking for it – change over time, and because it's software, the system itself changes over time as well as we try to improve us. This is the intelligence layer, and it's all about observation and forward-looking planning.

An interesting thing happens here – you may have noticed that at the last layer, we stopped talking about software and started talking about the interface of humans and software. We now find ourselves in a space where the system necessarily includes human actions. Cybernetics would say that this shows that software alone *is not* a truly viable system. And this makes sense – while a service can run on its own for a while, without human actions to repair and improve it, it will inevitably fail to respond to a changing environment and succumb to bit rot. Whether that's through other pieces of software it depends on changing, wider standards moving on, security issues, user behavior it doesn't handle, or security issues, a software system on its own *is not* self-sustaining, and is not a viable system.

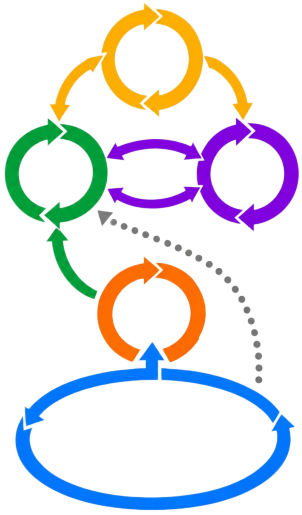
But we do long-term analysis, planning, and change as an integral part of a working piece of software. A simple example here would be capacity planning – we observe and extrapolate from prior trends to ensure we have enough hardware to serve in the future. Another example would be looking at our paging or incident trends and making a plan to fix something that is repeatedly going poorly. I mentioned we could have a coordination case where we are regularly overloading downstream systems. A plan to re-write or reconfigure the system to stop having that outage would be something

done at Layer 4 of our software system.

Control loops image AI generated (and really annoying to convince it to do)

Layer 5: Philosophy and Identity

- What's the service do? What's it *for*?
- 99% vs. 99.999% SLO?





And finally we have layer 5 of our viable system, which is the "philosophy" or "identity" layer. In our service, it's the things we know about what problems it's supposed to be solving, and what it means to be solving them well. Which brings us back to SLOs… because this is the system at play when we're *setting* an SLO. This is the system at play when we decide *what* to measure for our SLO for this service, and what threshold to set it to. What's the job of the service, and how reliable/fast/etc. does it need to be? This is also the layer at play when we say a particular service is the wrong one to be using to solve some specific problem – we understand the goals of that service (and presume that it's succeeding in those goal), and they don't match the design needs for the problem at hand. If the philosophy of our database is to serve reads from many places quickly, then perhaps it could be made to instead handle a write-heavy load… but it's not a good idea!

Control loops image AI generated (and really annoying to convince it to do)

Let's stack some turtles

- We have a team
- The team is a functional part of our software system
- Therefore, our team must also be a viable system!

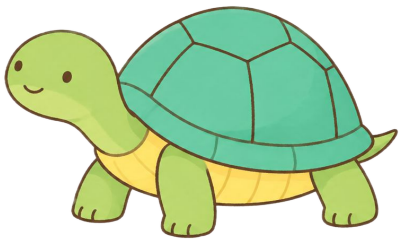


Alright, that's how to model a software service as a cybernetics viable systems model, and we found that some of its feedback and control layers were made up of SREs, who are on this *other* system called an SRE team. That's a system made up of humans – but since it's successfully functioning as a control system, it must be its own, enclosing viable system. Let's look at it!

Image: From Pixabay and published Creative Commons CC0 1.0 Universal Public Domain Dedication license. https://picryl.com/media/turtle-surprised-free-photos-d53e1f

Ops, we do ops

- Day-to-day work we do
- Page in, incident response out
- Building software



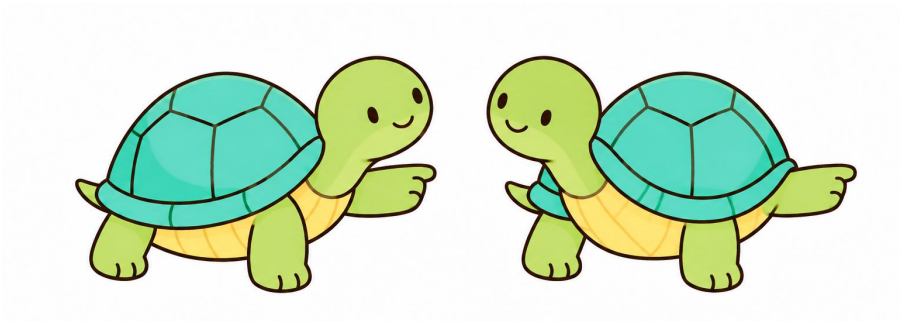
Remember, the operational layer of a system is its actual job; the thing it does (not just the thing that we as SRE would call "ops" – if we're writing new features, that's still "ops" under this model). Ops is all of the day-to-day part of our job – responding to tickets and pages yes, but also building new features in our services and platforms. It can be easy to focus on this direct output as the main (or only) job of SRE. After all, the purpose of the system is what it does – literally the operational layer is the *reason* for any system to exist. But just as our service would stop working very quickly with only an operational layer, we also know that a healthy SRE team doesn't just answer the pager (*or* just build software!). That would be an SRE team destined to stop working well very quickly, because it's missing all the feedback loops it needs to stay healthy – focused on the right work, and spending resources the right way.

I want to emphasize – an SRE team is its own viable system that produces an output consistently over time, and is also the control system of other systems. Really owning these control functions is one of the things that I think distinguishes SRE from more "traditional" ops, and I think makes us overall healthier and more effective teams, particularly over time, than a vanilla ops team.

Turtles generated by AI

Have you noticed we're glue?

- Informal team coordination
- Coffee, 1:1s, standups, standards…



Cybernetics is originally a management theory, so its original focus was largely on human systems. There, the coordination layer is usually *informal*. It's (mostly) not the part where we put work on our OKRs this next quarter or write our roadmaps. Instead, it's the part where we have lunch with each other (or an ops review, or a regular coffee 1:1, whatever), realize that the team running machine learning pipelines has the same monitoring problem as the team that owns the shared front-end for the site, and get them in a room together to figure out if we can solve the same problem once. Coordination is about making our operations efficient and effective.

Within your own team, this might be your daily standup – or your code review standards, DMs, and what you learn about what the rest of the team is working on from code reviews.

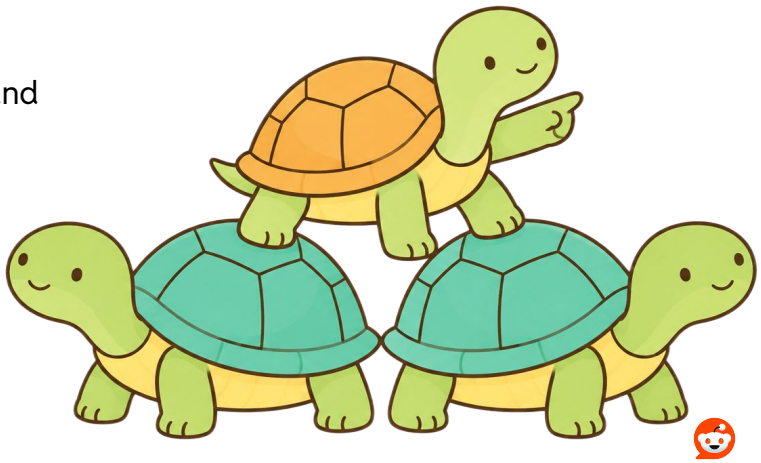
I've not done this study formally and can't find anyone who has (hit me up if you know of one!), but have you ever noticed that effective SREs tend to have larger interpersonal networks within a company than their developer peers? I think one reason for that is that the company (the system our team is a part of) is using us, intentionally or unintentionally, to provide extra organizational glue and more of the informal coordination layer that makes the whole company more efficient. Because we're solving "meta" problems that cross team boundaries a lot of the time, and doing it for many many services and teams at once, we're well-placed to do so. And whether or not we're thinking about it formally, it's natural for humans (and smart engineers) to coordinate to make the whole internal system better (and better at serving users), and we fall into that role.

So an SRE team, within a company (and while being its own viable system with its own layers), is often a coordination layer.

Turtles generated by AI

Controllers all around

- SLOs, SLOs, SLOs
- OKRs and incident command



Let's come back to some things we might do with an SLO. Some SRE teams use their SLO to direct resources between velocity (or cost) and reliability. As long as the system is within SLO, the team works on new features. If it passes below that threshold, new work stops and the team's output shifts to work improving the system's reliability. An SLO used this way (more or less the original way defined in the SRE book) is *exactly* a cybernetics control system. But an SLO that doesn't drive development priorities and resource allocations is likely still an integration and control system. It still defines a behavior, even if that behavior is "SLO crossed, now we have an incident and start paging people". What about an SLO you define but ignore? This is something I've seen developer teams do at companies that are excited about SRE practices, but maybe don't have everyone on board (or all of the actual control systems in place) – there's a big initiative to create SLOs, everyone builds some monitoring, and then there's a dashboard no one ever loads, that bit-rots into oblivion, with those long-ago SLOs on it. I think these are still control systems – just incredibly ineffective ones.

Looking around a bit, another obvious control system on our SRE team is whatever your company uses for quarterly planning. The whole point of OKRs is to commit to specific work, and *not* commit to other work, because you need to prioritize amongst limited resources. OKRs are a gimme though. Personally, I spend a moderate amount of time as an incident commander, and it's striking to me *how much* the role of an incident commander is about keeping everyone on-task: focused on the current most-important problem, not getting derailed into debugging or rabbit holes, and moving the response along – literally being the control system for the incident

handling process.

Your mileage may vary, I've found that SRE teams are great at some parts of this and not so much at others. We're great at SLOs, monitoring and alerting, and incident command – the parts of integration and control that are immediate and work more directly on the machine parts of the job. But I've *never* been on an SRE team that's any good at OKRs (I mean, maybe that all software teams, but SRE teams seem to really struggle here). I have a guess that this is because we're not being explicit enough about using OKRs to drive the *balance* of work between our priorities as a team ("we'll answer the pager less so we can write more features" is weirdly unpopular with management, but so is "we'll be answering the pager so we can't write more features").

Turtles generated by AI

Planning ahead

- Time to migrate everyone!
- Watching the tech debt
- Road maps



Most software teams do some amount of long-term planning. One thing that's interesting about SRE teams is that some of our plans are unusually likely to affect the whole company, not just our own teams. When we look at the systems around us and find a company-wide incident trend, there's a good chance that might become a company-level initiative. If we own a platform that's used broadly, and think it has too much much "tech debt", we might decide it's time to scrap the old version of the platform for v2, and get everyone to migrate.

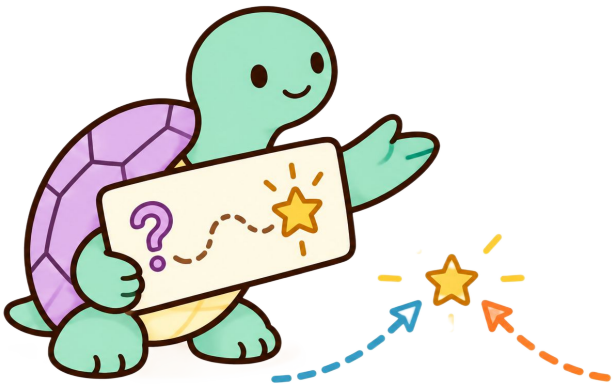
This is also pretty perceptible at an industry level – one of the things that I see happen here at SREcon (and more broadly across the industry) is a discussion and convergence on promising new systems and platforms that we all work to adopt (k8s comes to mind).

I think that SRE teams' intelligence layer is best when it's informed by data, but actively incorporates the variety we can observe as qualitative knowledge as well. This is another place where a good SRE team can have a substantial role to play in the larger company-level viable system. Understanding the reliability problems that are facing us as an organization and how we might solve those is an important piece of what we do.

Turtles generated by AI

So who are we?

- Which trend is worth following?
- What does it means to do a good job?





And here we are back at the layer that decides strategy – as a team, and given what we know from our intelligence layer, what risks do we take? Which improvements do we invest in? Do we need to prioritize reducing tech debt, or developing features?

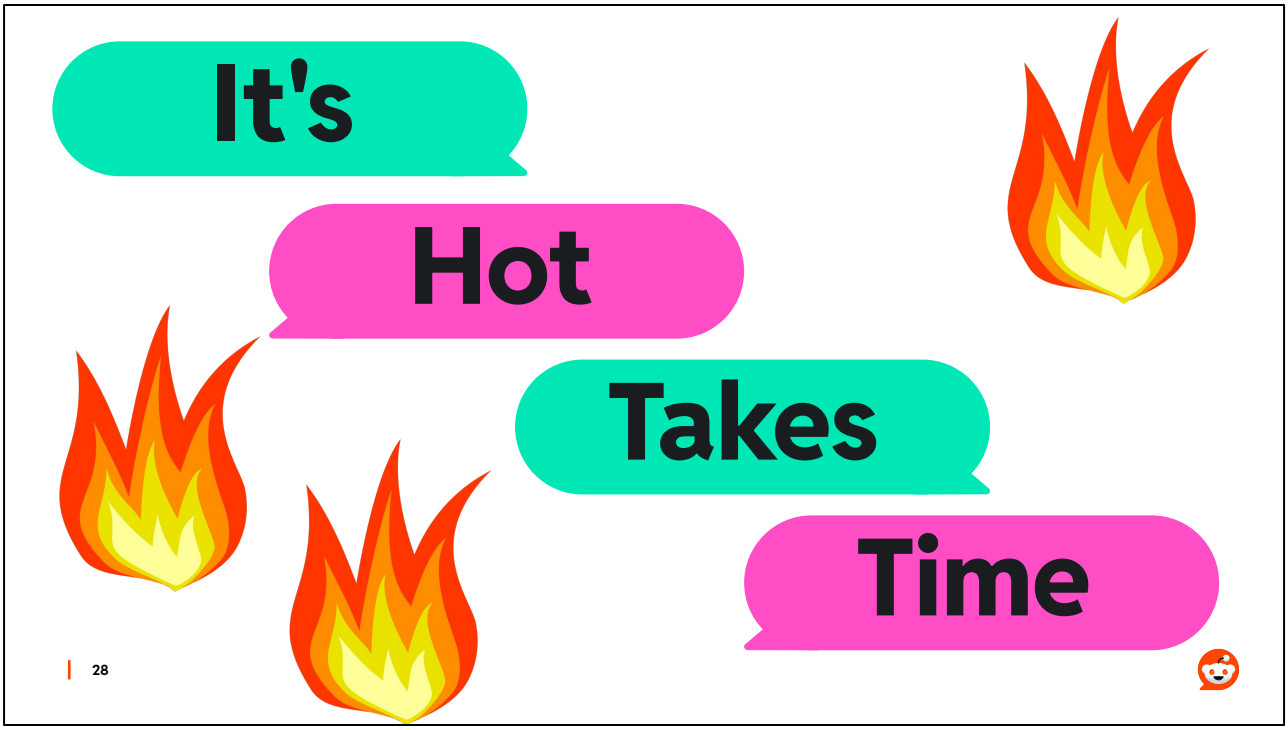
Remember I was just talking about how our SRE team might decide we think we want to move to v2 of some internal platform. Within the company, this is the decision-making about whether we ask teams to migrate, what kind of deadline we set for them, and how that priority interacts with feature work. Layer 5 is often described as having the role of deciding between resources for "now" (on immediate operations) versus longer-term investment.

For most of our developer peers, I find that the "what product do I build and why" is set at the corporate level. And when we want to push a company-level priority, that's generally true for us as well. But if I'm looking inside an SRE team, many of the ones I've been on have a much vaguer mission and ethos from our higher-ups. "Be the team that does reliability", often along with some amount of "and figure out what the best practices here are for yourselves". And this vaguer sense of identity means that we as teams do quite a bit of creating that identity for ourselves, and selling it upwards. So coming back to the question that started me off – why do we keep asking what it means to be SRE? Because it makes us effective. It keeps us calibrated to what a "good job" as an SRE team looks like; what we should be delivering and how to balance the competing demands of cost, reliability, new trends, AI adoption, …

Building that collective shared identity – which we do, in part, by having the same

conversation on the regular – gives us the power to function better on our individual teams, and across the industry.

Turtles generated by AI



And now you have an idea of one way to model a software system, and your own team, using a kind of obscure model (though one that created a lot of the concepts we still use!) from the 50s. Now that we've built the model, I've got some personal hot takes that come out of it for me.

Fire image by rawpixel.com https://www.rawpixel.com/image/6431380

Our job is so full of people

- Organizationally "between" many things
- Common to own outcomes that affect many teams
- Or systems/software chains that span many teams



I already touched on this a bit, but I want to come back to a personal observation that SREs have a lot more talking-to-people in our jobs than the average developer. Like… I'm not saying we're extroverts, but a bunch of the value of what we do is providing coordination across SWE teams. Outages rarely touch only one team's services – and it's our job to coordinate amongst them to fix the problem, both in the moment and after the fact for longer-term repairs. This certainly isn't true of every SRE team, but a commitment to owning the user experience also places us at a position of interacting between many teams, and coordinating among them. I don't think this is a bug (and that time we spend talking to people isn't wasted, within reason), but rather a feature of who we are and what we do.

Image generated by AI

Leaky Black Boxes are way Easier

- We don't *have* to manage our systems only from the outside
- Sometimes, the easiest way to remove variety is to… remove the variety



Cybernetics treats systems as black boxes – just their inputs and outputs, without needing to understand their internals. We can certainly operate our services as black boxes, and I've known plenty of SRE teams who have. This is what we're doing when we, say, have a service that is known to crash if it runs for more than 24 hours, so we create a script that auto-reboots it every 12 hours. We understand the behavior of the system, we want a different output, and we create that different output.

But an important aspect of SRE is that we can also be software engineers. We don't *have* to treat the services we run as a black box, and can instead understand and actually fix them. Rather than papering over a probable memory leak with an automatic reboot, we can change the internals of the system itself. This lets us decide to fix the underlying problem when it's the best option, but also to be realistic about the complexity of our software and the returns of fixing this with a simple solution or a more-complex and challenging one.

Image AI generated

AI incident diagnosis

"Across five frontier agents, the best RCA Accuracy is 25.3% on Medium-difficulty tasks (the realistic-input setting) and 10.0% on Hard"

- ORCA-bench: How Ready Are Language Model Agents for Oncall? arXiv:2607.28545 [cs.CL]

"However, the practical performance of these baseline agents remains very low [...], with overall perfect accuracy ranging from 3.9% to 12.5%"

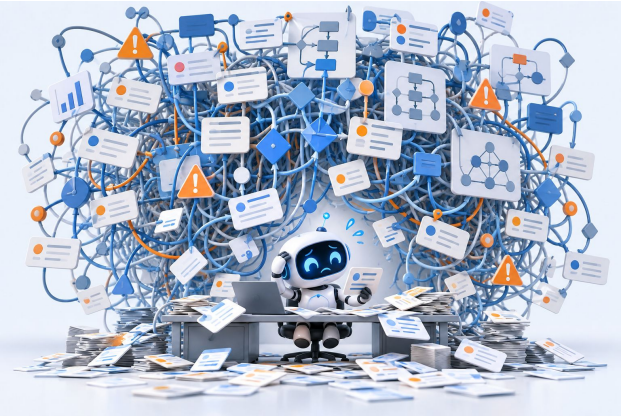
- Why Do AI Agents Systematically Fail at Cloud Root Cause Analysis? arXiv:2602.09937 [cs.AI]



So you may have seen some studies showing that AI agent accuracy for "RCA analysis" is around 25% or less. There's… honestly a lot of studies going around on the topic, and a lot of anecdata, and about a billion startups whose marketing material would like you to know that they can *definitely* sell you a product that will fix your incidents for you. I can tell you from at least my own experience that, while they can be handy once you already know what you're looking for (like "find me a PR that would generate xyz behavior that I'm seeing"), for expanding on some data you might already have, and for small/repetitive issues, they're definitely not able to go from "here's an alert for a real outage" to "and this is what to do about it". So… why are production outages harder?

"Requisite Variety"

- The control system has to have at least as many outputs dimensions as your operational system
- While still being possible to control itself



Earlier I mentioned this thing called "requisite variety". If you have a system, and you want to reduce its number of outputs (literally control your outage so you stop having it), you need your control system to have at least as much variety as the system under control. Our software systems are not just internally complex – they are constantly changing over time. While AI systems are quite complex, they do not produce as much variety (~possible outcome states) as our extremely complex and dynamic large-scale systems – and if they do, then they're impossible to control and predict *themselves*, leading them to have limited control in unusual states.

Not all outages are complicated. I've worked with plenty of teams where the vast majority of their incidents are solved by a scale-up or a rollback. But… one of our goals as an SRE team is not just to "fix" those outages, but to *prevent* them. And that means that both the system itself is now more complex (because it has things like autoscaling and gradual rollouts and canary analysis and…), *and* its outages are more complex (because we've prevented the simple ones). I'm gonna be honest with you because I promised hot-takes time. I think that papering over problems that we could fully prevent with better systems design or well-understood practices like autoscaling by instead adding an operational LLM to do a scale-up and solve our common outages is… probably not the cheapest or most reliable way to improve our system stability. And the complex problems, the hard ones we don't know how to prevent, are outside the capacity of an LLM to resolve.

Image AI generated

And "Viable Systems"

- The changes we make over time are part of the system's feedback loops



It's also the case that our outages aren't (and shouldn't be) isolated events that we fix in the moment and move on. We engage with them to understand what *exactly* went wrong in our system and to plan and prioritize short- and long- term improvements. We also look for patterns over time to improve the problems that are systemic, and not just single events. Software – LLMs included – neither plans nor has a sense of what "good" is. Those larger, strategic and forward-looking parts of SRE remain a human question, even if LLMs get substantially better at in-the-moment troubleshooting and mitigation. If that happens I think we're going to have to think quite hard about how we teach the skills for pattern-matching and long-term fixing without having individual outages to work from, but that's not a question I'm going to try to answer here and now.

Image AI generated



I hope I've managed to make something of the same light bulb go off in your head that went off in mine, and that I'm leaving you with a new toolkit of "ways to think about the work we do" that will be useful to you, or left you with some skepticism about promises to solve your whole job with a robot in the next year. Our systems are complex, and reducing that variety down to the desired outcomes isn't just a matter of building one or even two control loops on top, no matter how complicated they might be.

"Wrapping up" image courtesy of Matthew Dillon via Flickr under cc 2.0
https://www.flickr.com/photos/ruggybear/5063357354/

An existential crisis a day…





And finally, to bring us full-circle (and put a nice neat bow on my original question) – what would you say it is that you do here? Not just in your job as an SRE, but here at SREcon? And why is it that we do… whatever it is that we're here doing? Hopefully you can say now that it's not only reasonable and understandable that we engage in this regular exercise here of thinking about and re-parsing who we are and what it is that we do, it's <click> actively healthy and beneficial.

Image from the movie "office space"

… Keeps the outages away



actively healthy and beneficial. Existing as a full "viable system", with the ability to retrospect, understand a changing environment, and hold tight to (while evolving) who we are lets us actually be the teams that run healthier, more stable systems over time. A little regular chance to re-establish what we do, why and how we do it, and what this whole SRE thing is… is actually a critical part of what makes us SRE.

Image AI generated


reddit
Thank you