Ask-Me-Anything Engineering

EFFIE MOUZELI

Effie Mouzeli studied physics and distributed scientific computing but didn't turn out to be a physicist or a scientific computer scientist. She has worked as a systems engineer/SRE at a number of startups and small organizations (most of which are not with us anymore), where her responsibilities were usually automation, infrastructure architecture, and working closely with developers. Currently, she is on the SRE team that takes care of Wikipedia and its sister projects at the Wikimedia Foundation. emouzeli@runbox.no

Small organizations are the reality for a number of people doing operations. Despite that, there are limited resources on the subject of working in a systems team of a few engineers. On the contrary, there is literature on how large organizations implement SRE, how they got to 99.999% availability, and how to process millions of metrics per second. Those are really good and interesting reads, but I have been in the shoes of a person reading such articles and thinking, "I enjoyed reading this, but I can't use it." For the purpose of this article, the terms "small organizations/companies" but also "small-scale" will be used to describe organizations and startups where there is a single SRE or a small team of SREs.

Everything was just a few servers once, and everything started from something. Let's just think about how much we rely on products by small companies like local news sites or local ferry booking services. Moreover, we mustn't forget that some of us live in cities and countries where there are no large engineering teams, and those environments are the only available places to work. And those are just a handful of reasons why small-scale companies matter.

There are a few major challenges that a systems engineer at a small organization will have to constantly deal with:

- Paying off someone else's technical debt, almost alone
- Managing a live infrastructure, almost alone
- Caring for the development teams, almost alone

I will try to provide an overview of what it is like working in small-scale environments, what to expect, how SRE concepts can be beneficial, and what can be learned.

The Role of an A.M.A. Engineer

In small software companies, I always considered the systems team as setting their tempo to the development team. The development team needs that expertise so as to, in turn, set the standards for the organization. For example, if bootstrapping a new server takes a day, everyone will consider this normal and will never demand to have a new server ready in an hour. If pushing code to production requires 10 manual steps, the development team will never be able to deliver faster. It is up to the SRE to automate manual steps if possible.

Another important aspect of this role is to be the facilitator between development and infrastructure. It is up to the systems engineer to make the infrastructure accessible to developers, show them where the controls are, and be there for them. But beyond that, people come to us for answers when things don't add up. An Ask-Me-Anything engineer can be the expert on DNS, databases, networking, Linux, and monitoring. One can expect to receive a broad range of inquiries, from "My service can't access the database" to "My dad wants to buy a new laptop." For me, I believe the most bothersome question was, "What's the guest WiFi password?" That is, until I taped QR codes on every office wall.

In addition, our experience will help us to plan wisely for the future. We should be able to see a few steps ahead and have a plan about how the systems will grow along with the organization.

It is not unusual, though, for systems engineers not to get the credit they deserve. While large organizations go a long way to achieve "five nines," that is not always the case in small ones. Keeping the developers happy and productive while having few incidents does not translate financially, for example, like a new feature would. As Heidi Waterhouse has said, "No one remembers the crisis averted" [1]. Sadly, good systems engineering does not have a direct Return on Investment.

Small-Scale Technical Debt

Technical debt is every organization's Achilles' heel, regardless of size. There is almost always a mountain of it. Even at startups, there is rarely a dedicated person for operations in the beginning, meaning that the multi-hat engineering team is trying to make systems work and make technical decisions to the best of their knowledge.

Identifying what is generating technical debt is a very good start towards slowing it down. It is impossible to come up with a complete list of reasons behind technical debt in systems at small environments, but I can name a few from my experience.

Lack of Processes, Documentation, and History

When a few people are developing and running a product, it is normal to solve problems ad hoc. Minor and major problems are dealt with as they come up, and then are forever forgotten. **There are no runbooks, no postmortems, no histories.** But those issues derive from deeper ones: there are no standard processes for how to do things, e.g., introducing a new service to production, but also no documentation as to how things work. Especially in fast-paced environments like startups, we subconsciously consider documentation as a waste of time.

Cargo Culting

New and immature technologies are adopted under the false assumption that they can fix anything and that by using them, the organization can stay up-to-date and relevant. Together with the lack of appropriate systems background or experience, it just equals technical debt. Not to mention that sometimes, we go as far as fitting our problems into the solution we want to experiment with. Without drawing any lines, or putting any limits, this can lead to a Frankenstein infrastructure. Some notable examples are Kubernetes and Docker; they provide solutions, but how many times did a team with limited resources ask whether it had the human capital to go in this direction?

Short-Term Planning

Cargo culting itself is a symptom of another underlying problem: the culture of short-term planning. Everything moves so much faster when the development team is 20 people rather than 100 or 200. This team of 20 people has a list of features to add to the product, which in turn have a list of requirements. Still, there is no vision as to how the systems themselves should look a year after those changes. A simple example would be, "We estimate that if we market this new feature, we will gain 20% more clients within six months." That is great, but have we done any capacity planning to handle that traffic?

Waiting for a Hero

This does not generate technical debt per se, but it is a consequence of all the above. At some point the organization has so much duct tape and WD-40 that it simply waits for someone to make sense of the chaos and save it. How chaotic this chaos is depends on a number of factors. If this is a startup, the chaos is proportional to how late to the party a systems engineer has arrived. On a brighter note, at a startup it is highly likely one can talk to the people who created all that debt and get answers. If this is a long-running company, it depends on how many systems engineers have come and gone over the years as well as how many people assumed that role.

The Five Stages of Technical Debt

Let's assume that we have joined an organization as the first SRE. Our hypothetical new startup, **everythingsocks.io**, has 30 servers, 50k customers, and about 25 developers, all using the same account, *root*. Funding is secured, business is booming, and the future looks bright!

We arrive in a new fast-paced environment where we don't know anyone, we are required to run a live infrastructure we have never seen before, and we have no idea what is coming. One thing is certain though: we are going to go through the five stages of technical debt [2].

1. Denial

The product looks functional, as well as its systems, and we reckon our job is to initially keep everything running and then move forward. All we have to do is hold the wheel and drive. EverythingSocks looks like an awesome place to work after all. They have free lunches, a pool table, and free yoga lessons!

2. Anger

While we are sure that everything is going great, we begin to get interrupted. A developer reports they think the auth server is overloaded. They believe an additional server might help. Another one pops by, saying that they are getting some 500s, just like last week. A third one appears complaining they are unable

to access StackOverflow. It goes without saying that everything is urgent. We realize that we are smiling because we have no idea what is going on. We are frustrated.

3. Bargaining

This is the part where we believe that we must gain control. We are trying to show some faith in our abilities and not get too overwhelmed, even though everything is on fire. Our colleagues seem to be good people after all; we can make it work!

4. Depression

At this point we are desperate for information. We are running around trying to figure out what's under the ground we are standing on. The more we uncover, the more we feel despondent. We find out that everyone has root access to the databases, even the microservices, and the main application is logging plaintext passwords. This is depressing.

5. Acceptance

We get to acceptance when we finally see some light at the end of the tunnel. We have a better overview, we are feeling optimistic, and we have a plan of how to make it better.

Getting Control

With all these problems discovered, as a whole, there is a mountain to climb. The problems have to be broken down into pieces and prioritized. The development team and our intuition can help us get there.

Asking a Lot of Questions

But in addition to asking questions it's crucial to ask the right questions. For example, ask the development team what *they* need, what are their daily pain points, and what they believe should be improved. The team might request a proper staging environment but fail to mention that they manually delete application logs every week. Read between the lines!

Understanding the Product

A bad habit I have noticed among systems engineers is that they tend to distance themselves from knowing how the applications they are managing work. This is a mistake that can turn many incidents into a wild goose chase. How subsystems communicate with each other, what they are doing, as well as what external dependencies they have, should be something an SRE is aware of. For instance, if a payment provider is taking longer to respond, it might exhaust the application workers. If you don't know that one of the apps depends on a payment provider's response time, you will find out the hard way, through reading logs, stracing, tcpdumping, etc., while everything is on fire.

Documentation Is Bliss

As you are gathering information about literally everything, write it down: what you learn, what you think is missing, what needs improvement. Your ultimate goal is to eventually have a board with the work that needs to be done. You will feel hopeful when there is finally a comprehensive list of tasks that include immediate and future needs. This is how one gets to Acceptance!

Understand Your Limits

I strongly believe that a good engineer is able to understand what they can do under certain circumstances. Try not to rush. Do not start making promises that "it will take two days." If your work is fast but sketchy, it will keep coming back to haunt you, and that does not scale well. Equally important, having nothing delivered on time can become part of the culture.

When it comes to introducing new tools to help you in your day-to-day tasks, start with the familiar ones. You will find time later to try something new and fancier, together with researching. And if what you are researching is not working out, learn to let go and move on. The more time you spend on one front, the more everything else is falling behind. I was in a team, a newly formed team, that kept promising to migrate the infrastructure from one datacenter to another within three months, while migrating our servers from bare metal to VMs, while migrating from Chef to SaltStack, while production was running. What could possibly go wrong?

Consistency

Creating standard processes and rules, and then sticking to them and defending them, is your true ally: for example, processes about new server requests, new applications, new users, rules that all microservices should be managed by `systemd`, and all packages must be installed via configuration management. You need to keep snowflakes to a bare minimum as much as you need your sanity.

The Big Picture

I will lay down the components a functional infrastructure needs in order to be manageable. No matter how many servers we have or how much traffic we serve, we need all of them. The problem in small-scale is all will be implemented by a single person or a tiny team. That is the beauty and the difficulty of small-scale. The strategy here is divide and rule. Attacking all of them at the same time can be chaotic and stressful, so iterate, little by little!

Automation and Provisioning

Try to manually create a staging environment and document all steps; it will help you learn more about the product. Next time you revisit it, write scripts for that. Later on, have Jenkins run those scripts. Having Jenkins running silly bash scripts is better

than you running them. In the same fashion, decide how you will provision. Be it bare metal servers, virtual machines, or container images, provisioning must be automated. Infrastructure as code is the best way.

Observability

Based on current needs, decide on monitoring and alerting. If real-time monitoring looks like a lot right now, choose a more simple solution. You will know when a more sophisticated solution is needed. What about metrics? Which metrics can be used? Which metrics can be pulled from logs? Which are of business value, and which can be used as health indicators? Talk to the development team and figure it out. Together. For instance, low sock sales is a business metric that can imply systems issues.

Updating and Viable Backups

Keeping software and tools up-to-date while in a tiny team is very, very challenging. It is up to you to judge when and what should be updated given the time you have available. Viable backups are our dirty secret, and there are countless horror stories to prove it. We just strap them on flying unicorns and never look back. Even the most pessimistic personalities, when it comes to systems, hope for the best-case scenario: they won't have to use their backups. In other words, figure out what needs to be backed up and test those backups. You need the confidence and security that you can rely on them.

Security

Common practices like limiting access unless needed (e.g., to databases), using different database passwords in production and staging, keeping track with security updates, etc., are a very good start. Taking this lightly can lead to those upsetting stories about junior engineers deleting the production database on their first day.

Emergency Response

You are always on call. Sit with the development team and discuss what can be done in possible scenarios, along with some emergency checklists, and eventually create some early runbooks.

Legacy Systems

This is more common in existing companies than startups: very essential services running somewhere, but no one knows anything more about them. And when they break, you will be called on to fix them. For your own peace of mind, work with your colleagues on ways to remove those black boxes.

The aim is to bring the systems to a manageable state while assisting the development team with their deliverables. Balance comes through small iterations, improving what you have in each one. Don't rush; **let your systems mature.**

Building Habits and Culture

It is not uncommon in software companies for developers to dislike working with systems engineers, and vice versa. This is usually due to a lack of communication and bad attitude from all quarters. Certainly in a place with a single SRE, this is not going to work to anyone's benefit. Being approachable is key in building trust between you and the development team. After all, they are ***your*** team as well. Try to have standard meetings with each other, and share what is in the roadmap. Guess who will have to work extra hours if you are playing with alerting while developers are about to roll out a feature that needs a new dedicated database server. It is important to learn to meet each other halfway.

Furthermore, many developers are not proficient in systems engineering, and that is generally accepted in small companies. Instead of being frustrated for being asked for the 10th time, "How do I restart a service?" teach them and document it! Help them become better. Help them learn how to use what you are building. After leading a systems crash course with 15 developers, I cried with joy the first time a colleague used `ngrep` to debug an issue. Generally, the more self-serviced the development team is, the less toil for you.

Lastly, being arrogant is something that you can't afford. If people prefer running around production with scissors because they just don't want to deal with the SRE, then you're holding a time bomb. A few years ago, a group of developers wanted to experiment with Docker, but my team resisted even running rough tests with Docker. Eventually, the developers set up a staging environment using Docker at a cloud provider outside of our infrastructure. This meant that our proprietary code was deployed somewhere outside the control of the systems team. But who is to blame here?

It Takes a Village

Your efforts will go as far as management and the development team want to go. Same goes for if what you have started will turn into a team or teams in the future. If there is not enough management buy-in, there is an upper limit to what can change and improve. You may want to introduce service level objectives [3] so that, in turn, you can add meaningful alerts. This can't be done without developer assistance. In addition, it is impossible to have blameless postmortems if nobody wants to write them and if people prefer to point fingers at each other.

There are chances that one may really try to push for changes and not get the desired results. Aristotle wrote "one swallow does not make a spring" [4], and it is true. A person alone might not be enough if the rest of the team won't listen, and in my personal opinion, that is not a problem an SRE should be solving.

Small companies are intimate, and they provide us with that feeling of belonging, but they can also feel a little lonely, especially if one is flying solo. What helped me over the years was keeping in touch with other people doing the same job, through meetups, group gatherings, and chats. It is essential to be able to share the tech and non-tech problems at work with people who understand.

Is It Worth It?

This is a coming-of-age experience for an engineer. There are no safety nets and no one you can pass the ball to. Every decision will be more well thought out since any consequences will directly affect you and your peers. Moreover, when money is tight and time is limited, one gets creative. You work with what is in front of you; you can't simply add 100 servers or you can't waste days trying to find what is wrong.

You will acquire a really broad skill set. It will range from debugging tools, networking, databases, and programming, to project management, human resources, people skills, event planning (true story), and, possibly, how to help a colleague's dad find his dream laptop.

Mistakes will happen, things will break again and again, you won't always have all the answers, and sometimes you will create more technical debt than can be handled. **And at the end of the day, our systems will not be perfect, just manageable.**

References

[1] H. Waterhouse, "Y2K and Other Disappointing Disasters: Risk Reduction and Harm Mitigation," SREcon18 EMEA, USENIX, 2018: https://www.usenix.org/conference/srecon18europe/presentation/waterhouse.

[2] Coined after the Kübler-Ross model: https://en.wikipedia.org/wiki/Kübler-Ross_model.

[3] Service Level Objectives: https://landing.google.com/sre/sre-book/chapters/service-level-objectives/.

[4] "The Young Man and the Swallow": https://en.wikipedia.org/wiki/The_Young_Man_and_the_Swallow.