Engineer

Error budget

the share of failures a reliability target still allows

What it is

An error budget is the amount of failure a service is allowed over a stated period, worked out from its service level objective. If the objective says that 99.99% of requests must succeed this quarter, the remaining 0.01% is the budget: requests that may fail, by accident or by a risk taken on purpose, without anyone having broken a promise. It is counted in the same unit as the objective, usually failed requests, and it is spent the moment users meet a failure.

The idea comes from Google's Site Reliability Engineering book. In its chapter on embracing risk, product management sets the objective, the monitoring system acts as a neutral third party that measures what actually happened, and the gap between the two is the unreliability still left for the quarter. Product development and SRE agree that budget together, in advance. The book's example sits at the demanding end of the scale: a service that must answer 99.999% of queries has a budget of 0.001%, so a single problem that fails 0.0002% of the quarter's queries has spent a fifth of it.

The budget does not ask whose fault a failure was. The book is explicit that a network outage or a failed datacenter eats into it exactly as a bad release does, and that the rest of the quarter may then see fewer pushes. Users felt the failure, so it counts. Who has to act on it is a separate question, and the answer belongs in a written policy agreed before anything goes wrong.

Use the concept in a decision

Put "What is an error budget?" to work in a real decision. We can help relate the concept to the system you are building or buying. Monitoring and alerts that speak up while a fault is still small, and the figures that show how often it stays that way.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Built the company's first observability dashboard — the company's first view of its own systems

Built the company's first observability dashboard Get in touch →

Why it matters

The people who ship features and the people who keep a service running want different things, and without a shared number the argument between them is won by whoever negotiates better. The SRE book says such decisions should rest on neither politics nor fear nor hope, and recalls Google SRE's unofficial motto: "Hope is not a strategy." The budget replaces the argument with a figure both sides accepted beforehand. While budget remains, releases go out; when it is spent, they pause while the effort goes into testing and resilience instead. The book also describes gentler versions of the same loop, such as slowing releases down or rolling them back as the budget nears empty, and in a footnote calls the plain on-or-off version bang/bang control.

The more interesting effect is on behaviour. When the budget is large, developers can take risks: push more often, try the experiment. When it is nearly drained, the book observes, the developers themselves ask for more testing and slower pushes, because nobody wants to stall their own launch. The team becomes self-policing. The book attaches one condition: this works only if SRE really has the authority to stop launches once the objective is broken.

Releases are what the budget mostly governs because releases are what mostly breaks things: the SRE Workbook's example policy puts changes at roughly 70% of Google's outages. That makes the budget a guide to where effort should go. The Workbook's chapter on implementing SLOs works through a service with a 97% objective and a four-week budget of 109,897 failed requests. A bad API release that failed every request for four hours cost 14,066 of them, about 13%. A database server that died and took 20 hours to restore from backup cost an estimated 72,000, or 65%. The server failure looks far worse, but the example company sees one of those in five years against two or three bad releases a year, so on average the releases cost about twice as much budget. Fixing the release process pays off first.

A budget that is never spent says something too. The book notes that it exposes what an overly high target costs in inflexibility and slow innovation, and that a team struggling to launch can choose to loosen its objective, which enlarges the budget.

What it looks like in practice

Spending is a speed as well as a total. The Workbook's chapter on alerting calls that speed the burn rate: how fast, relative to the objective, the service is consuming its budget. At a burn rate of 1 the budget runs out exactly at the end of the window. For a 99.9% objective over 30 days, its table works out like this:

```
Burn rate   Error rate   Budget gone after
1           0.1%         30 days
2           0.2%         15 days
10          1%           3 days
1,000       100%         43 minutes
```

Alerting on burn rate beats paging whenever errors pass 0.1%, which would wake someone for every brief spike. The Workbook recommends checking several burn rates at once, each over a long window and a short one. Its starting point for a 99.9% objective pages when 2% of the 30-day budget goes in one hour (a burn rate of 14.4) or 5% goes in six hours (a burn rate of 6), and opens a ticket when 10% goes in three days (a burn rate of 1). The short window, a twelfth of the long one, makes the alert stop soon after the problem does. In the Workbook's example, the errors end and five minutes later the short-window average is back under the threshold, while the one-hour average stays above it for a full hour.

As a Prometheus rule file, the two paging conditions read as follows. The ratios are recording rules named the way the Workbook names them, each holding the share of failed requests over the window in its name:

```
groups:
  - name: orders-error-budget
    rules:
      - alert: OrdersErrorBudgetBurn
        expr: |
          (
            job:slo_errors_per_request:ratio_rate1h{job="orders"}
              > (14.4 * 0.001)
            and
            job:slo_errors_per_request:ratio_rate5m{job="orders"}
              > (14.4 * 0.001)
          )
          or
          (
            job:slo_errors_per_request:ratio_rate6h{job="orders"}
```

```
          > (6 * 0.001)
        and
        job:slo_errors_per_request:ratio_rate30m{job="orders"}
          > (6 * 0.001)
      )
    labels:
      severity: page
```

Follow one bad deploy through it. At 14:00 a release starts failing 2% of order requests, a burn rate of 20. The five-minute ratio passes the 1.44% line within four minutes, but the one-hour ratio reaches it only after the errors have run for about 43 minutes, so the page goes out at around 14:43. By then exactly 2% of the month's budget is gone, which is the trade the rule makes: a short blip wakes nobody, and a sustained burn cannot hide. Roll back at 14:50 and the five-minute ratio falls below the line within two minutes, ending the page; the one-hour ratio alone would have kept it firing until about 15:07.

Keeping paging proportionate is the same concern behind the company's first-party error monitoring and tracing, from 2025. It detects spikes and regressions with a cooldown, so that one bad deploy does not page anyone forty times, its alerting rules were fitted to that platform rather than to a generic one, and one of its 11 operator views is given to service level objectives.

Where it goes wrong

Counting a failure is confused with blaming someone for it. Everything users feel spends the budget, but not everything should stop the team's own releases. The Workbook's example policy lets feature work continue when a company-wide network problem caused the outage, or when another team's service did and that team has frozen its own releases; when the team's own bug or procedure spent the budget, reliability comes first. For an outage caused by another team's dependency, the Implementing SLOs chapter sets out two schools of thought, carry on or freeze anyway, and says the freeze makes users happier. Choose in advance, write the choice into the policy, and name who settles a disagreement about the arithmetic; the example policy sends it to the CTO.

Errors land in the wrong category. A failed load test or a penetration tester's rejected requests spend budget although no user was affected, and the example policy names miscategorised errors as a way budget disappears for nothing. The reverse is worse: real failures filed as something harmless keep the service inside

its objective on paper while users suffer, and the same policy makes reliability work mandatory when that is found. Keep traffic the objective does not cover out of the indicator, and check the classification after any incident whose cost looked wrong.

A large incident is paid for and forgotten. A freeze buys time, not understanding. The example policy ties the two together: a single incident that spends more than 20% of the four-week budget requires a postmortem with at least one action item of the highest priority, and one class of outage that spends more than 20% in a quarter must appear on the next quarter's plan.

The alert can never fire. Thresholds copied from the 99.9% table do not carry over to other targets. At 90% availability, the Workbook points out, a total outage spends only 1.4% of the budget in an hour, so a rule that pages at 2% an hour stays silent through it. At the other extreme, a 99.999% monthly target is exhausted by a total outage in 26 seconds, faster than many monitoring systems even collect a sample. Tune the windows and thresholds to the target, and for very high targets accept the Workbook's conclusion that only design defends them: roll a change out to 1% of users first, and the same failure leaves 43 minutes before the budget is gone.

Low traffic turns one failure into a page. At ten requests an hour, a single failed request is a 10% error rate for that hour. Against a 99.9% objective, the Workbook notes, that one request pages at once, because it spends 13.9% of the 30-day budget: the whole month allows only seven failures. The Workbook's remedies are to generate artificial traffic, to combine several small services into one for monitoring, or to change the product so that one failed request matters less.

The target is raised before the service can meet it. A team that needs a tighter objective and adopts it straight away lives in permanent emergency, with the budget always spent. The Workbook suggests an aspirational objective instead: measured and tracked beside the current one, but named in the policy as requiring no action until the service is ready for it.

Further reading

- Embracing Risk — chapter 3 of Google's Site Reliability Engineering book, where the error budget is introduced as a way to settle release decisions with data.

- Alerting on SLOs — the SRE Workbook on burn rates, multiwindow alerts and low-traffic services.
- Example Error Budget Policy — a complete policy: when releases stop, which outages are exempt, and when a postmortem is required.
- Implementing SLOs — how to compute a budget from an objective and use it to choose between reliability investments.
- Alerting rules — the Prometheus documentation for the rule file format used above.

Glossary

Choose for your actual constraints

Bring the deployment path, the operational risks and the people on call. We can scope a change with checks, a recovery route and clear ownership.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is an error budget?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/error-budget/