# From Sysadmin to SRE in 2587 Words

VLADIMIR LEGEZA

Vladimir Legeza is a Site Reliability Engineer in the Search Operations team at Amazon Japan. For the last few decades, he has worked for various companies in a variety of sizes and business spheres such as business consulting, Web portals development, online gaming, and TV broadcasting. Since 2010, Vladimir has primarily focused on large-scale, high-performance solutions. Before Amazon, he worked on search services and platform infrastructure at Yandex.
vlegeza@amazon.co.jp

"If you cannot measure it, you cannot improve it."

*—William Thomson, Lord Kelvin*

Site Reliability Engineering is a set of practices that allow a variety of companies to run and support systems at large scale efficiently and cost-effectively. The key difference that distinguishes sysadmins from SREs is the single property: the point of observation. There is only one fundamental principle that can drive you to this relatively new field and lead you to understand all these practices, origins, adaptations, and further improvement ideas that finally will increase your users' and customers' loyalty and satisfaction.

Instead of jumping directly into the definition of principles, let's figure it out ourselves through the following examples.

Let's say a manager asked you to create a small new service: "A sort of a simplified Web crawler. It has to receive a base URL, download its content, find and return a list of all URLs retrieved from that page with the status whether it is valid and accessible or not." This task is more or less straightforward. An average software developer can immediately begin the implementation using not more than a hundred lines of high-level code.

An experienced sysadmin given the same task will, with high probability, try to understand the technical aspects of the project. For instance, she may ask questions like, "Does the project have an SLA?" and dig deeper: "What load should we expect, and what kind of outages do we need to survive?" At that point, prerequisites might be as simple as, "The load will be no more than 10 requests per second, and we expect that responses will take no longer than 10 seconds for a single URL request."

Now let's invite an SRE to the conversation. One of his first questions would be something like, "Who are our customers? And why is getting the response in 10 seconds important for them?" Despite the fact that these questions came primarily from the business perspective and did not clarify any technical details, the information they reveal may change the game dramatically. What if this service is for an "information retrieval" development team whose purpose is to address the necessity of the search engine results page's content validation, to make sure that the new index serves only live links? And what if we download a page with a million links on it?

Now we can see the conflict between the priorities in the SLA and those of the service's purposes. The SLA stated that the response time is crucial, but the service is intended to verify data, with accuracy as the most vital aspect of the service for the end user. We therefore need to adjust project requirements to meet business necessities. There are lots of ways to solve this difficulty: wait until all million links are checked, check only the first hundred links, or architect our service so that it can handle a large number of URLs in a reasonable time. The last solution is highly unlikely, and the SLA should therefore be modified to reflect real demands.

What we've just done is to raise the discussion to a new level—the business level. We started with the customer and worked backward. We understood the service's use cases, identified its key aspects, and established and adjusted the SLA. Only now can we begin to architect the solution. This is the exact meaning of the first of Amazon's leadership principles: "Customer Obsession—Leaders start with the customer and work backwards" (https://www.amazon.jobs/principles). Absolutely the same idea appears in the first of Google's "Ten Things" philosophy: "Focus on the user and all else will follow" (https://www.google.com/intl/en/about/philosophy.html).

At this point, I want to present a short, three-character terminology clarification to avoid confusion or uncertainty:

SLI: Service Level Indicator is a single, measurable metric related to service behavior that is carefully chosen with a deep understanding of its value's meaning. Every possible value should be clearly defined as "good" or "bad." Also, all barrier values that turn "good" to "bad" and vice versa should be precisely specified. It can be measured on its own terms and conditions and may have more than one axis of measurement. However, the rule of thumb is that every indicator must be meaningful.

Example SLI: 99% of all requests per one calendar year should be served in 200 ms.

SLA: Service Level Agreement is a set of SLIs that defines the overall behavior of what users should expect from the service. A good SLA represents not only a list of guarantees but also contains all possible limitations and actions that may take place in specific circumstances: for instance, graceful degradation in a case of primary datacenter outage, or what happens if a certain limit is exhausted.

SLO: Service Level Objective is absolutely the same set of SLIs that an SLA has but is much less strict and usually raises the bar of the existing SLA. The SLO is not what we have to have, but what we want to have.

Example: For an SLA, a single SLI might be defined as "99% of all requests should be served in 200 ms," and in the SLO the same indicator may look like "99.9% of all requests should be served in 200 ms."

For further details, please refer to Google's *Site Reliability Engineering* (https://landing.google.com/sre/book/chapters/service-level-objectives.html).

The principle that the user's perspective is fundamental is very powerful and leads us to the understanding of vital service aspects. Knowing what is valuable for the customer provides a precise set of expectations that have to be finally reflected in the SLA. And by being carefully crafted, the SLA may shed light on many dark corners of a project, predicting and preventing difficulties and obstacles.

But first, the SLA is designated as a reference point to understand how well a service is performing. There might be hundreds of metrics that reflect a service's state, but not all of them are appropriate for an SLA. Although the number of SLIs tends to be as minimal as possible, the final list of SLIs should cover all major user necessities.

Only two relatively simple questions should be answered positively to indicate that an investigated metric is a good candidate to be chosen, or otherwise, should definitely not be.

- Is this metric visible to the user?
- Is this metric important enough to the user (and from his/her perspective as a service customer) that it needs to be at a certain level or inside a particular range?

## Internal SLAs

What if the service is not an end-customer-facing one. Should this service have its own SLA too? To clarify the "Yes" answer, let's step back a little bit from the technical terminology; we will see that the SLA itself is nothing but a list of criteria of what you can expect from the service, that is, a simple list of expectations.

When you are thinking of a new service to be used in your project, one of the first questions you want answered is, will this service meet your expectations or not? It does not matter what kind of service it is: it might be an external ticket-based authentication system, local corporate storage, or a cross datacenter distributed message-passing bus. But to figure out whether you can you use it or not, you should know what this service promises you and what limitations are applied. Hence, every producer-consumer relationship is built on certain expectations, and, hence, every service should provide a list of guarantees for all valuable expectations regardless of whether the service is an internal or external one.

To support this statement let's consider a message distribution service that consists of four main components:

- "Data receiver": accepting and registering messages
- "Data transformer": adjusting message content with data from separate external sources
- "Distributor": delivering messages to multiple endpoints
- "Consumer": receiving data from the endpoint over a "publisher–subscriber" model

At the moment, this system is working fine: no errors, no alarms.

One day, one of the top project managers comes to us and poses the following: "One of the projects we are working on now uses the 'message distribution service.' From time to time, we will need to send a huge amount of data in a short time period. Can we handle this? Or what should we have in order to use this service?"

Let's work this out gradually. Having actual numbers for the data amount is handy. Let's say it will be three times more than the maximum known peak-time value. However, this will not provide us with a clear understanding of whether we will be able to handle this traffic or not. The reason is simple: even if we know how much data is managed by the service right now during peak times, we would still need to know the break point where we reach the service's capacity limit to be able to compare it with the forecast.

Our message distribution service has several components. As we are aware, the slowest component is the one that dictates overall service capacity: the "strength of the chain is determined by the weakest link." So now we have to spend some time to establish a performance-testing environment and identify breakpoints for every component separately and determine which component is the bottleneck.

So far, we have data that will tell us about traffic-handling possibilities. And if it is fine, we are ready to go on to announce that no changes are required. Of course, businesspeople may ask another question: "Why are we over-scaled that heavily?" but that is a different story, and hopefully it is not the case.

Our calculation reveals that we can deal only with half of expected growth. And we now need to at least double throughput. The deeper we dig, the more complicated planning becomes. Even if we assume that all the components can scale linearly (and in real life, we have to prove this assumption), we are unable to compare performance directly without accounting for one more shared restriction: the delivery time.

Our goal is to pass a number of messages throughout the service from the entry point to the final consumer, and transfer it in a reasonable amount of time.

First of all, we have to provide an actual value for the "reasonable time" metric overall and for every individual component. Only then can we measure the size of an input that the application is able to receive, process, and guarantee to pass as an output inside that "reasonable time." This will lower the throughput even further. However, the positive outcome is that we can now predict the output and compare performance across components.

Time constraints are nothing but SLIs, and the maximum number of messages is the variable that has to be adjusted to process messages during spikes and not break the time limits defined in this indicator. One interesting property of an SLI is that it only rarely changes.

So far, we know:

- Where the bottleneck is. And we can predict where the bottleneck will relocate from where it is now (literally this means that now we know the next slowest component, the next after that one, and so on).
- Expectations for every component (number of messages that can be processed by a single application instance and the expected amount of time that message can spend inside this component).
- Our capacity and how much capacity is actually in use. We are also able to predict capacity drops in case of a variety of outages and make resource reservations accordingly.

And this is still not the end of the story!

## External Dependencies

As you remember, the "data transformation" component has some external dependencies. The "external" means that we are only consumers and cannot control its behavior. The question is, "What capacity can these external services provide and how will their limitations affect our component's performance and scalability?" We want to know what we may expect, and, technically, we are asking for an SLA. Once we get it, we will finally have all we need to figure out what should be adjusted and where and how.

But this real-world scenario gets even more complicated. There may be many types of messages with different priorities and time restraints. It would be tough to say what we should do if the load will grow for only one data type and other types will still be definitely affected. However, by applying the "divide and conquer" principle, we declare specific criteria for every type individually; then it will be clearly noticeable what data may have experienced stagnation and how this issue should be addressed.

A short example: due to high load spikes in high-priority messages, other message deliveries will be slowed down from a few seconds to several minutes. The key question then is, "Are several minutes' delay acceptable or not?" If we know exact barrier values and can quickly identify "good" and "bad" values, then there will be no problem taking the right action. Otherwise, we will fall into a state of not knowing and can only guess what to do or whether we should do anything at all.

As you can see, SREs mostly focus on service efficiency and quality. Architecture and what stands behind is secondary, at least until a service delivers results according to a user's expectations.

## Nontechnical Solutions

Technical solutions are not the limit of SLA potential, and SLAs will give you a hand in other fields as well. The SLA determines, for instance, the right time to hand a new service over to the SRE team to support something as simple as, "If a product meets expectations (i.e., does not violate an SLA), then the product is ready; otherwise, it is not."

All new software projects will have some prerequisites long before they pass architecture review and a couple of proof-of-concept models have been built. When it is believed that an application is ready to be officially launched and begin serving live production traffic, all SLIs become live, and both objectives and agreements start to count. This is the measure's starting point. Because the SLA defines statements over time (the frequently used period is one calendar year), the project should last in this state a significant portion of this time (several months or a quarter) to collect enough datapoints that confirm that the service is stable enough and that there are no agreement violation risks.

## Conclusion

The SRE philosophy differs from that of the sysadmin just by the point of view. SRE philosophy was developed based on a simple, data-driven principle: look at the problem from the user and business perspectives, where "user" means "to take care of product quality," "business" equals "managing product cases and efficiency," and "data-driven" signifies "not allowing assumptions." Identify, measure, and compare all that is important. Everything else is the result of this.

If all this sounds very difficult and complicated, start with these steps:

- Divide large services into a set of smaller ones (treat each component as an individual service).
- Identify service relationships and dependencies.
- Establish an SLO for each service first and maintain it.
- Reassign SLOs to SLAs where required.

Now, as an SRE, you can control systems more accurately and can precisely know when, what, and how much you should scale up. You can do this by efficiently identifying bottlenecks and relocating them from one service to another in a controlled manner. By fine-tuning every part of system capacity to the optimal amount, you will lower costs and raise the bar for an overall positive customer experience.