Are We All on the Same Page?
Let's Fix That

LUIS MINEIRO

Luis's broad background in software engineering includes experience in DevOps, system administration, networking, and more. Luis has been with Zalando since 2013, working with approximately two hundred engineering teams increasing the observability and reliability of the Zalando e-commerce platform, currently heading Site Reliability Engineering.
luis@zalando.de

Industry has defined as good practice to have as few alerts as possible, by alerting on symptoms that are associated with end-user pain rather than trying to catch every possible way that pain could be caused. Organizations with complex distributed systems that span dozens of teams can have a hard time following such practice without burning out the teams owning the client-facing services. A typical solution is to have alerts on all the layers of their distributed systems. This approach almost always leads to an excessive number of alerts and results in alert fatigue. I propose a solution to this problem by paging only the team closest to the problem.

The Age of the Monolith

Many organizations became successful running a monolith. In the age of the monolith we had single, large boxes that did everything—they handled every request. There were some minor evolutions of this basic model, namely for redundancy and availability, but that's not so relevant. What's important—monoliths were simple. They were easy to reason about and easy to monitor.

This was the time of the Ops and Dev silos. The Ops people monitored the hardware and checked whether the monolith process was up. The Devs monitored the requests and the responses.

This approach had its own share of problems, particularly as businesses grew and the approach didn't allow the business to scale further. Microservices have become the solution for those problems.

Modern Microservices

The diagram in Figure 1 is a possible representation of a typical business operation in e-commerce websites—placing an order.

Founded in 2008 in Berlin, Zalando is Europe's leading online fashion platform and connects customers, brands, and partners. It has more than 200 software delivery teams. Organizations such as Zalando can have north of 60 microservices involved in such a business operation, including some so-called legacy ones. Other organizations can actually be simpler or more complex, so mileage may vary. The relevant question is, how do we monitor and alert on this?

The industry came up with new job roles, some call them DevOps, some call them SRE, but the name is not important. We could call them Cupcake Fairies; it doesn't matter. What matters is how we monitor didn't change much, and the new roles didn't change anything. We still check whether boxes are alive, processes are responsive, and individual microservices succeed. Most times, we also check whether responses are fast enough.

When it comes to monitoring, I'd say that we're just monitoring distributed monoliths.

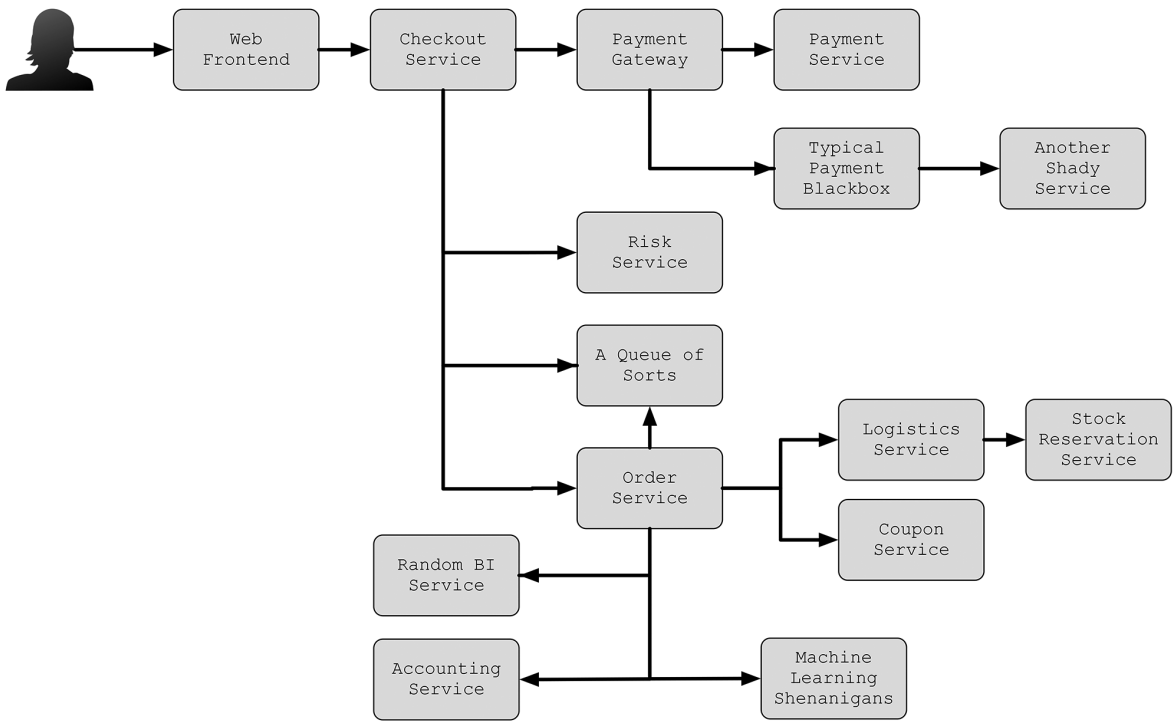


Figure 1: An example set of the microservices involved in fulfilling a customer request. Arrows show the flow through the services and also indicate dependencies.

Problem Statement

What about alerting? What happens when the Accounting Service from the example diagram in Figure 1 has an outage? What almost always happens is that dozens (or hundreds) of alerts come up, making it look like all services failed.

I call this the Christmas Tree effect. Lots of blinking lights, almost the same as Christmas except the happiness level is different, and definitely no one is getting any presents!

This approach almost always leads to an excessive number of alerts and results in alert fatigue. Only one of those teams can actually do something about it—the one operating the Accounting Service.

The alternative to this is to alert on symptoms instead. That's something the industry already accepted—in theory. How would it look if we were alerting on symptoms?

We can measure signals like latency and errors where the Web front end calls the Checkout Service. This is a good place to measure such service level indicators, where the signal-to-noise ratio is optimal and as close as possible to the customer pain.

What happens when alerting on the symptom if the Accounting Service has an outage?

The alert created based on the symptom will be triggered. This looks better. Is there anything wrong with the approach? What happens with this approach if the Payment Service has an outage? The same alert will be up. The team owning the client-facing service, and typically the owner of the alert rule, gets the paging alert *for each and every possible failure in the distributed system!*

This sort of pivoting is a serious problem that hasn't been addressed properly as far as I know. Alerting on all the layers of the distributed system is not healthy, and the alternative, alerting on symptoms, can result in bombing the team owning the client-facing service.

In a Twitter thread [1] early this year, Jacob Scott (@jhscott) brought up the question—"In a 'microservices organization' where teams own specific components/services of a distributed production system, who is responsible for triage/debugging/routing of issues that don't present with a clear owner? And how do they not hate their lives?" Charity Majors' (@mipsytipsy) reply, that I totally agree with, was "alright, this is a damn good question. and tbh i am surprised it doesn't come up more often, because it gets right to the beating heart of what makes any microservices architecture good or bad." This captures the essence of the problem. The so-called "microservices organizations" struggle to figure this out.

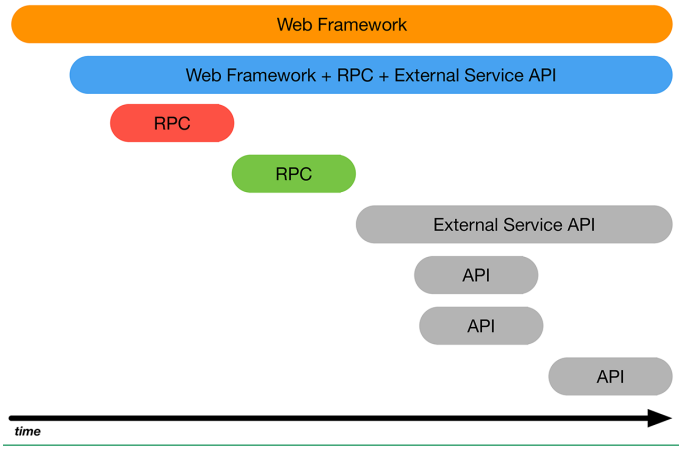


Figure 2: An example trace containing many spans from different microservices

Adaptive Paging

At Zalando we started addressing this problem with a custom alert handler that leverages the causality from tracing and OpenTracing's semantic conventions to page the team closest to the problem. We called it Adaptive Paging.

Five-Minute Introduction to OpenTracing

OpenTracing is a set of vendor-neutral APIs and a code instrumentation standard for distributed tracing. A trace tells the story of a transaction or workflow as it propagates through a distributed system. It's basically a directed acyclic graph (DAG), with a clear start and a clear end—no loops. A trace is made up of spans representing contiguous segments of work in that trace.

You can find a lot more details by checking distributed tracing's origins, namely the Dapper paper [2].

It's worth mentioning that OpenTracing has merged with another instrumentation standard—OpenCensus—resulting in OpenTelemetry. OpenTelemetry will offer backwards compatibility with existing OpenTracing integrations. The concepts and strategy for Adaptive Paging are still valid.

Spans

A Span is a named operation which records the duration, usually a remote procedure call, with optional Tags and Logs. This is probably the most important element of OpenTracing. A trace is a collection of spans.

Operations can trigger other operations and depend on their outcome. For example, *place_order* triggers and depends on all the other operations, including *update_account* in the *accounting-service*. This causality is important.

Tags

The other most relevant element from OpenTracing is Tags. A tag is a "mostly" arbitrary key-value pair, where the value can be a string, number, or bool. Every operation can have its own set of tags.

We can consider Tags as metadata that enrich the operation abstraction (the span) with additional context.

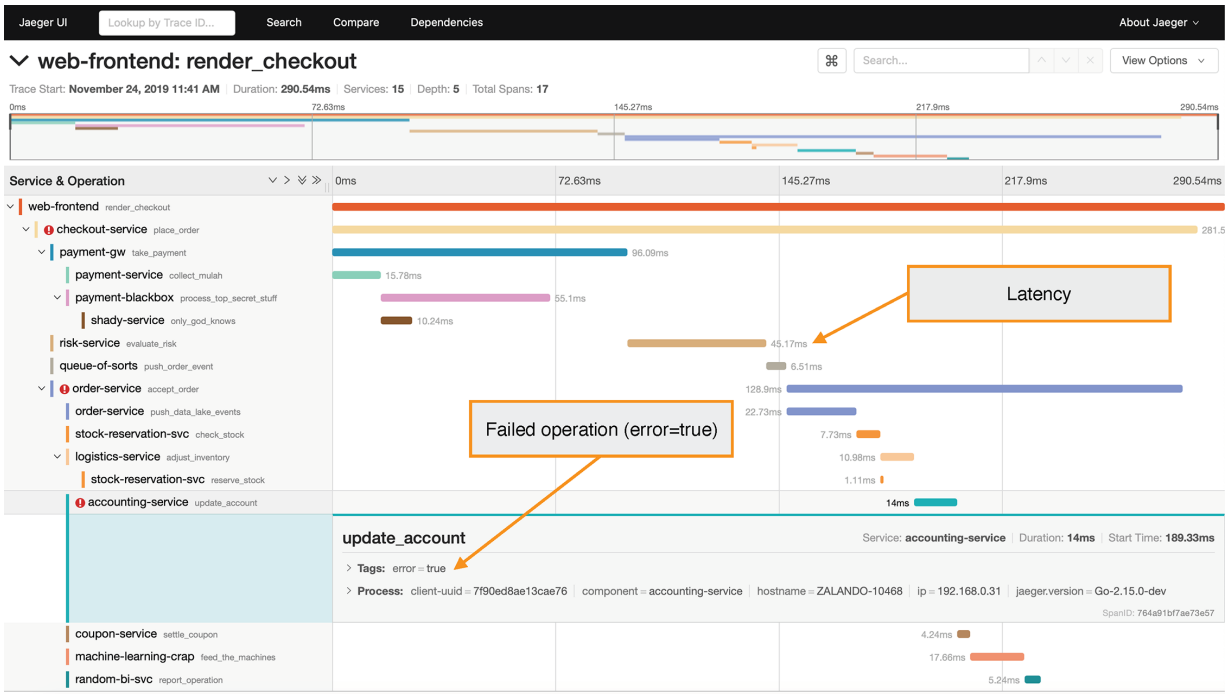


Figure 3: Screen capture of a trace in the open source tracing tool Jaeger [3]

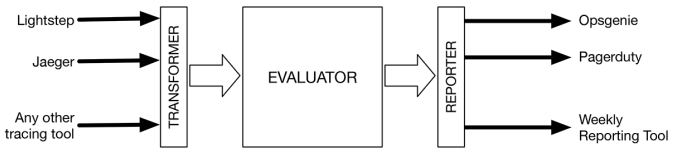


Figure 4: Adaptive Paging components and data flow

Semantic Conventions

OpenTracing's semantic conventions establish certain tag names and their meanings. The existing conventions are strong enough to set certain expectations and enable tools to apply different behaviors when analyzing the tracing data.

OpenTracing Monitoring Signals

OpenTracing can provide, implicitly, measurements for latency and throughput (number of operations over a certain time period). Through the semantic conventions it's also possible to measure errors, by checking the spans with the *error* tag set to the Boolean value *true*.

Latency, traffic, saturation, and errors are the Four Golden Signals [4]. If you can only measure four metrics of your user-facing system, focus on these four. They are great for alerting.

In this article we'll focus on one concrete signal—errors.

Alert Handler

Let's assume that an alert was configured for the *place_order* operation which has a service level objective (SLO) of 99.9 success rate. A typical way to measure this would be to query the tracing back end for spans that match a certain criteria. The keys *operation* and *component* are implicit on most tracing systems and represent the named span and the microservice itself, respectively. An expression such as *component: checkout_service && operation: place_order* represents the symptom and is where we want to measure customer pain. Different tools, open source and commercial, will usually provide different means to configure the alert itself. That's not in the scope of this article.

Adaptive Paging is an alert handler, and its architecture is broken down into three main components. The *transformer* is the actual alert handler, typically a webhook, and it's vendor specific. It's possible to have multiple alert handlers. The webhook receives alerts and converts them into symptoms. Then the symptom is passed to an evaluator, which implements the actual root-cause identification algorithm. The evaluator tries to determine the most probable root cause and generates a report. After the report is created it is made available to any reporter(s) which can deliver the page via different vendor-specific implementations or store debugging data to troubleshoot the alert handler itself.

Transformer

The transformer receives or collects vendor-specific exemplars and converts them into a vendor-agnostic data model that we called Symptoms. Exemplars are traces that should be representative of the symptoms that led to the alert being triggered. Some vendors can include exemplars as part of the alert payload. If they're not part of the payload, the transformer can query the tracing back end for exemplars that match the same criteria of the alert rule during the time of the incident.

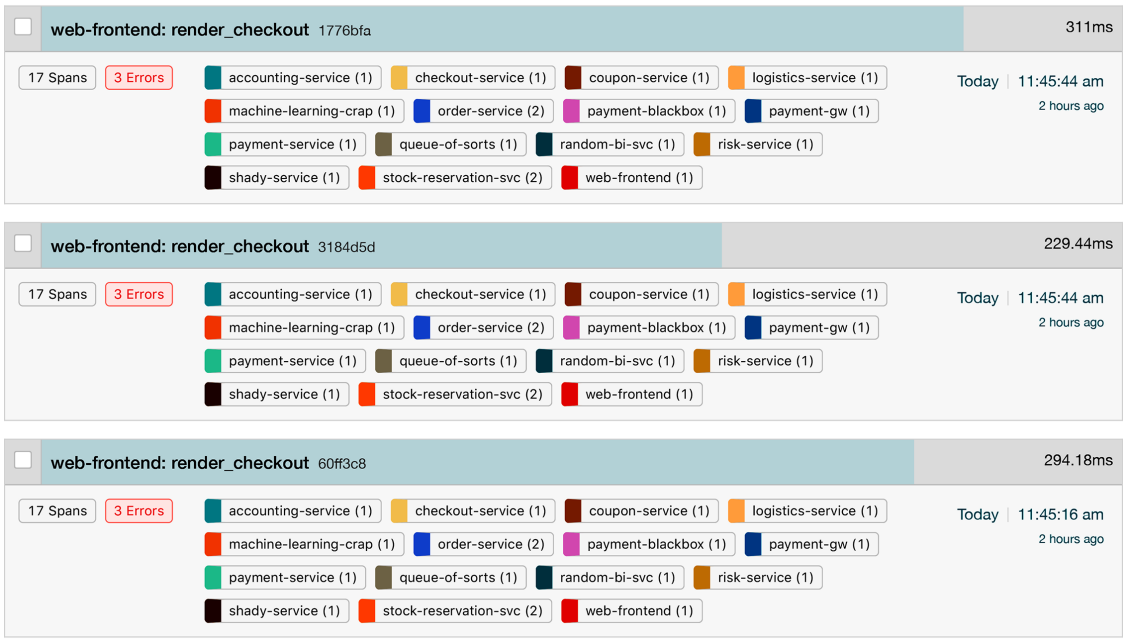


Figure 5: Collection of traces (exemplars) that contain the failed operations (*error=true*)

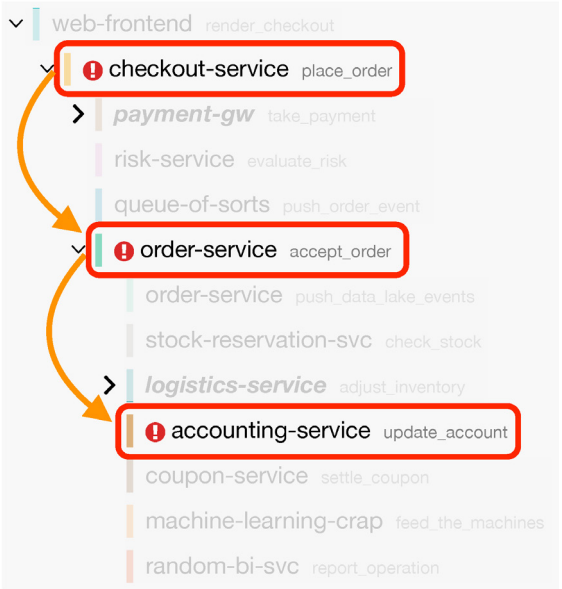


Figure 6: Probable root cause algorithm inspecting failed operations

Evaluator

The evaluation algorithm can have many different implementations. There can be different implementations for different signals—latency or errors, for example, or for any other known criteria for which a certain root-cause-identification algorithm performs better.

Example Errors Algorithm

The following example is one possible implementation to identify the probable root cause for errors. All exemplars (traces) are analyzed. Starting at the span that was defined as the signal source, each trace is inspected in a recursive way. For every child span, its tags and respective values are checked to decide which path to take.

In the example from Figure 6, none of the operations *take-payment*, *evaluate_risk,* or *push_order_event* were tagged as failed (**error=true**).

The *accept_order* operation in the order-service was tagged. The algorithm follows the path where **error=true**.

The same process is repeated. None of the operations of the *order-service*, *stock-reservation-svc*, *logistics-svc,* or the others which were triggered by *accept_order* were tagged with errors.

Only the *update_account* operation in the *accounting-service* was tagged as failed.

Without any child spans to continue the traversal, the *update_account* operation in the *accounting-service* is selected as the most probable cause of the errors.

After all exemplars are analyzed, a Report is generated.

Reporting

The Report generated by the evaluation algorithm contains information about the operation and microservice that is considered the most probable root cause. For reporters that page on-call engineers, the implementation needs to map the operation and/or service to the respective team or on-call escalation.

Putting It All Together

Going back to the original example, what happens if the Accounting Service has an outage and we're using Adaptive Paging? As you can guess, the team that operates the Accounting Service will get the single page triggered.

A similar situation would happen if any of the services involved in the "Place Order" operation breached its SLO, but the team that operates the probable root cause is the only one getting the paging alert—the one that will be able to actually do something about it—that is, no more page bombing.

Challenges

As mentioned before, the detection algorithm can adopt many different strategies. Zalando's current implementation uses a couple of heuristics that are easy to reason about.

Some of the things we had to work around when creating Adaptive Paging were:

- Multiple child spans tagged as errors: follow each path, attribute the probable cause a score. Analyze more exemplars and adjust the scores. Worst case scenario, page multiple probable causes. Paging two teams is still better than paging everyone.
- Missing instrumentation or circuit breaker open: either of these situations results in a premature evaluation of the probable root cause. We leveraged the semantic conventions to allow the caller to identify the callee, suggesting to the evaluator algorithm who to page, using the *peer.service=foo* and *span.kind=client* tag to suggest which service would be the target. This has the side effect of being a good incentive for teams to instrument their services.
- Mapping services to escalation: the service identified as probable root cause may not have a mapping to an on-call escalation. The evaluator keeps a stack of the probable causes and uses the one that is available and hopefully closest.

Finding probable causes due to latency is a challenge of its own. The strategy that we considered requires us to query the baselines for each operation and service combination, using that information to select which combination has a bigger variation at the time of the incident. This strategy can be a bit expensive, increasing the time to dispatch the paging alert.

Next Steps

Adaptive Paging was created with a multi-vendor reality in mind. Observability still has a ways to go, and some vendors are pushing the boundaries as we speak. Distributed tracing is still not a commodity, just like unit testing wasn't when it was initially introduced. No one would challenge the benefits of unit testing, and I believe no one will challenge the benefits of proper observability of distributed systems.

We've also started looking at some excellent work from LinkedIn—MonitorRank [5] from 2013, which fits nicely into Adaptive Paging; it's something we're considering as a possible improvement to the evaluator.

With Adaptive Paging we hope to contribute to improve the alerting situation, in particular paging alerts that burn out humans.

References

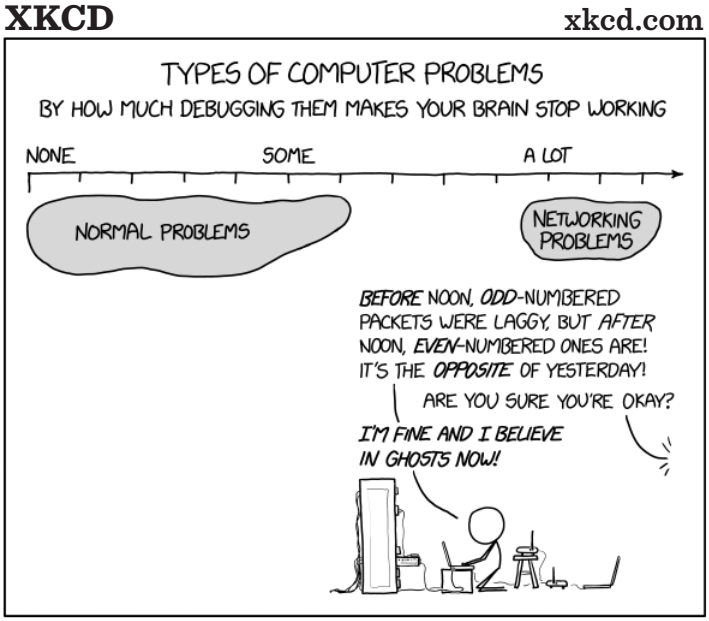
[1] Twitter thread on page bombing: https://twitter.com/mipsytipsy/status/1120911207903268864.

[2] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, C. Shan-bhag, "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure": https://ai.google/research/pubs/pub36356.

[3] Jaeger: https://www.jaegertracing.io/.

[4] Four Golden Signals: https://landing.google.com/sre/sre-book/chapters/monitoring-distributed-systems/.

[5] M. Kim, R. Sumbaly, S. Shah, "Root Cause Detection in a Service-Oriented Architecture," SIGMETRICS '13: http://infolab.stanford.edu/~mykim/pub/SIGMETRICS13-Monitoring.pdf.



Getting Things Done

TODD PALINO

Todd Palino is a Senior Staff Engineer in Site Reliability at LinkedIn on the Capacity Engineering team, creating a framework for application capacity measurement, management, and change intelligence. Prior to that, he was responsible for architecture, day-to-day operations, and tools development for one of the largest Apache Kafka deployments. He is a frequent speaker at SREcon, as well as other venues, on the topic of SRE culture and best practices, and is the co-author of *Kafka: The Definitive Guide*, available from O'Reilly Media. Out of the office, you can find Todd out on the trails, training for the next marathon.
tpalino@gmail.com

Two years ago, I found myself in a bad place, both at work and at home: overworking, ignoring my family, and angry all the time. It took months to understand the problem: I had no idea what work needed to be done. I could only focus on whatever was right in front of me, screaming for my attention. What I needed was a list of the work that I had committed to, that I trusted to be reviewed and complete, presented in a way that made it easy for me to pick the right work to do. I accomplished this with "Getting Things Done"—a process for handling work in a predictable and trusted way.

I lacked organization, which doesn't mean planning your day to the minute—as an SRE I live by the adage that no plan survives contact with the enemy. Not having organization meant I lacked the ability to respond appropriately to new work and ideas with a clear and creative mind. Martial artists practice "mind like water." Chefs have their "mise en place." For engineers, we have "inbox zero." You may call it a fantasy in an interrupt-driven world, but that only reinforces the need to have the planned work neatly maintained.

Most problems with inbox zero come from setting ourselves up to fail. We make a list of things to do today, leading to frustration when the inevitable interrupt happens and ruins our plans. We treat email inboxes as to-do lists, forcing us to continually re-read messages and decide each time what the next action is. Worse, we fail to fully catalog our work, both in the office and at home, and don't set aside the time needed for regular maintenance. When our partial attempts fail, we throw up our hands and declare organization to be an impossible task.

Enter GTD

There are many systems available for personal organization. For the last decade, I've used a system called "Getting Things Done" (GTD for short). Developed by David Allen, and documented in his book of the same name, the concepts have remained the same over the years, even as technology has changed. This is because it's not prescriptive regarding the tools that you use for organization. The process is described, with the characteristics that your trusted system must have, without placing bounds on implementation. It does not require specific software, or even any software at all. Last year, I was using a paper notebook.

You may not be sure what a trusted system is, but you're already using one: your calendar. We recognize that our brains are bad at remembering meetings, events, and the details for them, so we offload them into a calendar. Regardless of the calendar tool you use, when you get an invitation it goes into your calendar. You note whom you are meeting with, when and where the meeting is, and some details on the topic. Once you've done that your brain can let go of the information. This happens because you're consistent about using your calendar—you're checking it at appropriate times, or you trust that a notification will alert you just in time for a meeting.

This is the essence of a trusted system: a list of all of your commitments, which your brain trusts to be complete and regularly reviewed. We have to do this because brains do not organize information in a way that is conducive to getting work done efficiently. It believes that

everything is important, all the time, so it continually cycles through your to-do list. It frequently drops items. It interrupts you randomly with information that you can't use at that time. In order to fix this, we need to make something other than our brain responsible for handling this information.

The tool is just one component—a way to store and present information. What makes it trusted is the process around how you use that tool. GTD's process is made up of five core steps:

- Capture
- Organize
- Clarify
- Reflect
- Engage

Capture

In order to organize work, you first need to collect the pieces of information that prompt us to create it. This is the essence of capture: create a habit of writing everything down. The goal is to only ever have a thought about something to do once—as soon as that happens, you write it down and it enters your trusted system by going into an inbox.

We have several kinds of inboxes. Email is just one type, and you probably have more than one account. Other inboxes include a notes app on your phone, your physical mailbox, and your pocket. It's just a place where you collect stuff that you need to do something with later. Know where all those inboxes are, but have the fewest possible. Most critical is to make sure you always have a way to make a note, paper or electronic, wherever you are.

As soon as you have an idea, write it down. This could be as trivial as "I'm getting low on milk" or as ambitious as "I'd really like to run a marathon." Treat this like brainstorming: don't filter. Capture all ideas, big or small, and only the idea: you don't need to figure out what the next step is, or even whether or not it's truly something that requires action. Capture is about getting it out of your head so you can continue with what you were doing with a clear head.

Organize

Before we discuss how to process everything we've captured, we should have somewhere to store our work. Like our calendar, this needs to be convenient to refer to wherever we are: in the office, running errands, or at home. Most will choose software for this, with far too many options to cover here. Two of my favorites that are tailored for GTD are OmniFocus and NirvanaHQ. Let's talk instead about what we're going to store in this system.

Actions are things that you can actually do. They are a single, discrete step: for example, a phone call. "Make a phone call and email a summary" is at least two actions. This is like a database transaction: we have to do the action all at once, or we roll back and start over. Our actions will not only have a clear statement of the work to do, they will also have a context. This is where you have to be, or what tool you need, in order to complete the action. Phone calls need a phone, so a good context is "Phone." Looking at a website requires "Internet." Locations can be contexts: there are things that can only be done at "Home," like organizing your spice drawer. People can be contexts, which is helpful for tracking delegated actions or agenda items for your next one-on-one meeting.

What's not needed are due dates or priorities. Priorities are a losing proposition, as you'll constantly waste time re-prioritizing work every time something new arrives. The context will help us filter down the number of actions available to us at any point in time, which will make it easy to see the important ones. As far as due dates are concerned, if something is time sensitive, think about whether or not it should be on your calendar instead. Doing this makes sure that the work gets completed before it is due.

The other concept is a project, defined as a desired result that requires more than one action. Examples might be "August vacation" or "Publish a book." It is a logical container for actions that accomplish a single goal. It's important to have these containers because a project is also a placeholder. When you complete the next action for a project, you need something to continue tracking that project and prompt you to define the next action.

Our organizational system comprises several lists:

- Next actions, preferably able to be organized by context
- Projects (a simple list is sufficient here because the actions will be on the previous list)
- A "Waiting For" list of all the actions we have delegated
- A "Someday/Maybe" list of the projects we might want to do later

Time-sensitive items should go on your calendar, which may be a separate trusted system, and reference items will be stored separately. This keeps your system reasonably sized, so you can carry it with you all the time. This is critical, because you have to be able to refer to it when you need to know what work you should be doing.

Clarify

Let's get back to all of the stuff that we captured. It's time to process it. Set aside the time on your calendar for this. I find that doing this any less than every weekday (except vacations) makes me anxious. I also triage my email inboxes throughout the day as I know there will be interrupts, like last-minute meeting requests. You may have certain inboxes that are processed less frequently, which is OK as long as you are consistent.

Clarify is a process of taking each thing in our inboxes and asking the question, "What is this?" The rule is that you will go through your inbox in order, one item at a time, and nothing goes back in. When you're done, your inboxes will be empty.

Select a single item or email and ask the first question: is this something that requires you to take an action? If not, it is one of three things:

1. **Reference.** Something you need to know, or refer back to later, such as a manual or other document. Reference items need to be stored, and there are many ways to do this: filing cabinets, flash drives, or bookshelves. Like inboxes, minimize the places you store reference and make items easy to find when you're looking for them.
2. **"Someday/Maybe."** Ideas you're not ready to commit to yet. For example, you might have "Run a marathon" or "Summit Mount Everest": maybe soon, but not today. This list of ideas will prompt you to think about them later on, to start when you're ready.
3. **Trash.** If it's not actionable, and not one of the above, throw it out. This might make you uncomfortable. Take that opportunity to evaluate your decision about whether or not it's actionable. If you can't throw it away, it probably represents something you need to do.

For actionable items, determine what the very next action is to move towards completion. Let's think about a couple examples from our day to day:

"Schedule one-on-one meeting." This is a single action: we need to send an invite for the meeting.

"Fix buffer overflow bug." This is not one action: we need to write the code to fix the bug, open and wait for a review, commit the fix, and deploy it. This is a project. We will add "Fix buffer overflow bug" to our "Projects" list. We also need the very next action to take, which is "Write the code to fix the bug."

Now ask how long the action will take to complete. If the answer is two minutes or less, do it right now. It will take less time than it will to track it. For scheduling a meeting, do that now because it will be quick.

Writing code is going to take longer. Actions like this are handled in two ways:

1. **Delegate.** Someone else will do it. If we ask a teammate to write the code, we'll add, "Alice—Code fix for overflow bug" to a "Waiting For" list. This tracks the action and who has it. When we review later, we might need to remind Alice about the work.
2. **Defer.** We will do it. Actions for a specific time can go on our calendar at the time it needs to be done. Otherwise, add it to our "Next Actions" list.

Reflect

It's not enough to put all of these projects and actions into a system, we also need to review that system with a consistent cadence. This is not the same as actually doing the work that we have defined—we're going to talk about that when we get to Engage. Reflect helps ensure that the system represents the totality of the work we need to do, as far as we are aware. This is the step that soothes the brain. When you understand, subconsciously, that anything in the system is going to have your eyes on it in some fixed and recurring time frame, only then will your brain be willing to let it go and trust the system. You know that you'll come back to it at the appropriate time.

How often do you need to reflect? It depends on what makes you comfortable, but a good start is to schedule a weekly review every Friday for two hours. This goes on your calendar because it's important to guard the time. Do not be afraid, or think it is selfish, to reserve time for yourself on your calendar. By doing so, you will make yourself more productive overall.

Routine and habit is the name of the game when it comes to GTD, and the weekly review is no different. Start with clearing your head: capture any thoughts in your head that are bouncing around, and process your inboxes to zero. Then you're going to move into reviewing your entire trusted system to make sure everything is current:

- Look at "Next Actions" and check completed actions or capture new ones.
- Review last week's calendar to make sure you captured action items.
- Review next week's calendar to surface actions to prepare for it.
- Check your "Waiting For" list, sending any needed reminders.
- Review your "Projects" list, making sure each has a next action.
- Review your "Someday/Maybe" list, and pull anything to start into the "Projects" list.

After you finish the weekly review, you're going to feel like you really have your life together. This is one of the reasons I like to do it on Friday afternoon: it lets me go into the weekend knowing that I'm fully organized, and I can set work behind me and be present with family and friends.

Engage

So far we've talked about organizing things, not actually doing them, and there's a good reason for that. When your work is well organized, it's easy to select the right thing to do at any point in time and get it done. You're going to be working from your "Next Actions" list, because this represents all of the things you can do right now without waiting for something else. With the previous four steps in place, we are comfortable that the next actions list represents the totality of the work that we are aware of. We can

quickly narrow down what we can do right now on the "Next Actions" list by four criteria:

- Context: Filter out actions that don't match the contexts available to you.
- Time: If you have 15 minutes available, you can't do an action that will take longer.
- Energy: At the end of the day, you might only have the brain-power to read an article.
- Priority: With actions filtered down, it's easy to pick out the highest value one.

What happens when you're interrupted with an alert or some other interruption? First, don't get sucked into automatically doing work as it appears. Knowing your planned work helps with the decision on whether the new item is a higher priority. If it is, set aside your planned work and focus on the new work. Your trusted system will be there when you finish, and you don't need to worry about keeping track of where you were. Just pick up the next action that fits based on the four criteria.

Organizing your work will result in fewer of these interruptions, as well. Many of them exist because we didn't know what our commitments were. We forgot about that bug fix we meant to do. We buried an email that asked us to review a document by a certain date. Prioritizing proactive work will reduce the amount of reactive work required.

Next Actions

Organization is a project, and here are some next actions you can take to free up your brain to do what it's good at: being creative and solving problems.

1. Borrow or purchase a copy of *Getting Things Done* by David Allen.
2. Read the book.
3. Select GTD software (or other tool) to implement your system.
4. Schedule time with yourself for your first pass through Clarify and Organize.
5. Schedule time with yourself for a weekly Reflect session.
6. Schedule time with yourself for a daily Clarify session.

What I have presented here is an overview: there is more to be gained from reading through David Allen's book. You'll gain a deeper understanding of how to work with the GTD concepts as well as an introduction to other topics, such as horizons of focus—how to work with long-term planning and frame the question of what you want to be when you grow up.

The first time you work through processing inboxes, it's going to take a lot of time. You'll need to look at how many emails, and other pieces of paper, you have and make a decision about how much time you need. Don't shortchange yourself—time spent here will return to you 10 times over. Break it down into multiple sessions if needed, but don't leave too much time between them.

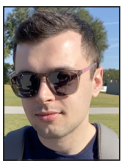
Finally, remember that organization is a habit that you need to build. Getting Things Done provides a structured process for managing our work, but it only works if you follow it consistently. It will take time to remember to capture every idea, and you will need to be diligent about guarding your time for both Clarify and Reflect at first. The feeling that you get after clearing out your inboxes and reviewing your trusted system will ensure that the habits, once established, will be hard to break.

It's an SLO World

What Theme Parks Can Teach Us about User-First Reliability

JAIME WOO AND EMIL STOLARSKY

Jaime Woo is an award-nominated writer and has been published in the *Globe and Mail*, *Financial Post*, *Hazlitt*, and *The Advocate*. He spent three years as a molecular biologist before working at DigitalOcean, Riot, and Shopify, where he launched the engineering communications function. He's spoken at SXSW, IA Summit, SREcon Americas, and SREcon EMEA, and was a guest lecturer at the University of Toronto's Rotman School of Business. He is co-founder of Incident Labs and is co-authoring the forthcoming book *SRE for Mere Mortals*.
jaime@incidentlabs.io



Emil Stolarsky is a Site Reliability Engineer who previously worked on caching, performance, and disaster recovery at Shopify and the internal Kubernetes platform at DigitalOcean. He has spoken at Strange Loop, Velocity, and RailsConf, was a program co-chair for SREcon19 EMEA, and is a program co-chair for SREcon20 Americas West. He has guested on the podcasts InfoQ and Software Engineering Daily, and contributed a chapter to the O'Reilly book *Seeking SRE*. He is co-founder of Incident Labs and is co-authoring the forthcoming book *SRE for Mere Mortals*.
emil@incidentlabs.io

In an always-on world, predictable reliability is paramount. Service level indicators (SLIs) and objectives (SLOs) are cornerstones in site reliability engineering (SRE) for purposeful reliability. SLIs are chosen measurements that act as signals for achieving your reliability goals; SLOs are the targets for SLIs. User-first SLIs and SLOs are the gold standard, and we use the concept of theme parks, those paragons of complex systems optimizing for user happiness, to demonstrate examples of strong SLIs and SLOs in contrast to useless ones.

The massive, iconic theme parks of Orlando, Florida, are impressive for children—the rides, character actors, and sights synthesize into magical, larger-than-life playgrounds. It was surprising then to realize how much more impressive the spaces become when revisited as adults. The infrastructure that manages hundreds of thousands of visitors daily, the attention to detail across the "lands"—even in places where people might not immediately notice—evoke awe and appreciation for the levels of planning and effort.

The lessons for site reliability engineers from theme parks are not immediately obvious, until you realize that SLOs are rooted in asking what level of service must be provided to keep users happy. And where else could you glean lessons about how to engineer for happiness than at the so-called happiest place(s) on earth?

Useful SLIs

Let's begin with how to find a useful SLI. A useful SLI must contain the following four parameters:

- Relate to the experience and/or satisfaction of your user
- Use a measurable quantity related to your service level
- Be as specific as it can be
- Provide enough information to be actionable

Similar to the massive infrastructure that is behind what appears as simple user-facing experiences, underneath the colorful, playful facades of theme parks are subterranean levels where workers manage the infrastructural and logistical components of the park, including electrical operations, transporting character actors, waste removal, deliveries, and food service [1].

Visitors rarely think (or even know) about these hidden parts—and that's the preference of theme parks so as not to ruin the illusion. The only thing that matters is the experience visitors paid to have, and everything that is out of view exists only in support of that experience.

Take waste removal: should trash begin to pile up around the park, visitors would complain about seeing garbage on the park grounds, rather than, say, faulty waste removal mechanisms 20 feet below them. And, theoretically, those mechanisms could break and guests wouldn't notice whether staff cleared the paths of trash often enough. So the amount of garbage on the floor is a stronger SLI than waste removal machinery uptime.

That doesn't mean ignore everything internal: instead, we acknowledge that something can be important yet not necessarily urgent. That ambiguity around urgency highlights the disadvantage in using such metrics as guidance for reliability: because it's subjective, it's more difficult to gauge reasonable boundaries around allowable downtime and, therefore, to create meaningful error budgets to justify any downtime. Anything that users interact with directly affects their ability to do what they need to do, and thus prioritizing work is clearer.

For example, take a database service with multiple replicas. It might be tempting to use uptime as an SLI, but there are many scenarios where uptime gets dinged but customers don't feel the impact. For instance, if a single replica goes down, traffic won't be affected. Instead, a more effective SLI would be to track read-query success rates, which are necessary for customer requests to be successful.

From SLIs to SLOs

Upon determining SLIs, you have to assess the right target SLOs. From our theme park example, we've figured out that guests would be unhappy with trash everywhere. Now, we want to know what their threshold is, based upon their needs and expectations.

With small piles of garbage everywhere, the park technically remains operational, but it would be a poor experience for guests, potentially discouraging them from returning or even asking for refunds. On the other hand, ensuring no piece of trash stays on the ground for longer than a few minutes would be an excessive waste of resources. How then to choose the right level?

Luckily, engineers need not—and should not—do this alone. Different business units across the organization will have their own insights into users, and when site reliability engineers work with teams like support, engineering, and product and bring those insights together, you're likelier to have meaningful SLOs. At Disney World, you're unlikely to see trash on the ground for longer than 15 minutes, as that's the interval, in crucial locations, at which trash in bins get sucked into an underground automatic vacuum collection (AVAC) system and transported away.

With our example SLI of read-query success rates, after discussing with other business units, we may learn that users typically notice degradation in the service when fewer than 95% of read-queries succeed over a period of 30 minutes. Waking up engineers the moment any read-query fails would be premature, but we could set a slightly more stringent internal SLO that once read-queries drop below a 97% success rate, alerts get sent out.

What Is the Experience?

With the need to be user-focused firmly established, we can move from what users see to what users experience. The distinction between the two is that one measures what users interact with, and then the second looks at those interactions and translates them into their meaning. In the field of UX, they understand the distinction: "While you cannot directly *design* a person's experience of a product, you *can* take steps to ensure that their experience is a positive one by employing a user-centered design process," writes Matt Rintoul, experience design director of creative agency Say Yeah! [2].

At this point, we should make a vital distinction: who your users are matters. For this article, we focus on human users rather than programs that use your service (although users can be both, depending on which part of the service each touches).

Thinking about how different users interact with your system is a useful exercise. The response times from programs are more reliable and faster than with humans. You can also tell a program to attempt a request again in five minutes. Unlike machines, humans perceive things relatively, something we'll return to later in this piece. This matters because treating humans as rational actors, as economists do, can simplify things, but you need to be careful it doesn't oversimplify. Context matters.

What Constitutes a Satisfactory Experience?

Rarely is a service entirely down. Instead, individual components may lag or fail, and even with some parts of the experience deprecated there may be no change in a user's core experience. At a theme park, the food stands, for example, could be out of service, and the park could still run. If the restrooms all failed, however, it'd be a different story.

For a technical example, on a video-streaming service, there are many components to the total experience, from searching content, user-curated lists, viewing history, and playing content. Each component can be mapped to see if it is running or not, and this matters because the components are weighted related to user satisfaction: if customers can still continue watching a film they were in the middle of then they will be happy even if they cannot amend their list of media to watch.

Specificity matters because when outages occur, or decisions around what work should be prioritized, you make best use of your limited resources by understanding which parts of the service matter most to customers. You can then also manage the number of things being tracked in dashboards to prevent information overload. An engineer needs to weigh the tradeoff of adding another SLI to monitor against the level of dissatisfaction users will have if it goes down.

Timing Matters

Just as components of your service are relative, with differing weights, this is also true for time: not every minute is the same. If your users don't notice an outage, should it count toward

your error budget? At theme parks, for instance, electronic gates require fingerprint identification for entry. If this system went down an hour before the park closed, while that's not ideal there's also the nontrivial question of who was impacted?

This isn't permission to ignore outages that happen during the off hours. You still want to know how often your service is going down, which provides a better way to understand the behavior of your system. But does your service truly need to be up 24/7? Are there periods when the service is lightly used? It isn't zero impact, but it has less impact, so do your metrics reflect this? Importantly, is a low-impact event worth the human capital of waking a team at three in the morning?

As an example, a food delivery service that solely works with restaurants on the East Coast of the United States: most restaurants do not operate between two to six in the morning, and perhaps the service has data showing that orders drop off after 10 p.m. and only revive at 10 a.m. for lunch orders. An SRE team could decide that alerting overnight for low-severity incidents isn't worth sleep-deprived and grumpy engineers and instead send pages in the morning.

Users Have a Multitude of Experiences

Rarely are users a homogeneous monolith. Instead, they are heterogeneous, each with their own (albeit, potentially overlapping) needs. In UX, the practice of creating personas acknowledges that users have different perspectives and rationales. When considering SLIs and SLOs, we should avoid blanket aggregation of users for the sake of simplicity.

Returning to the theme parks of Orlando, think about the different types of visitors: parents and guardians, children, aunts, uncles, grandparents, and adults without children. They speak different languages. They have different accessibility needs. They have different cultural perspectives. As a result, theme parks provide experiences to cater to the wide range of needs and expectations.

An example was the introduction of single-rider lines: rides often seat visitors in pairs and therefore can have unused capacity when groups have an odd number of people or for solo visitors. Worse, solo visitors would wait as long as large groups, even as they could see empty seats on the ride. By creating a line just for individual riders who don't mind sharing with strangers, the excess capacity can be used up—providing a quicker queueing experience for all guests.

Users of technical systems are just as varied. They can come from different geographic regions, be of different sizes, vary in their frequency of use, and so on. And aggregating them is just as pernicious. An example is when a company has their datacenter in North America, where the majority of their customers are. If the data is aggregated, a customer located in Eurasia facing subpar performance might not trigger an alert: the user may become unhappy, even if all SLOs appear to be met.

User Perception Matters

Unlike machines, humans perceive interactions based on their past experiences and attempt to create context based on what has happened: a machine might make several attempts to connect without those attempts creating any kind of storyline. This is less true for humans, where they build theories based on patterns, and it is at our own peril to ignore this fact.

We cannot, obviously, measure how users feel at every moment because it is intrusive and expensive. We also do not want to rely on users venting their frustration at customer support or online on Twitter either, because then it's too late. But we can start thinking about user perception as a factor in our SLOs and acknowledge that it plays a role if we are to be truly user-focused.

Perception is by definition subjective, sometimes in counterintuitive ways. An illustration comes from a phenomenon called paradoxical heat: when a person holds a warm pipe in their left hand and a cool pipe in their right hand, they sense painful heat, even if neither pipe individually feels unbearable. We are unaware of a directly analogous phenomenon for SRE, but a similar idea might be having two minutes of downtime, followed by two minutes of availability, followed by another two minutes of downtime.

This won't feel like four minutes of outage: anyone who has experienced spotty WiFi coverage will understand the oddly intense anguish that comes from intermittent connectivity. It can feel worse than not having Internet access at all, because it robs you of your sense of control over the situation: should you keep trying or do something else? Not knowing whether the next outage will be in a minute or not at all can be very frustrating. So we can't just look at the raw data itself but have to also think about how that data represents experiences. Four one-minute outages alternating over an eight-minute period may feel worse than a continuous four-minute outage.

Perception also plays a role in the least interesting part of visiting a theme park: waiting in line for a ride. However, huge investments have been made to create engaging and sometimes interactive experiences during the queue to make the experience feel less painful. Before a Harry Potter-themed ride, visitors roam an immaculate set modeled after Hogwarts, the fictional wizarding school, and the immersion makes the time seem to go by faster.

Theme parks also post estimated wait times so that visitors feel a sense of control about whether or not to join the queue—and these times are padded so that guests feel delight at "saving" time. Isolating wait time provides some information, but if you have set up a standard and even sticking to it leads to unpredictable outcomes, then you must realize that you need more information to guide your decisions.

How do you learn about these expectations? You can look at user-behavior data, such as when customers drop off, and try to figure out a trend. Or you can ask them directly through surveys and interviews. But it's important to think about when is the right time and place to ask them. Asking after a major outage will yield different answers than after a period of calm, and asking them before their issue is resolved is different from asking afterwards.

You will also want to pair up with someone who understands how to craft useful survey questions: for example, you do not want to create leading, ambiguous, or unclear questions, and you want to use a Likert scale. Poorly designed survey questions lead to low quality data, and sometimes people can assume that surveys are the problem, but that's blaming the tool rather than the person wielding it: more than likely it is how surveys are created and conducted that are the problem.

Benefits

We all have limited resources, especially time. When we choose the most meaningful, user-focused SLIs and SLOs, we make the most of those resources. You're prioritizing for the experience your users want and creating the boundaries for services. If something goes down, but it doesn't impact user experience, it's still important, but it isn't necessarily urgent. Just because we can do something doesn't mean we should. We can wake people up in the middle of the night to manage an incident, but are we alerting for the right things? What matters and what doesn't?

There is a broader benefit: the third age of SRE is upon us, and it is one that posits that reliability is cross-functional, something that not just developers and technical project managers need to think of, but also accountants, lawyers, and customer support teams.

Yet this isn't a one-way street. Just as everyone should have a reliability mindset, we must remember why reliability matters. It's not just done for its own sake (and actually can be costly, a detriment to feature velocity, and cause for burnout) but because customer experience matters and customers demand reliability. Reliability that doesn't include a user-focus is only tackling part of the problem, and when it becomes more developer-focused than customer-focused it becomes about ego. So everyone must have a user-oriented mindset.

Such a shift can be frightening because users can seem subjective, but, unless our only users are machines, that's how it goes. What we can do is approach it differently, with wonder and excitement. How can we delight our users the way theme parks spark joy for visitors? Our favorite example of thinking about users: at the Disney World parks, designers created different floor textures for each land, so that even your feet know when you're moving into a new experience. It may be at a level beyond what we need, but we can afford to walk a few steps in the right direction.

References

[1] https://en.wikipedia.org/wiki/Disney_utilidor_system.

[2] M. Rintoul, "User Experience Is a Feeling," UX Matters, October 2014: https://www.uxmatters.com/mt/archives/2014/10/user-experience-is-a-feeling.php.