Reliable by Design

The Importance of Design Review in SRE

LAURA NOLAN

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

Every organization has regrets about software that doesn't scale, that's difficult to run, or hard to use, or where we just wish we'd done something differently early on, when it would have been easier and cheaper. Sometimes we need to execute fast and accrue technical debt, but often the right thing would have been as easy and fast as the wrong thing—and those are our failures as a profession.

In many organizations (especially larger ones), when a new system is being built or a major change is planned for an existing system, a design (also often known as an RFC, or Request for Comment) is written and reviewed by peer engineers. This is a document that describes the planned change, including the reasons for making it, and alternatives to the proposed design that were considered and rejected. The ideal level of detail is just enough that any competent software engineer could implement the system from the design—in other words, it should be significantly higher-level than code, while clearly describing requirements, system architecture, dependencies, and tradeoffs.

Of course, not every change needs a design document, and people often aren't sure where to draw the line. My heuristic is that any project that is going to lead to the creation of new monitoring or runbooks, or large revisions to existing ones, merits a written design. This does not mean that failure to create required monitoring or runbooks excuses the need to produce a design document.

I'm going to nail my colors to the mast here and say that if you're not producing designs and participating in design reviews with partner teams, then you're not doing SRE but some other flavor of operations. SRE is predicated on having agency and on teams having a voice in decisions that affect the systems they are responsible for. Without designs and a review process, teams don't have the insight they need into the changes that others are planning in the production environment, so having that voice in significant decisions becomes impossible.

Written designs have many advantages over informal discussion or presentations. As an author, the written form pushes you to think through details that you might not otherwise spend enough time on. As a reviewer, it gives you time to reflect on the proposed change. It also works better for distributed teams, because feedback can be given and responded to in an asynchronous manner. A long-term advantage of written designs is that they can provide a history of major changes in your organization's systems as well as the reasons behind them and the decisions made. Over time, the reality of your systems will diverge from original designs, but an archive of design documents will still be a valuable resource.

The design review process can be problematic in a few ways on a human level. One problem is time: feedback on designs may drag on for several weeks if there are many interested reviewers. I recommend setting a clear deadline for feedback (in the header of the document itself). Around two or three weeks is ample. If there are unresolved discussions at this point, then schedule meetings to discuss (either one meeting or multiple one-to-one meetings). This will save time overall, and it is easier to resolve technical disagreements face to face.

Another big problem is the use of the design review process to show off, debate matters of taste, or nitpick. This kind of behavior makes people reluctant to write and share designs,

and the resulting failure to communicate leads to repeated work, a lack of shared understanding, and failures to catch major problems that colleagues would have noticed. Design review comments should be well intentioned and solely about the important points in the design rather than the color of the proverbial bike shed. Think carefully before commenting. Many large organizations develop norms and guidelines for technical discussions, including pointing out and discouraging this kind of "bike shedding."

I've never seen much guidance on how to perform design reviews as a peer engineer or as a technical lead. People tend to read the document and apply their expertise in an ad hoc way. As someone who has reviewed a fair number of such designs, I've found that it's time-consuming, and I often worry that there's something important I haven't thought about. There's no structured way to approach the problem.

Atul Gawande's book *The Checklist Manifesto* [1] may point towards a solution. Gawande is a surgeon. He noticed that it was very common to make errors in complex surgical procedures. He distinguished between two kinds of errors: errors of ignorance, where not knowing something causes a mistake, and errors of ineptitude, where we don't make proper use of what we know. In the modern world, surgery is such a complex task that forgetting steps, or failing to plan ahead for some eventuality, is almost inevitable. Gawande looked at what other professionals do—in professions like civil engineering and aviation—and it turns out they use checklists to avoid errors of ineptitude.

Checklists may sound like a tedious process—and nobody really likes more process—but bear with me. Surgical checklists are not a substitute for professional expertise. In fact, they absolutely require that expertise to execute them. They are not long manuals that prescribe every detail of every step in a process but instead are prompts, intended to make sure you don't accidentally leave out a key step in a complex task. Surgical checklists are quite short, leaving minutiae to the judgment of those using them; the WHO safe surgery checklist [2] fits on one page, although it does refer to other checklists that may need to be consulted under certain circumstances.

It turns out that well-crafted checklists make a big difference in surgical outcomes—a 2009 study showed that the WHO checklist reduced the incidence of post-surgical complications by a third. In addition to making sure basic (but important) things aren't forgotten, they also encourage and empower all members of a team to point out omissions or problems. They can make teams work better.

I believe checklists can help us improve our system designs too. There is a lot of wisdom in the SRE profession about how to design operable, scalable, reliable, distributed systems. We can add a lot of value at this stage of the process. But there's no checklist to help us do it. What might such a checklist look like? Here's my version [3]:

- **What and why**: do I understand the need for the change, the design itself, and how the proposal relates to other systems?
- **Who**: are there affected teams that haven't been asked to look at this design? If there are privacy or security implications of this system, are there appropriate reviewers?
- **Alternatives considered**: is building a new system the right approach?
- **Stickiness**: what's hard to change about the proposed system?
- **Data**: consider consistency, correctness, encryption, backup, and restore strategies.
- **Complexity**: where is this design overly complex, and can that complexity be reduced?
- **Scale and performance**: how does the design support the scale and performance needed?
- **Operability**: how will the system support (or not) the humans running it?
- **Robustness**: how does the design handle failures, and other issues such as overload?

This high-level checklist is fairly terse, as a usable checklist needs to be—remember, this is here to prompt your expertise, not to replace it. For some designs, some sections of the checklist may not apply—maybe the design in question is a piece of automation that doesn't need to scale, or a stateless service that doesn't need to deal with some of the data considerations. The sections below give more detail for each item on the checklist and, in some cases, further sub-checklists.

The **what and why** questions are first because they are the most important. If you read a design and don't understand it and why it's needed, then the design is missing information or lacking in clarity. If you don't understand it when you're reviewing the design document, you definitely won't understand it when you're trying to respond to a production fire. The best way forward here is to tell the author which parts you're having trouble with and ask them to update the document before proceeding.

Next, who:

- Is there a good reason that you've been asked to review this system? It's good to understand whether the author is looking for some particular expertise or perspective from you, and make sure you've addressed that.
- It's also useful to check who else has been asked to review and that all the affected teams have been asked. Support or operations teams are often left out to the detriment of all involved. Owners of systems that the new system will depend upon should usually be asked to review new designs.
- Many changes should be reviewed specifically for privacy and security.

Alternatives considered is a subject often neglected but important:

- Is there an open-source tool, or a similar proprietary system at this organization, that might work? If so, did the author of the design talk to owners of those similar systems about this use-case? Proliferation of systems is hugely costly. It takes time to build and maintain them, and it complicates an organization's production environment.

Stickiness: give special consideration to thinking about which aspects of a proposed system will be hard to change in the future.

- Imagine you're trying to migrate all the users of the system away from it to its replacement or that you're planning a major change of some sort. What aspects of the design will make that easier or harder? For example, allowing users to extend your code limits what you can do in the future and makes migrating them to replacement systems much more difficult, and so does tight coupling with other systems.
- What assumptions are baked into the architecture or the data model that might change in the future?

Data:

- What is the flow of data through the system?
- What are the data consistency requirements, and how does the design support them?
- Which data can be recomputed from other sources and which cannot?
- Is there a data loss Service Level Objective (SLO)?
- How long does data need to be retained, and why?
- Does it need to be encrypted at rest? in transit?
- Are there multiple replicas of the data?
- How do we detect and deal with loss or corruption of data?
- How is data sharded, and how do we deal with growth and resharding?
- How should data be backed up and restored?
- What are the access control and authentication strategies?
- Have relevant regulations such as GDPR and any data residency requirements been addressed?

Complexity:

- Does each component of the system have a clearly defined role and a crisp interface?
- Can the number of moving parts be reduced?
- Is the design similar to existing systems at this organization? Is it built using standard building blocks (K/V stores, queues, caches, etc.) that engineers at this organization already understand? Does it use the same kinds of plumbing such as RPC mechanisms, logging, monitoring, and so on?
- Does the proposal introduce new dependencies (e.g., uses a different type of message queue than other systems in the same organization) and if so, is that really necessary?

Scale and performance:

- What are the bottlenecks in this system that will limit its scale and throughput (not forgetting the impact of writes and locking)?
- What's the critical path of each type of request, and how do requests fan out into multiple sub-requests?
- What is the expected peak load, and how does the system support it?
- What is the required latency SLO, and how does the system support it?
- How will we capacity plan and load test?
- What systems are we depending on, and what are their performance limits and their documented SLOs?
- What will it cost to run financially?

Operability:

- How does the design support monitoring and observability? For instance, systems involving queues may require extra care in monitoring.
- Do all third-party system components provide appropriate observability features?
- What tools will be available to operators to understand and control the system's behavior during production incidents? How will these tools make clear to the operator what specific actions they should take to avoid surprises?
- What routine work is going to be needed for this system? Which team is expected to be responsible for it? How much of it can and should be automated, and will that automation reduce the operating team's understanding of the system?
- How do we detect abusive users or requests, and what action can we take in response?
- If the design involves relying on third parties (such as a cloud provider, hardware or software vendor, or even an open-source community), how responsive will vendors be to your feature requests or problems?
    - Are all configurations stored in source control?

Robustness:

- How is the system designed to deal with failure in the various physical failure domains (device, rack, cluster/AZ, datacenter)?
- How will it deal with a network partition or increased latency anywhere in the system?
- Are there manual operations that will be required to recover from common kinds of failure?
- How could an operator accidentally (or deliberately) break the system?
- Is there isolation between users of the system?
- What are the smallest divisible units of work and data, and will we likely see hotspotting or large shards?
- What are the hard dependencies of this system, and can we degrade gracefully? How to ensure soft dependencies don't become hard dependencies?

- How can we restart this system from scratch, and how long will that take? Do we depend on anything that might depend on this system? Don't forget DNS and monitoring.
- How will this system deal with a large spike of load?
- Does the system use caching, and if so, will it be able to serve at increased latency without the cache?
- Is the control plane fully separate from the data plane?
- Can I canary this design effectively (e.g., leader-elected designs are hard to canary)?
- Can this system break its back ends by making excessive requests?
- Can this system autonomously drain capacity, and how have risks been managed, in particular with respect to human operators' ability to understand and control the system?
- Can this system autonomously initiate resource-intensive processes like large data-flows (perhaps for recovery purposes), and how are those risks managed?
- Can this system create self-reinforcing phenomena (i.e., vicious cycles)?

These are the things I think about when reviewing a design. No two systems are the same, so not all of these questions make sense for every type of system. As with the WHO surgical safety checklist, local variations are very much encouraged. This is a starting point [3].

All systems involve risk, and all systems make tradeoffs. Better system design won't eliminate all problems. We just can't anticipate everything—errors of ignorance are inevitable. But errors of ineptitude are avoidable, and part of maturing as a profession is getting more systematic about reducing errors of ineptitude.

A good design helps us to understand tradeoffs and risks more thoroughly and make reasoned, deliberate choices that make the most sense for our organizations. Taking the time now to write a design for your team's next big project and get it reviewed by your peers might be the most impactful work you can do.

References

[1] A. Gawande, *The Checklist Manifesto: How to Get Things Right* (Metropolitan Books, 2009).

[2] WHO Safe Surgery Checklist: https://www.who.int/patientsafety/safesurgery/checklist/en/.

[3] L. Nolan, SRE Reliable by Design checklist: https://www.usenix.org/sites/default/files/fall19_sre_checklist.pdf.