Site Reliability Engineering or DevOps

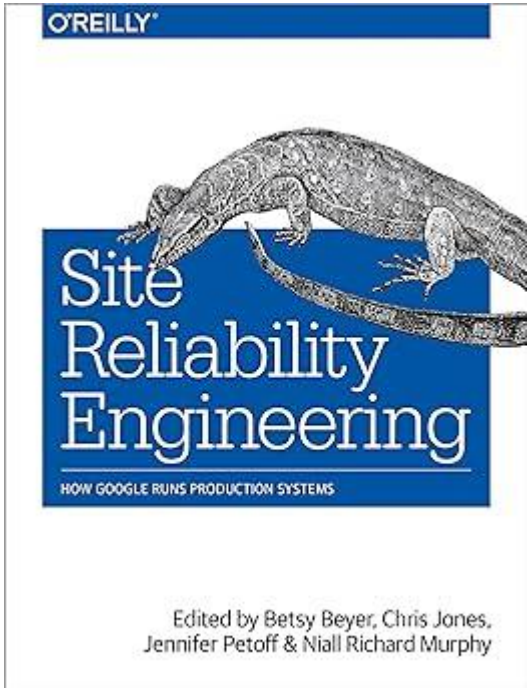
dr. Rok Piltaver, siječanj 2024

FAKULTET INFORMATIKE I DIGITALNIH TEHNOLOGIJA
SVEUČILIŠTE U RIJECI

Site Reliability Engineering

Literature

- Site Reliability Engineering book online
- SRE audiobook (text-to-speech)
- Lecture notes (i.e. book summary) on Merlin
- SRE book quotes by chapter (GitLab)
- Other SRE Google books



Traditional approach: sysadmins

Tasks:

- Assemble existing components and **deploy** to produce a service
- **Respond to events** and updates as they occur
- **Grow team** to absorb increased work as service grows

Pros:

- **Easy** to implement because it's **standard**
- **Large talent pool** to hire from
- Lots of **available software**

Cons:

- **Team size scales** with system load: manual change management and event handling
- Ops is fundamentally **at odds with dev**: ops resistance to changes, devs hiding launches

Google’s approach: SREs

Have **software engineers do operations** (candidates able to pass dev hiring + additional skills).

Results:

- SREs would be **bored by doing tasks by hand**
- Have the **skillset** necessary **to automate** tasks
- Same work as operations team but with **automation instead of manual labor**
- Google places a **50% cap on** the amount of **“ops” work** for SREs

Pros:

- **Cheaper** to scale
- **Circumvents devs/ops split**

Cons:

- **Hard to hire** for
- May **require management support** (stop releases for a quarter if error budget is depleted)

Site Reliability Engineer (SRE)

- Professional who applies SW **engineering principles to operational tasks**.
- Ensures **availability**, **reliability**, **scalability**, **efficiency**, and **performance**.
- Works on **complex**, **distributed** systems and services at a **massive scale**.
- **Balancing** the need for:
  - rapid **innovation** and feature development with the equally important
  - requirement of maintaining a **reliable and robust** production environment.
- Create **sustainable and scalable infrastructure**:
  - **minimizes** service **disruptions**
  - provides a **positive user experience**

SRE Activity Categories:

Software engineering: writing code (automation scripts, creating tools, adding features for scalability and reliability)

Systems engineering: consulting on architecture, design, and productionization for dev teams and configuring production systems with **lasting improvements** (e.g. monitoring, load balancing and server configuration, tuning OS parameters).

Toil: work directly tied to running a service that is repetitive, manual, etc.

Overhead: administrative work (hiring, HR paperwork, team/company meetings, bug queue hygiene, peer reviews and self-assessments, and training courses).

Responsibilities of an SRE Team

- **Define** and **measure SLI** and **SLO** to quantify reliability and performance.
- **Infrastructure design** for scalable, reliable, and fault-tolerant system.
- Conduct **performance testing** & analyze system behavior under different conditions.
- **Security**: implement security best practices and ensure systems and data protection.
- **Disaster recovery planning**: test disaster recovery plans intended for major failures.
- Maintain comprehensive **documentation** for systems, processes, and procedures.
- **Deployment and release engineering**:
  - ensure **smooth and reliable** deployments
  - implement strategies for **canary releases, feature flags**, and **rollback** mechanisms
- **Change management**:
  - infr. & sys. **configuration changes** without degrading reliability or performance
  - **risk** assessment, change **approval** process, **rollback** planning,
  - communicate **upcoming changes**, **monitoring** and observability, post-change **evaluation**

Responsibilities of an SRE Team

- **Automation**:
  - Develop **automation tools** to manage systems efficiently.
  - **Automate** repetitive **operational tasks** to minimize manual intervention.
- Implement **monitoring and alerting** systems to detect issues proactively.
- **Capacity** and performance **management** and **planning**:
  - Monitor and **manage system capacity** to ensure optimal performance.
  - Ensure that systems can **handle increased loads** without compromising performance.
  - **Optimize resource utilization** and identify areas for improvement of efficiency.
  - Analyze system usage patterns and **plan for** scalability to meet **future demands**.
- **Incident response** and management:
  - Participate in **on-call rotations** to **address critical issues** and mitigate incidents promptly.
  - Conduct **post-mortems** to analyze and learn from incidents.
  - **Implement improvements** based on post-mortems to prevent future incidents.

Service Level Indicators, Objectives and Agreements

Service Level Indicator (SLI)

SLI is a carefully defined quantitative measure of some aspect of the level of service that is provided.

Examples:

- request **latency** (how long it takes to return a response to a request)
- **availability** (fraction of the time that a service is usable).
- **error rate** (percentage requests that result in errors or failures)
- **throughput** (the amount of data that can be processed in a time period)
- **end-to-end latency** (the total time a user waits for the response)
- **durability** (share of data with preserved long-term availability and integrity)

Service Level Objectives (SLO)

SLO is a (range of) target value(s) for a service level that is measured by an SLI.

- Reflect **what users care about**.
- Can and should be a major **driver in prioritizing work** for SREs & developers.
- Publishing SLOs to users **sets expectations** about how a service will perform.
- SLOs should specify:
  - **how** they're measured and
  - **conditions** under which they're valid.
  - Example: 99% (averaged over 1 minute) of GET RPC calls will complete in less than 100 ms (measured across all the backend servers).

"Defend the SLOs you pick: if you can't ever win a conversation about priorities by quoting a particular SLO, it's probably not worth having that SLO."

Service Level Agreements (SLA)

SLA are an explicit or implicit contract with your users that includes consequences of meeting (or missing) the SLOs they contain.

- The **consequences** are most easily recognized when they are **financial**: a rebate or a penalty - but they can take other forms.
- Easy way to tell the difference between an SLO and an SLA:
  - ask: "what happens if the SLOs aren’t met?"
  - if there is no explicit consequence, then you are looking at an SLO.

Principles of SRE

Ensuring a Durable Focus on Engineering

- **50% ops cap**: ops work is redirected to product teams on overflow
- Target **max 2 events per** 8-12 hour on-call **shift**
- Blameless **postmortems** for all serious incidents
- Provides **feedback** mechanism **to product team** as well as keeps load down
- Engineering work **makes ops work easier**, faster, and more reliable through:
  - Automation
  - Improved reliability

Move Fast Without Breaking SLO

The goal isn't zero outages:

- difference between 5 9s and 100% **isn't noticeable to users**
- but **requires tremendous effort**.

Error budget:

- quantified **acceptable level of** service **disruptions**/errors within a timeframe
- acknowledges the **trade-off** between **reliability** and feature **velocity.**
- **aligns SRE and product devs**: spend budget to get max feature velocity.
- allows **objective discussion**: how phased rollouts and canary experiments can maintain tolerable levels of errors

Monitoring

- Should **never require a human to interpret** any part of the alerting domain.
- Three valid kinds of monitoring output
  - **Alerts**: human needs to take action **immediately**
  - **Tickets**: human needs to take action **eventually**
  - **Logging**: no action needed, for diagnostic or **forensic** purposes only

Emergency Response

Reliability is a function of:

- Mean-time-to-failure (MTTF) - how reliable is the system and
- Mean-time-to-recovery (MTTR) - how good is SRE

Playbooks works better than hero generalists who can respond to everything.

Humans add latency: **automated** systems have higher availability (lower MTTR)

Change Management

70% of **outages due to changes** in a live system.

Mitigation:

- implement **progressive rollouts**,
- **monitoring**
- **rollbacks**
- **remove humans** from the loop

Demand Forecasting and Capacity Planning

Ensure **sufficient capacity** and redundancy to serve projected **future demand** with the required availability.

Steps in capacity planning:

- **organic demand** forecast (beyond time required for acquiring capacity)
- incorporation of **inorganic demand** sources (feature launches, marketing campaigns) into the demand forecast
- regular **load testing** of the system to correlate raw capacity (servers, disks) to service capacity

Provisioning

Process of setting up and preparing:

- **computing resources** (physical or virtual servers, storage, and networking),
- **services** (apps, cloud services, DBs, web servers, middleware, user accounts)
- **infrastructure** (network components)
- to ensure they are **ready for use**.

- Adding capacity is **riskier than load shifting**, since it often involves:
  - spinning up **new instances**/locations,
  - **making significant changes** to existing systems (config files, load balancers)
- Should be done quickly and **only when necessary**: meet response time goal.
- **Overprovisioning** costs money (not knowing what you actually need).

Efficiency and Performance

Utilization refers to the extent to which a system or its components are effectively used or occupied over a period of time.

- Measures **efficiency** and **resource usage** of a system - how well the available resources are being utilized to support the desired level of service.
- Is a function of:
  - **demand** (load of the system),
  - **capacity** (capability to handle demand),
  - **software efficiency**
- Translates directly to **money**.
- Load slows down systems.

Principles: Embracing Risk

Embracing Risk

Reliability isn't linear in cost - additional increment of reliability can cost 100x more

- Cost associated with **redundant equipment**
- Cost of building out **features for reliability** as opposed to "normal" features
- Goal: make systems **reliable enough, but not too reliable**!

Measuring service risk: metric represents property of system to optimize:

- **aggregate availability** = successful requests / total requests (good approx.)
- set **quarterly targets**

Risk tolerance of services is usually not objectively obvious

- Product owners and SRE **translate** business objectives into explicit objectives
- What **level of availability** is **required**? Below background error rate of ISP.
- How can **service cost** help locate a service on the **risk continuum**?

Motivation for Error Budgets

Tensions between product development and SRE:

- product **velocity** (push new code as quickly as possible) vs
- **reliability** (push back against a high rate of change).

Typical tensions:

- **Software fault tolerance**: too little = brittle, unusable product; too much = product no one wants to use.
- **Testing**: not enough = embarrassing outages, privacy data leaks; too much = you might lose your market.
- **Push frequency**: how much should we work on reducing that risk of a push (change to running software or its configuration), versus doing other work?
- **Canary duration and size**: test a new release on a small subset of a typical workload. How long do we wait, and how big is the canary?

Forming Error Budget

The two teams jointly **define a quarterly error budget** based on SLO to provide a clear, objective metric that determines how unreliable the service is allowed to be.

This removes the politics, negotiating skills, fear/hope from negotiations between the SREs and the product developers when deciding how much risk to allow.

Process:

1. Product Management **defines SLO** (successfully serve 99.999% of queries).
2. Uptime **measured** by monitoring system (problem caused 0.0002% queries to fail).
3. The difference is the **remaining budget** of unreliability (failure rate of 0.0008%).
4. New releases can be **pushed if** there is some **error budget remaining**,
   i.e. if uptime > SLO (e.g. the problem spent 20% of error budget).

Benefits of an Error Budget

- Provides a **common incentive** that allows both product development and SRE to focus on **finding the right balance** between innovation and reliability.
- **Manage release velocity**: slow down releases when error budget is close to 0, if error budget is spent, releases are temporarily halted while additional resources are invested in system testing and development to make the system more resilient and improve its performance.
- **Dev team** becomes self-policing and **manages their own risk**: large budget allows more risk, when it's drained, developers will push for more testing or slower velocity.
- It highlights the **costs of overly high reliability targets**, in terms of both inflexibility and slow innovation. The team can elect to **loosen the SLO** in order to speed up launching new features.

Principles: Eliminating Toil

What Is Toil

Toil is the kind of work tied to running a production service that tends to be:

- manual,
- repetitive,
- automatable,
- tactical,
- devoid of enduring value, and that
- scales linearly as a service grows.

- Not just “work I don’t want to do”.
- Not every task deemed toil has all these attributes.
- Example: (semi)manual step-by-step deployment process.

Why Less Toil Is Better

Toil **at least for ~1/3** of an SRE's time:

- **on-call** shifts and
- **interrupt handling** (non-urgent service-related messages and emails)
- 1 week of primary + 1 week of secondary **on-call** in 6-person rotation cycle

50+% of SRE's time should be spent **on engineering project** work that will:

- **reduce** future **toil** or
- add **service features** (improve reliability, performance, utilization = reduce toil)
- SRE organization **scales up sublinearly** with service size
- Toil > 50% is a sign that the **manager should spread toil** load more evenly

Is Toil Always Bad?

Toil isn't always bad: it's fine in small doses:

- Can produce a **sense of accomplishment.**
- Predictable and repetitive tasks (low-risk/low-stress) can be **calming.**

Why toil is bad:

- **Tends to expand** if left unchecked and can quickly fill 100% of everyone's time.
- **Low morale**: toxic if experienced in large quantities (burnout, boredom, discontent).
- **Career stagnation:** spending too little time on projects (complain loudly).
- **Slows progress**: excessive toil (manual work, firefighting) -> slow feature velocity
- **Promotes attrition**: too much toil motivate best engineers quit.
- B**reach of faith**: new SRE hires with the promise of project work will feel cheated.

What Qualifies as Engineering?

- Engineering work is **novel** and intrinsically **requires human judgment**.
- It produces a **permanent improvement** in your service, and is guided by a strategy.
- It is frequently **creative** and **innovative**
- Taking a design-driven approach to solving a problem: **the more generalized, the better**.

Principles: Automation

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

*“Automation is a **force multiplier**, not a panacea.”*

*“Failure is inevitable - optimize to **recover quickly** using automation.”*

*“The most functional **tools** are usually written **by those who use them**.”*

Value of Automation

- **Consistency**: manual tasks lead to **mistakes**, oversights, and reliability problems, therefore consistency is the primary value of automation.
- **A platform**: automatic systems provide an extendable platform that can be **applied to more systems**, **centralizes mistakes** (bug fixed in the code will be fixed once and forever) and can **run continuously**/frequently.
- **Faster Repairs**: automation that resolves common faults reduces mean time to repair resulting in **higher availability** and increased developer velocity.
- **Faster non-repair actions**: in the infrastructural situations (e.g. failover or traffic switching) humans don't usually react as fast as machines.
- **Time savings**: effort saved in not requiring a task to be performed manually versus the effort required to write it.

Hierarchy of Automation Classes

- **No automation**: database master is failed over manually between locations.
- **Externally** maintained **system-specific** automation: SRE’s failover script.
- **Externally** maintained **generic** automation: add DB support to a "generic failover" script that everyone uses.
- **Internally** maintained **system-specific** automation: DB ships with its own failover script.
- Systems that **don’t need any automation**: DB notices problems, and automatically fails over without human intervention.

Principles: Monitoring

Monitoring

Monitoring means collecting, processing, aggregating, and displaying **real-time quantitative data about a system**: query and error counts, processing times, etc.

Types of monitoring:

- Lots of **white-box monitoring**: based on metrics exposed by the internals of the system, (logs, JVM Profiling Interface, service emitting internal statistics).
- Some **black-box monitoring** for critical stuff: externally visible behavior as a user would see it.

Monitoring system should address two questions:

- **what**'s broken (indicates the symptom),
- and **why** (indicates a possibly intermediate cause)

Why Monitor?

- Analyze **long-term trends**: size, speed and growth of database, DAU growth.
- Compare over time or do **experiments**:
  - Memcache hit rate improvement due to an extra node,
  - Is my site slower than it was last week?
- **Alerting**: human notification pushed to bug/ticket queue, email alias, or pager.
- Building **dashboards**: web app that provides a summary view of a service's core metrics important for the user.
- **Debugging**: e.g. what caused the increase in latency?

Four Golden Signals

- **Latency**: the time it takes to service a request.
- **Traffic**: how much demand is there:
  - requests per second for a web service
  - concurrent sessions for audio streaming system
  - transactions per second for a key-value storage
- **Errors**: the rate of requests that fail.
- **Saturation**: how "full" your service is:
  - in a memory-constrained system, show memory
  - in an I/O-constrained system, show I/O
  - e.g. database will fill it’s hard drive in 4 hours

Setting Reasonable Expectations

- **Never** trigger an alert simply because "something seems **a bit weird**."
- Effective alerting systems have **good signal and very low noise**.
- **Page** must be very simple to understand and robust (a clear failure):
  - should be able to react with a **sense of urgency** (max a few times a day)
  - page should be **actionable**
  - page response should **require intelligence**
  - should be about problem/event that **hasn’t been seen before**
- Worrying about the **tail of the distribution** (p99): collect request counts bucketed by latencies (a histogram with exponential boundaries), rather than actual latencies.
- Track **error latency** (slow error is even worse than a fast error), as opposed to just **filtering out errors** (factoring errors into overall latency gives misleading results).
- **Avoid “magic” systems**.
- **10-20% of SRE team** dedicated to building and maintaining monitoring.

Principles: Release Engineering

Release Engineering

Release engineers work with devs and SREs to define how software is released.

- **Self-service model**
  - develop best practices & tools that allow devs to run their own release processes
  - automated build system and deployment tools make releases truly automatic
- **High Velocity**: roll out customer-facing features as quickly as possible,
  - Frequent releases - fewer changes between versions - easier testing and troubleshooting
  - Could be “push on green” and deploy every build or hourly builds and deploys.
- **Hermetic builds**:
  - Tools ensure that building same revision number always gives identical results.
  - Builds depend on fixed ver. of build tools (compilers), and dependencies (e.g. libraries).
- **Enforcement of Policies and Procedures**: who can perform operations:
  - approving source code changes, creating a new release, approving cherry picks, deploying a new release, making changes to a project’s build configuration

Principles: Simplicity

Simplicity

- Software simplicity is a **prerequisite to reliability**.
- **Virtue of boring**!
- Push back when **accidental complexity** is introduced.
- Code is a liability: **remove dead code** or other bloat (negative lines of code).
- **Minimal APIs**: smaller and clear APIs are easier to test and more reliable.
- **API versioning**: use old version while devs upgrade to new one in a safe way.
- **Modularity**: ability to make changes to parts of the system in isolation.
- **Small releases** are easier to measure (don't released 100 changes together).

"Perfection is finally attained not when there is no longer more to add, but when there is no longer anything to take away."