• **Existing System's Growth in Complexity and Traffic Volume**

As the system grows in complexity and traffic volume, generating a **corresponding increase in events and updates,** the sysadmin team grows to absorb the additional work.Because the sysadmin role requires a markedly different skill set than that required of a product's developers, **developers and sysadmins are divided into discrete teams: "development" and "operations" or "ops."**

• **Development Velocity vs. Operational Stability**

At their core, the development teams want to launch new features and see them adopted by users.

At their core, the ops teams want to make sure the service doesn't break.

At the end of the day, our job is to **keep agility and stability in balance in the system**.

• **Scalability and Reliability**

When products are successful far beyond any early estimates, and their usage increases by more than two orders of magnitude, keeping pace with their load necessitates many design changes.

Such **scalability changes, combined with ongoing feature additions, often make the product more complex, fragile, and difficult to operate.**

At some point, the original product architecture becomes unmanageable and the product needs to be completely rearchitected. Rearchitecting the product and then migrating all users from the old to the new architecture requires a large investment of time and resources from developers and SREs alike, slowing down the rate of new feature development during that period.

Four key themes of **reliability**:

**- Preparedness and Disaster Testing:**

/ Ensure that systems react the way we think they will

/ Determine unexpected weaknesses

/ Figure out ways to make the systems more robust in order to prevent uncontrolled outages

**- Postmortem Culture**: Important to evaluate all of the following:

/ What happened

/ The effectiveness of the response

/ What we would do differently next time

/ What actions will be taken to make sure a particular incident doesn't happen again

**- Automating Away Repetitive Work and Operational Overhead:** Automation lowers operational overhead and frees up time for our engineers to proactively assess and improve the services they support.

**- Structured and Rational Decision Making:** Decisions should be informed rather than prescriptive, and are made without deference to personal opinions.

**• Managing Risk**

**Unreliable systems can quickly erode users' confidence, so we want to reduce the chance of system failure.** However, experience shows that as we build systems, cost does not increase linearly as reliability increments - an incremental improvement in reliability may cost 100x more than the previous increment. The costliness has two dimensions:

- The cost of redundant machine/compute resources: The cost associated with redundant equipment that, for example, allows us to take systems offline for routine or unforeseen maintenance, or provides space for us to store parity code blocks that provide a minimum data durability guarantee.

- The opportunity cost: The cost borne by an organization when it allocates engineering resources to build systems or features that diminish risk instead of features that are directly visible to or usable by end users. These engineers no longer work on new features and products for end users.

**• Software Fault Tolerance**

**How hardened do we make the software to unexpected events?** Too little, and we have a brittle, unusable product. Too much, and we have a product no one wants to use (but that runs very stably).

**• Monitoring and Alerting Systems**

There are many **reasons** to monitor a system, including:

**- Analyzing long-term trends:** How big is my database and how fast is it growing? How quickly is my daily-active user count growing?

**- Comparing over time or experiment groups:** How much better is my memcache hit rate with an extra node? Is my site slower than it was last week?

**- Alerting:** Something is broken, and somebody needs to fix it right now! Or, something might break soon, so somebody should look soon.

**- Building dashboards:** Dashboards should answer basic questions about your service, and normally include some form of [the four golden] signals (latency, traffic, errors, and saturation).

- Conducting ad hoc retrospective analysis (i.e., debugging): Our latency just shot up; what else happened around the same time?

[

The Four Golden Signals of Monitoring

• Latency

The time it takes to service a request.

It's important to distinguish between the latency of successful requests and the latency of failed requests.

For example, an HTTP 500 error triggered due to loss of connection to a database or other critical backend might be served very quickly.

• Traffic

A measure of how much demand is being placed on your system, measured in a high-level system-specific metric. For a web service, this measurement is usually HTTP requests per second.

• Errors

The rate of requests that fail, either explicitly (e.g., HTTP 500s), implicitly (for example, an HTTP 200 success response, but coupled with the wrong content), or by policy (for example, "If you committed to one-second response times, any request over one second is an error").

• Saturation

How "full" your service is.

A measure of your system fraction, emphasizing the resources that are most constrained (e.g., in a memory-constrained system, show memory; in an I/O-constrained system, show I/O).

]

**• Automation**

Automation is the term generally used for writing code to solve a wide variety of problems, although the motivations for writing this code, and the solutions themselves, are often quite different.

More broadly, in this view, **automation is "metasoftware" - software to act on software**.

There are a number of use cases for automation:

- User account creation

- Cluster turnup and turndown for services

- Software or hardware installation preparation and decommissioning

- Rollouts of new software versions

- Runtime configuration changes

- A special case of runtime config changes: changes to your dependencies

**• Continuous Build and Deployment**

**Define build targets, rules for deployment.**

Build phase addresses production aspects such as instrumentation and metrics, operational and emergency controls, resource usage, and efficiency.

Deployment strategies:

- Is the deployment local area or wide area?

- What kinds of quorum are used, and where are the majority of processes?

- Does the system use sharding, pipelining, and batching?

**• Configuration Management**

Although configuration management may initially seem a deceptively simple problem, **configuration changes are a potential source of instability**.

As a result, our approach to releasing and managing system and service configurations has evolved substantially over time. Today [there are] several models for distributing configuration files.

**• Minimal APIs**

**Writing clear, minimal APIs is an essential aspect of managing simplicity in a software system.**

The fewer methods and arguments we provide to consumers of the API, the easier that API will be to understand, and the more effort we can devote to making those methods as good as they can possibly be.

In software, less is more! A small, simple API is usually also a hallmark of a well-understood problem.

**• Cloud Elasticity**

- Elasticity refers to the ability to add resources to handle higher loads and remove resources when the load decreases.

- **Elasticity is achieved in the cloud by scaling resources such as VMs and databases.**

- Scaling in and out (horizontal scaling) refers to increasing and decreasing the number of resources devoted to a task - for example, increasing the number of VMs serving website users from 10 to 15.

- Scaling up and down (vertical scaling) refers to replacing existing resources with more or less powerful ones -- for example, replacing a web-server VM containing 2 cores and 4 GB of RAM with one containing 4 cores and 8 GB of RAM.

- Cloud platforms offer autoscaling mechanisms to scale resources in response to fluctuating demand without human intervention. There are two primary approaches to autoscaling:

/ Time-based autoscaling, also known as scheduled autoscaling, is most appropriate when loads are cyclical and predictable.

/ Metrics-based autoscaling can handle both predictable and unpredictable loads.

[

Time-based

Scale resources on a predetermined schedule. For example, if your organization's web site experiences the highest loads during working hours, configure autoscaling so that resources scale up or out at 8:00 a.m. every morning, and scale down or in at 5:00 p.m. each afternoon. Time-based scaling is sometimes referred to as scheduled scaling.

Metrics-based

If loads are less predictable, scale resources based on predefined metrics such as CPU utilization, memory pressure, or average request wait time. For example, if average CPU utilization reaches 70%, automatically bring additional VMs online, and when it goes back down to 30%, deprovision the extra VMs.

]

**Site Reliability Engineering (SRE)**

• SRE is what you get when you treat operations as if it's a software problem.

• **Site Reliability Engineering is a discipline that incorporates aspects of software engineering and applies them to infrastructure and operations problems.**

• The main goals are to create scalable and highly reliable software.

• While DevOps raise problems and dispatch them to Dev to solve, the SRE approach is to find problems and solve some of them themselves.

• DevOps and SRE developed roughly in parallel.

• DevOps and SRE are separate even though they attempt to address similar challenges.

• Automation and monitoring/observability are two best practices that are central to both DevOps and SRE

How SRE and DevOps differ is a topic still under considerable discussion in the field.

There are some broadly agreed upon differences, including:

• SRE is an engineering discipline that focuses on reliability, DevOps is a cultural movement that emerged from the urge to break down the silos typically associated with separate Development and Operations organizations.

• SRE can be the name of a role as in "I'm a site reliability engineer (SRE)", DevOps can't. No one, strictly speaking, is a "DevOps" for a living.

• SRE tends to be more prescriptive, DevOps is intentionally not so. Nearly universal adoption of continuous integration/continuous delivery and Agile principles are the closest it comes in this regard.

The two operations practices, DevOps and SRE, share a mutual love of monitoring/observability and automation (perhaps for different reasons).

**Tenets of SRE**

• An SRE team is responsible for the availability, latency, performance, efficiency, change management, monitoring, emergency response, and capacity planning of their service(s).

**Availability**

In general, for any software service or system, 100% is not the right reliability target because no user can tell the difference between a system being 100% available and 99.999% available. Thus, the marginal difference between 99.999% and 100% gets lost in the noise of other unavailability, and the user receives no benefit from the enormous effort required to add that last 0.001% of availability.

If 100% is the wrong reliability target for a system, what, then, is the right reliability target for the system? This actually isn't a technical question at all - it's a product question, which should take the following considerations into account:

**• What level of availability will the users be happy with, given how they use the product?**

**• What alternatives are available to users who are dissatisfied with the product's availability?**

**• What happens to users' usage of the product at different availability levels?**

**Latency**

**The time it takes to service a request.** It's important to distinguish between the latency of successful requests and the latency of failed requests. For example, an HTTP 500 error triggered due to loss of connection to a database or other critical backend might be served very quickly; however, as an HTTP 500 error indicates a failed request, factoring 500s into your overall latency might result in misleading calculations. On the other hand, a slow error is even worse than a fast error! Therefore, it's important to track error latency, as opposed to just filtering out errors.

**Performance**

Resource use is a function of demand (load), capacity, and software efficiency. SREs predict demand, provision capacity, and can modify the software. These three factors are a large part (though not the entirety) of a service's efficiency.

Software systems become slower as load is added to them. A slowdown in a service equates to a loss of capacity. At some point, a slowing system stops serving, which corresponds to infinite slowness. SREs provision to meet a capacity target at a specific response speed, and thus are keenly interested in a service's performance. **SREs and product developers will (and should) monitor and modify a service to improve its performance**, thus adding capacity and improving efficiency.

**Efficiency**

**Efficient use of resources is important any time a service cares about money.** **Because SRE ultimately controls provisioning, it must also be involved in any work on utilization**, as utilization is a function of how a given service works and how it is provisioned. It follows that paying close attention to the provisioning strategy for a service, and therefore its utilization, provides a very, very big lever on the service's total costs.

**Change Management**

**Best practices use automation** to accomplish the following:

**• Implementing progressive rollouts**

**• Quickly and accurately detecting problems**

**• Rolling back changes safely when problems arise**

This trio of practices effectively minimizes the aggregate number of users and operations exposed to bad changes. As a result, both release velocity and safety increase.

**Monitoring**

Monitoring is one of the primary **means by which service owners keep track of a system's health and availability**.

**There are three kinds of valid monitoring output:**

**• Alerts**

Signify that a human needs to take action immediately in response to something that is either happening or about to happen, in order to improve the situation.

**• Tickets**

Signify that a human needs to take action, but not immediately. The system cannot automatically handle the situation, but if a human takes action in a few days, no damage will result.

**• Logging**

No one needs to look at this information, but it is recorded for diagnostic or forensic purposes. The expectation is that no one reads logs unless something else prompts them to do so.

**Emergency Response**

Reliability is a function of mean time to failure (MTTF) and mean time to repair (MTTR). **The most relevant metric in evaluating the effectiveness of emergency response is how quickly the response team can bring the system back to health** - that is, the MTTR.

**Capacity planning**

Capacity planning should take both organic growth (which stems from natural product adoption and usage by customers) and inorganic growth (which results from events like feature launches, marketing campaigns, or other business-driven changes) into account.

**Several steps are mandatory in capacity planning:**

**• An accurate organic demand forecast, which extends beyond the lead time required for acquiring capacity**

**• An accurate incorporation of inorganic demand sources into the demand forecast**

**• Regular load testing of the system to correlate raw capacity (servers, disks, and so on) to service capacity**

**Key SRE Principles and Practices**

**• Service Level Indicator (SLI)**

**A quantifiable measure of service reliability.**

**In other words, a quantifiable measure of the reliability of your service from your users' perspective.**

**Good SLIs are a measurable analogy for user happiness.**

**SLIs - Metrics Over Time**

**- Request latency**

**- Batch throughput**

**- Failures per request**

[

SLI Menu provides guidelines for the types of SLIs that may be used when measuring a given critical user journey (CUJ).

SLI Menu

- To track the reliability of a request response interaction in a user journey, measure: availability, latency, and quality.

- For data processing: freshness, coverage, correctness and throughput.

- For storage: throughput and latency.

]

**• Service Level Objective (SLO)**

**A binding target for a collection of SLIs.**

**In other words, SLO sets a reliability target for an SLI over a period of time.**

**An SLO is fundamental tool for prioritizing reliability versus other features, and communicating the expectations of a service through objective data.**

**An SLO is an internal promise to meet customer expectations.**

**Being out of SLO must have consequences which redirect engineering effort towards making reliability improvements.**

**SLOs should capture the performance and availability levels that, if barely met, would keep the typical customer of a service happy:**

**"Meeting SLO Targets" => "Happy Customers"**

**"Sad Customers" => "Missing SLO Targets"**

**• Service Level Agreement (SLA)**

**A business agreement between a customer and service provider based on SLOs.**

**In other words, an external promise that comes with consequences.**

**An SLA describes the minimum level of service you promise to provide and what happens otherwise.**

**• Error Budget**

**An SLO implied acceptable level of unreliability.**

**An SLO implies an acceptable level of unreliability and this is a budget that can be allocated.**

[

This acceptable rate of failure is the error budget that can be actively spent - if it is not consumed by service downtime - on risky development activities like releasing new features making configuration changed.

A/B testing (a.k.a. bucket testing or split-run testing, is a user experience research methodology) etc.

]

**• Benefits of Error Budget**

**Common incentive for devs and SREs:**

Find the right balance between innovation and reliability.

**Dev team can manage the risk themselves:**

They decide how to spend their error budget.

**Unrealistic reliability goals become unattractive:**

These goals dampen the velocity of innovation.

**Dev team becomes self-policing:**

The error budget is a valuable resource for them.

**Shared responsibility for system uptime:**

Infrastructure failures eat into the error budget.

[

• Setting SLOs and SLIs

SLIs have a consistent format and range from 0-100%

The SLI Equation

SLI = (good events/valid events) x 100%

The proportion of valid events that were good.

For each critical user journey ranked by business impact:

1. Choose an SLI specification from the menu

2. Specify detailed SLI implementation

3. Validate that it doesn't have coverage gaps

4. Set SLOs based on past performance or business need

You should choose 3-5 SLIs per user journey.

SLI implementation includes event + success criteria + where or how you record the SLI.

SLO should include: target and a measurement window

Measuring SLIs sources: Log processing, Application Server Metrics, Front-end infrastructure Metrics, Synthetic Clients (Probers) or Data, Client Instrumentation

]

[

• Happiness test

Services need SLO targets that capture the performance and availability levels that, if barely met, would keep a typical customer happy.

]

[

• Reliability

The most important feature of any system is its reliability.

A service is reliable if it performs as its user expect.

Reliable enough: Acknowledging that a specific quantity of unreliability is acceptable provides a budget for failure that can be spent on developing and launching new features.

Improve reliability by reducing:

Time to detection

Time to resolution

Impact of outages

Frequency of outages

]

**• Toil**

**Work directly tied to running a service that is manual, repetitive, etc.**

**• Blameless Postmortem**

**A truly blameless postmortem culture results in more reliable systems.**

**SLIs, SLOs, and SLAs**

**• SLIs drive SLOs which inform SLAs.**

**• SLI = (Good Events / Valid Events) x 100%**

**• SLI ranges from 0% to 100%.**

**• 0% means nothing works.**

**• 100% means nothing is broken.**

**• This scale lends itself to the concept of an Error Budget.**

**• SLO is a target percentage.**

**• Error Budget = 100% - SLO**

**Error Budget**

**• An error budget is a way of tracking the aggregate reliability of a service over time.**

**• It is important to accompany this with a shared understanding of how the organization will react should the amount of acceptable unreliability be exceeded.**

When setting up an error budget for a service, it is important to decide (and document) how the organization will respond should it be exhausted.

There are many possible responses, you need to choose and stick to the ones that work best in your particular context.

There is no one right reaction for every one and every situation.

**• Product Management defines an SLO, which sets an expectation of how much uptime the service should have per quarter.**

**• The actual uptime is measured by a neutral third party.**

**• The difference between these two numbers is the "budget" of how much "unreliability" is remaining for the quarter.**

**• As long as the uptime measured is above the SLO - in other words, as long as there is error budget remaining - new releases can be pushed.**

**• If you exceed your error budget for a service,**

There are many ways to "spend" an excess error budget.

**Increasing the release cadency or feature velocity, deploying more beta features, conducting more disruptive testing**, and so on.

It is also fine (though this comes with its own set of problems) to do nothing and **simply operate at a greater level of reliability**.

All of these responses are situation specific and should be decided in your specific context.

**• If you exhaust your error budget for a service,**

**React in the service- or organizationally-specific way previously agreed upon when creating that error budget.**

**Toil**

**• Toil is the kind of work tied to running a production service that tends to be manual, repetitive, automatable, reactive, devoid of enduring value, and that scales linearly as a service grows.**

Characteristics of Toil

Manual

This includes work such as manually running a script that automates some task. Running a script may be quicker than manually executing each step in the script, but the hands-on time a human spends running that script (not the elapsed time) is still toil time.

Repetitive

If you're performing a task for the first time ever, or even the second time, this work is not toil. Toil is work you do over and over. If you're solving a novel problem or inventing a new solution, this work is not toil.

Automatable

If a machine could accomplish the task just as well as a human, or the need for the task could be designed away, that task is toil. If human judgment is essential for the task, there's a good chance it's not toil.21

Reactive rather than proactive

Toil is interrupt-driven and reactive, rather than strategy-driven and proactive. Handling pager alerts is toil. We may never be able to eliminate this type of work completely, but we have to continually work toward minimizing it.

No enduring value

If your service remains in the same state after you have finished a task, the task was probably toil. If the task produced a permanent improvement in your service, it probably wasn't toil, even if some amount of grunt work - such as digging into legacy code and configurations and straightening them out - was involved.

O(n) with service growth (Scales linearly as service grows)

If the work involved in a task scales up linearly with service size, traffic volume, or user count, that task is probably toil. An ideally managed and designed service can grow by at least one order of magnitude with zero additional work, other than some one-time efforts to add resources.

**• Task that needs human judgment may or may not be toil.**

We have to be careful about saying a task is "not toil because it needs human judgment."

We need to think carefully about whether the nature of the task intrinsically requires human judgment and cannot be addressed by better design.

**For example, one could build a service that alerts its SREs several times a day, where each alert requires a complex response involving plenty of human judgment.**

[Such a service is poorly designed, with unnecessary complexity.]

**The system needs to be simplified and rebuilt** to either eliminate the underlying failure conditions or deal with these conditions automatically.

**Until the redesign and reimplementation are finished, and the improved service is rolled out, the work of applying human judgment to respond to each alert is definitely toil.**

**• Eliminating Toil**

**- Identify and Measure Toi**l

/ Adopt a data-driven approach to identify and compare sources of toil.

/ Quantify the time saved by toil reduction projects.

**- Engineer Toil Out of the System**

/ Eliminate toil by changing the system.

/ Develop operationally friendly software that is not toilsome.

**- Reject the Toil**

/ Delay the toil so that tasks accumulate for batch or parallelized processing.

/ Working with toil in larger aggregates reduces interrupts and identifies patterns of toil.

**- Use SLOs to Reduce Toil**

/ Ignore certain operational tasks if doing so does not exceed the service's error budget.

**- Promote Toil Reduction as a Feature**

/ Couple strategy with the desirable features or business goals.

/ Complementary goal will compel customers to give up toil-generating systems.

**- Automate Toil Response**

/ Consider how to best mirror the human workflow in software.

/ Automation shouldn't eliminate human understanding of what's going wrong.

**• Reducing toil and scaling up services are the Engineering in SRE.**

**• Suggested Breakdown of Work for an SRE**

**- 50% engineering work and 50% operational work including toil-intensive work.**

**- 50% is seen as an upper bound on operational load, smaller would be better.**

**Blameless Postmortem**

**• The primary goals of writing a postmortem are to ensure that the incident is documented, that all contributing root cause(s) are well understood, and, especially, that effective preventive actions are put in place to reduce the likelihood and/or impact of recurrence.**

**• It is important to define postmortem criteria before an incident occurs so that everyone knows when a postmortem is necessary.**

• Blameless postmortems are a tenet of SRE culture.

• For a postmortem to be truly blameless, **it must focus on identifying the contributing causes of the incident without indicting any individual** or team for bad or inappropriate behavior.

• A blamelessly written postmortem assumes that everyone involved in an incident had good intentions and did the right thing with the information they had.

• Blameless culture originated in the healthcare and avionics industries where mistakes can be fatal.

• These industries nurture an environment where every **"mistake" is seen as an opportunity to strengthen the system**.

**• Triggers of Postmortem**

- User-visible downtime or degradation beyond a certain threshold

- Data loss of any kind

- Engineer intervention (release rollback, rerouting of traffic, etc.)

- A resolution time above some threshold

- A monitoring failure (which usually implies manual incident discovery)

**SRE Engagement Model**

**Phase 1: Architecture and Design**

SRE can influence the architecture and design of a software system in different ways:

**• Creating best practices**, such as resilience to various single points of failure, that a developer team can employ when building a new product

**• Documenting the dos and don'ts of particular infrastructure systems** (based upon prior experience) so developers can choose their building blocks wisely, use them correctly, and avoid known pitfalls

• Providing early engagement consulting to **discuss specific architectures and design choices** in detail, and to help validate assumptions with the help of targeted prototypes

• Joining the developer team and **participating in development work**

**• Codesigning part of the service**

Fixing architectural mistakes becomes more difficult later in the development cycle. Early SRE engagement can help avoid costly redesigns that become necessary when systems interact with real-world users and need to scale in response to service growth.

**Phase 2: Active Development**

As the product takes shape during active development, **SREs can start productionizing** the service - getting it in shape to be released into production. Productionalization typically **includes capacity planning, setting up extra resources for redundancy, planning for spike and overload handling, implementing load balancing, and putting in place sustainable operational practices like monitoring, alerting, and performance tuning**.

**Phase 3: Limited Availability**

As a service progresses toward Beta, the number of users, variety of use cases, intensity of usage, and availability and performance demands increase. **At this stage, SRE can help measure and evaluate reliability.** It is strongly recommend to **define SLOs before general availability (GA)** so that the service teams have an objective measure of how reliable the service is. The product team still has the option to withdraw a product that can't meet its target reliability.

During this phase, **the SRE team can also help scale the system by building a capacity model**, acquiring resources for upcoming launch phases, and automating turnups and in-place service resizing. **SRE can ensure proper monitoring coverage and help create alerts** that ideally match the upcoming service SLOs.

While service usage is still changing, the **SRE team can expect an increased amount of work during incident response and operational duties** because the teams are still learning how the service works and how to manage its failure modes. **It is recommend to share this work between the developer and SRE teams.** That way, the developer team gains operational experience with the service and the SREs gain experience with the service in general. Operational work and incident management will inform the system changes and updates the service owners need to make before GA.

**Phase 4: General Availability**

In this phase, the service has passed the Production Readiness Review and is accepting all users. While **SRE typically performs the majority of operational work**, the developer team should continue to field a small part of all operational and incident response work so they don't lose perspective on these aspects of the service. They might permanently include one developer in the on-call rotation to help the developers keep track of operational load.

In the early phase of GA, as the developer team focuses on maturing the service and launching the first batches of new features, it also needs to stay in the loop to understand system properties under real load. In the later stages of GA, the developer team provides small incremental features and fixes, some of which are informed by operational needs and any production incidents that occur.

**Phase 5: Deprecation**

No system runs forever. If and when a better replacement system is available, the existing system is closed for new users and all engineering focuses on transitioning users from the existing system to the new one. **SRE operates the existing system mostly without involvement from the developer team, and supports the transition with development and operational work.**

While SRE effort required for the existing system is reduced, SRE is effectively supporting two full systems. Headcount and staffing should be adjusted accordingly.

**Phase 6: Abandoned**

Once a service is abandoned, the developer team typically resumes operational support. **SRE supports service incidents on a best-effort basis.** For a service with internal users, **SRE hands over service management to any remaining users.**

**Phase 7: Unsupported**

There are no more users, and the service has been shut down. **SRE helps to delete references to the service in production configurations and in documentation.**

**SRE Team Implementations**

**Pros**

**• Easy to get started on an SRE journey without organizational change.**

**• Lets you test and adapt SRE practices to your environment at low cost.**

**Cons**

**• Time management between day-to-day job demands vs. adoption of SRE practices.**

**Recommended for: Organizations** **without the scale to justify dedicated SRE team staffing,** and/or organizations **experimenting with SRE practices before broader adoption.**

• **Types of SRE team implementations**

**1. Types of SRE Team Implementations**

**2. Kitchen Sink, a.k.a. "Everything SRE"**

**3. Infrastructure**

**4. Tools**

**5. Product/Application**

**6. Embedded**

**7. Consulting**

**1. Kitchen Sink, a.k.a. "Everything SRE"**

**This describes an SRE team where the scope of services or workflows covered is usually unbounded.** It's often the first (or only) SRE team in existence, and may grow organically, as it did when Google SRE first got started. We've since adopted a hybrid model, including the implementations listed below.

**Pros**

• **No coverage gaps between SRE teams**, given that only one team is in place.

**• Easy to spot patterns and draw similarities between services and projects.**

• SRE tends to act as a glue between disparate dev teams, creating solutions out of distinct pieces of software.

**Cons**

• There is usually a lack of an SRE team charter, or the charter states everything in the company as being possibly in scope, **running the risk of overloading the team.**

• As the company and system complexity grows, **such a team tends to move from being able to have deep positive impact on everything to making a lot more shallow contributions**. There are ways to mitigate this phenomenon without completely changing the implementation or starting another team.

• Issues involving such a team may negatively impact your entire business.

**Recommended for: A company with just a couple of applications and user journeys**, where adoption of SRE practices and demand for the role has outgrown what can be staffed without a dedicated SRE team, but where the scope remains small enough that multiple SRE teams cannot be justified.

**2. Infrastructure**

**These teams tend to focus on behind-the-scenes efforts that help make other teams' jobs faster and easier.** Common implementations include maintaining shared services (such as Kubernetes clusters) or maintaining common components (like CI/CD, monitoring, IAM or VPC configurations) built on top of a public cloud provider. This is different from SREs working on services related to products - i.e., customer-facing code written in house.

**Pros**

• **Allows product developers to use DevOps practices to maintain user-facing products** without divergence in practice across the business.

• **SREs can focus on providing a highly reliable infrastructure**. They will often define production standards as code and work to smooth out any sharp edges to greatly simplify things for the product developers running their own services.

**Cons**

• Depending on the scope of the infrastructure, issues involving such a team may negatively impact your entire business, similar to a Kitchen Sink implementation.

• Lack of direct contact with your company's customers can **lead to a focus on infrastructure improvements that are not necessarily tied to the customer experience**.

• **As the company and system complexity grows, you may be required to split the infrastructure teams**, so the cons related to product/application teams apply.

**Recommended for: Any company with several development teams,** since you are likely to have to staff an infrastructure team (or consider doing so) to define common standards and practices. It is common for large companies to have both an infrastructure DevOps team and an SRE team. The DevOps team will focus on customizing FLOSS and writing their own software (think features) for the application teams, while the SRE team focuses on reliability.

**3. Tools**

**A tools-only SRE team tends to focus on building software to help their developer counterparts measure, maintain, and improve system reliability or any other aspect of SRE work, such as capacity planning.**

One can argue that tools are part of infrastructure, so the SRE team implementations are the same. It's true that these two types of teams are fairly similar. In practice, tools teams tend to focus more on support and planning systems that have a reliability-oriented feature set, as opposed to shared back ends on the serving path that are normally associated with infrastructure teams.

As a side effect, there's often more direct feedback to infrastructure SRE teams; a tooling SRE team runs the risk of solving the wrong problems for the business, so it needs to work hard to stay aware of the practical problems of the teams tackling front-line reliability.

**The pros and cons of infrastructure and tools teams tend to be similar. Additionally**, for tools teams:

**Cons:**

• You **need to make sure that a tools team doesn't unintentionally turn into an infrastructure team, and vice versa.**

• There's a **high risk of an increase of toil and overall workload**. This is usually contained by establishing a team charter that's been approved by your business leaders.

Recommended for: Any company that needs highly specialized reliability-related tooling that's not currently available as FLOSS or SaaS.

**4. Product/Application**

In this case, **the SRE team works to improve reliability of a critical application or business area**, but the reliability of ancillary services such as batch processors is the sole responsibility of a different team - usually developers covering both dev and ops functions.

Pros

• Provides a **clear focus for the team's effort** and **allows a clear link from business priorities to where team effort is spent.**

Cons

• As the company and system complexity grows, new product/application teams will be required. The **product focus of each team can lead to duplication of base infrastructure or divergence of practices between teams**, which is inefficient and limits knowledge sharing and mobility.

**Recommended for: As a second or nth team for companies** that started with a Kitchen Sink, infrastructure, or tools team and have a key user-facing application **with high reliability needs that justifies the relatively large expense of a dedicated set of SREs.**

**5. Embedded**

**These SRE teams have SREs embedded with their developer counterparts, usually one per developer team in scope.** Embedded SREs usually share an office with the developers, but the embedded arrangement can be remote.

The work relationship between the embedded SRE(s) and developers tends to be project- or time-bounded. During embedded engagements, the SREs are usually very hands-on, performing work like changing code and configuration of the services in scope.

**Pros**

**• Enables focused SRE expertise to be directed to specific problems or teams.**

**• Allows side-by-side demonstration of SRE practices**, which can be a very effective teaching method.

**Cons**

**• It may result in lack of standardization between teams, and/or divergence in practice.**

**• SREs may not have the chance to spend much time with peers to mentor them.**

**Recommended for: This implementation works well to either start an SRE function, or to scale another implementation further.** When you have a project or team that needs SRE for a period of time, then this can be a good model. This type of team can also augment the impact of a tools or infrastructure team by driving adoption.

**6. Consulting**

This implementation is very **similar to the embedded implementation** described above. The **difference is that consulting SRE teams tend to avoid changing customer code and configuration** of the services in scope.

**Consulting SRE teams may write code and configuration in order to build and maintain tools for themselves or for their developer counterparts.** If they are performing the latter, one could argue that they are acting as **a hybrid of consulting and tools implementations.**

**Pros**

**• It can help with further scaling an existing SRE organization's positive impact by being decoupled from directly changing code and configuration.**

**Cons**

**• Consultants may lack sufficient context to offer useful advice.**

**• A common risk for consulting SRE teams is being perceived as hands-off (i.e., little incurred risk), given that they typically don't change code and configuration**, even though they are capable of having indirect technical impact.

**Recommended** for: It is recommend to wait to staff a dedicated SRE team of consultants until your company or complexity is considered to be large, and **when demands have outgrown what can be supported by existing SRE teams of other various implementations**. Keep in mind that it is recommended to staff one or a couple part-time consultants before you staff your first SRE team.

**SRE Team Lifecycles**

• **SRE Practices Without SREs**

Even if you don't have SREs, you can adopt SRE practices by using SLOs.

SLO are the foundations for SRE practices.

The performance of service relative to SLOs should guide the business decisions.

The following practices - can be achieved without even having a single SRE - are the crucial steps toward implementing SRE practices:

**- Acknowledge that 100% is not the right reliability target.**

**- Set a reasonable SLO target.** This SLO should measure the reliability that is most important to the users.

**- Agree on an error budget policy that will help defend the user experience.** Use the error budget to help guide:

/ Tactical actions to mitigate outages or to manage changes that return the system to a reliable state

/ Longer-term prioritization of work to make the system more reliable and use less of the error budget

**- Measure the SLO and commit to following the error budget policy.** This commitment requires agreement from company leadership.

Even if an organization doesn't have SRE staff, it is worthwhile to set SLOs for critical customer applications and to implement an error budget policy, if only because an implicit 100% SLO means a team can only ever be reactive.

This SRE principle allows you to make data-informed decisions about how to ensure the reliability of the application.

• **Starting an SRE Team**

An SRE team may be started in a number of ways:

**- Creating a new team as part of a major project**

**- Establishing a horizontal SRE team (a team of SREs consults across a number of teams)**

**- Converting an existing team (for example, an operations team)**

The approach that's best for the organization is highly situational. A team needs enough SREs to handle the operational tasks required to run the service.

Outside of a large SRE organization, a team likely can't embrace this concept from day one. This principle is open to interpretation and can be difficult to put into practice organizationally. The stages of building a team, using Tuckman's performance model and stages of forming, storming, norming, and performing:

**- Forming**

**/ The team assembled should have combined experience and expertise.**

**- Storming**

**/ Once assembled, the team needs to start working collaboratively.**

**- Norming**

**/ Norming entails reaching broad agreement on best practices for the organization's SRE teams.**

**- Performing**

**/ Partner on all architecture design and change with complete workload self-determination.**