Most organizations that claim to have SLOs actually have dashboards with arbitrary thresholds that nobody looks at. We know because we audit them. The number on the dashboard says "99.95% availability" but nobody can tell you what happens when it drops below that number. There is no error budget policy, no automated alerting tied to burn rate, and no meeting where the SLO data influences what the team works on next. That is not an SLO program. That is decoration.

This post covers how we implement SLOs that actually change how teams make decisions, using tools we deploy in production every week.

Why Most SLO Implementations Fail

Three reasons, in order of frequency:

1. Vanity SLOs. The team picks "99.99% availability" because it sounds good. They have no idea what 99.99% actually requires (52 minutes of downtime per year, including deployments, dependency failures, and infrastructure issues). When the SLO is impossible to meet, people stop paying attention to it.

2. No error budget policy. An SLO without an error budget policy is a number without consequences. The error budget is the acceptable amount of unreliability. The policy defines what happens when you have budget remaining (ship features) and what happens when budget is exhausted (stop feature work, focus on reliability). Without this policy, the SLO is informational. Informational things get ignored.

3. Nobody looks at them. SLOs exist in a Grafana dashboard that three people bookmarked six months ago. There is no regular review cadence. No one brings

SLO data to sprint planning or roadmap discussions. The data exists but does not flow into decisions.

Start with User Journeys, Not Infrastructure Metrics

The first mistake teams make is measuring what is easy instead of what matters. CPU utilization, memory usage, and disk I/O are infrastructure metrics. They tell you about the health of your machines, not the experience of your users.

Start by listing the critical user journeys:

- **Login flow.** User submits credentials, receives a session token.
- **Search.** User enters a query, receives relevant results.
- **Checkout.** User submits payment, order is confirmed.
- **API integration.** Partner system sends a request, receives a valid response within SLA.

Each of these journeys has a measurable SLI (Service Level Indicator) that directly reflects user experience. A server at 90% CPU utilization might be serving requests perfectly. A server at 30% CPU might be returning 500 errors because of a database connection leak. Infrastructure metrics do not tell you what the user is experiencing.

Choosing the Right SLI Type

There are four SLI types that cover nearly every use case:

Availability: The proportion of valid requests that are served successfully. This is the most common SLI. "Did the request succeed or fail?"

```
SLI = (total requests - error requests) / total requests
```

Latency: The proportion of valid requests served faster than a threshold. "Was the response fast enough?"

```
SLI = requests faster than 300ms / total requests
```

Throughput: The proportion of time the system processes at or above a minimum rate. Less common but important for data pipelines and batch processing.

Correctness: The proportion of valid requests that produce a correct result. Hardest to measure but critical for systems where wrong answers are worse than no answers (financial calculations, search relevance).

For most web services, you want one availability SLI and one latency SLI per critical user journey. Do not over-index. Three to five SLOs for your entire service is usually the right number. More than that and you are back to dashboard decoration.

Setting Honest SLO Targets

Here is what each nines level actually means in practice:

TARGET	MONTHLY DOWNTIME	WHAT IT REQUIRES
99%	7h 18m	Basic monitoring, manual response
99.5%	3h 39m	On-call rotation, basic automation
99.9%	43m 50s	Automated failover, redundant systems, fast deploys
99.95%	21m 55s	Multi-region, canary deployments, comprehensive testing

TARGET	MONTHLY DOWNTIME	WHAT IT REQUIRES
99.99%	4m 23s	Active-active multi-region, zero-downtime deploys, near-instant detection

Most services should start at 99.5% or 99.9%. If you are not currently measuring your reliability, you have no basis for setting a target. Run your SLI measurement for 30 days before committing to a target. Use the baseline data to set a target that is slightly above your current performance. This gives you a meaningful but achievable goal.

When to use lower targets: Internal tools, non-revenue-critical services, and services with offline alternatives. Setting everything to 99.99% is wasteful. A 99% SLO on an internal admin dashboard is perfectly reasonable and saves engineering effort for services that actually need high reliability.

Error Budgets as a Decision-Making Tool

The error budget is what makes SLOs actionable. If your SLO is 99.9% availability over a 30-day rolling window, your error budget is 0.1% - roughly 43 minutes of downtime or the equivalent number of failed requests.

The error budget policy should be explicit and agreed upon by engineering leadership and product management:

When error budget is healthy (>50% remaining):

- Ship new features at normal velocity.
- Deploy with standard canary process.
- Take on technical debt reduction if there is slack.

When error budget is at risk (25-50% remaining):

- Increase canary bake time from 15 minutes to 1 hour.
- Require rollback plans for every deployment.
- Prioritize one reliability improvement per sprint.

When error budget is exhausted (<25% remaining or spent):

- Halt feature development until budget recovers.
- All engineering effort goes to reliability improvements.
- Post-incident reviews for any incident that consumed budget.
- Product manager is informed and roadmap adjusts.

This is the cultural shift. Product managers need to understand that the error budget is finite and that burning it on incidents means features get delayed. Engineers need to understand that having budget remaining means they should ship, not hoard reliability.

Implementing SLOs with Prometheus

The core of our SLO implementation is Prometheus recording rules that calculate SLI metrics over multiple windows. Here is a concrete example for an HTTP availability SLO:

```
# Prometheus recording rules for HTTP availability SLO
groups:
  - name: slo:api-availability
    interval: 30s
    rules:
      # Total request rate
      - record: slo:api_requests:rate5m
        expr: sum(rate(http_requests_total{job="api-server"}[5m]))

      # Error request rate (5xx responses)
      - record: slo:api_errors:rate5m
        expr: sum(rate(http_requests_total{job="api-server",
status=~"5.."}[5m]))

      # Error ratio over multiple windows for multi-burn-rate alerting
      - record: slo:api_errors:ratio_rate5m
        expr: slo:api_errors:rate5m / slo:api_requests:rate5m

      - record: slo:api_errors:ratio_rate30m
        expr: |
          sum(rate(http_requests_total{job="api-server", status=~"5.."}
[30m]))
          /
          sum(rate(http_requests_total{job="api-server"}[30m]))

      - record: slo:api_errors:ratio_rate1h
        expr: |
          sum(rate(http_requests_total{job="api-server", status=~"5.."}
[1h]))
          /
          sum(rate(http_requests_total{job="api-server"}[1h]))

      - record: slo:api_errors:ratio_rate6h
        expr: |
          sum(rate(http_requests_total{job="api-server", status=~"5.."}
[6h]))
          /
          sum(rate(http_requests_total{job="api-server"}[6h]))

      - record: slo:api_errors:ratio_rate1d
        expr: |
          sum(rate(http_requests_total{job="api-server", status=~"5.."}
[1d]))
          /
          sum(rate(http_requests_total{job="api-server"}[1d]))

      - record: slo:api_errors:ratio_rate3d
        expr: |
          sum(rate(http_requests_total{job="api-server", status=~"5.."}
[3d]))
          /
          sum(rate(http_requests_total{job="api-server"}[3d]))
```

The multi-window approach is critical. A 5-minute error spike might not matter. A sustained 6-hour elevation is a real problem. The alerting rules use multiple burn rates to distinguish between the two.

Multi-window, multi-burn-rate alerting:

```yaml
  - name: slo:api-availability:alerts
    rules:
      # Fast burn: 14.4x budget consumption over 1 hour
      # Pages immediately - something is very wrong right now
      - alert: APIHighErrorBurnRate
        expr: |
          slo:api_errors:ratio_rate1h > (14.4 * 0.001)
          and
          slo:api_errors:ratio_rate5m > (14.4 * 0.001)
        for: 2m
        labels:
          severity: critical
          slo: api-availability
        annotations:
          summary: "API error budget burning fast"
          description: "API error rate is {{ $value | humanizePercentage }} over the last hour. At this rate, the entire 30-day error budget will be consumed in 2 days."

      # Slow burn: 3x budget consumption over 3 days
      # Tickets - something is degraded but not urgent
      - alert: APISlowErrorBurnRate
        expr: |
          slo:api_errors:ratio_rate3d > (3 * 0.001)
          and
          slo:api_errors:ratio_rate6h > (3 * 0.001)
        for: 15m
        labels:
          severity: warning
          slo: api-availability
        annotations:
          summary: "API error budget burning slowly"
          description: "API error rate has been elevated for 3 days.
Error budget consumption is 3x the sustainable rate."
```

The `14.4` and `3` multipliers come from Google's SRE workbook. A 14.4x burn rate means you will exhaust your entire 30-day error budget in 2 days. A 3x burn rate means you will exhaust it in 10 days. The short window (5m or 6h) paired with the long window (1h or 3d) prevents false positives from brief spikes.

Sloth: SLO Generation from YAML

Writing recording rules and alerting rules by hand for every SLO is tedious and error-prone. We use Sloth to generate all of this from a simple YAML spec:

```
# slo-specs/api-availability.yaml
version: "prometheus/v1"
service: "api-server"
labels:
  team: platform
  environment: production
slos:
  - name: "api-availability"
    objective: 99.9
    description: "API server HTTP availability"
    sli:
      events:
        error_query: sum(rate(http_requests_total{job="api-
server",status=~"5.."}[{{.window}}]))
        total_query: sum(rate(http_requests_total{job="api-server"}
[{{.window}}]))
    alerting:
      name: APIAvailability
      labels:
        team: platform
      annotations:
        summary: "API availability SLO burn rate alert"
      page_alert:
        labels:
          severity: critical
          routing: pagerduty
      ticket_alert:
        labels:
          severity: warning
          routing: slack

  - name: "api-latency-p99"
    objective: 99.0
    description: "API server P99 latency under 500ms"
    sli:
      events:
        error_query: |
          sum(rate(http_request_duration_seconds_bucket{job="api-server",le="0.5"}[{{.window}}]))
          -
          sum(rate(http_request_duration_seconds_count{job="api-server"}[{{.window}}]))
        total_query: sum(rate(http_request_duration_seconds_count{job="api-server"}[{{.window}}]))
    alerting:
      name: APILatency
      labels:
        team: platform
      page_alert:
        labels:
          severity: critical
      ticket_alert:
        labels:
          severity: warning
```

Run `sloth generate -i slo-specs/ -o prometheus-rules/` and it produces complete `PrometheusRule` resources with all the recording rules, multi-window alerting, and error budget calculations. We commit the generated output to Git and deploy it via ArgoCD.

Sloth also generates Grafana dashboard JSON. The dashboards show SLO compliance, error budget remaining, and burn rate over time.

Grafana Dashboard for Error Budgets

Here is a snippet of the Grafana dashboard JSON we deploy for error budget visualization. This panel shows remaining error budget as a percentage over a 30-day rolling window:

```
{
  "panels": [
    {
      "title": "Error Budget Remaining (30d)",
      "type": "gauge",
      "datasource": "Prometheus",
      "targets": [
        {
          "expr": "1 - (slo:api_errors:ratio_rate30d / 0.001)",
          "legendFormat": "Budget Remaining"
        }
      ],
      "fieldConfig": {
        "defaults": {
          "unit": "percentunit",
          "min": 0,
          "max": 1,
          "thresholds": {
            "mode": "absolute",
            "steps": [
              { "value": 0, "color": "red" },
              { "value": 0.25, "color": "orange" },
              { "value": 0.5, "color": "yellow" },
              { "value": 0.75, "color": "green" }
            ]
          }
        }
      }
    },
    {
      "title": "Error Budget Consumption Over Time",
      "type": "timeseries",
      "datasource": "Prometheus",
      "targets": [
        {
          "expr": "slo:api_errors:ratio_rate5m / 0.001",
          "legendFormat": "Budget burn rate (5m)"
        },
        {
          "expr": "slo:api_errors:ratio_rate1h / 0.001",
          "legendFormat": "Budget burn rate (1h)"
        }
      ],
      "fieldConfig": {
        "defaults": {
          "unit": "percentunit",
          "custom": {
            "drawStyle": "line",
            "fillOpacity": 10
          }
        }
      }
    }
  ]
}
```

The gauge panel turns red when error budget is exhausted and green when healthy. This is the panel we put on the team's TV dashboard. When it turns orange, people notice. When it turns red, feature work stops.

Alert Routing: Page on SLO Burn, Not on Symptoms

One of the most impactful changes we make when onboarding teams to SLOs is consolidating their alerting. A typical team has 40-60 alerts: CPU high, memory high, disk full, latency spike, error rate up, queue depth growing, and so on. Most of these are symptom-based and fire frequently enough that the team ignores them.

We replace this with SLO-based alerting:

- **Page (PagerDuty, phone call):** Only for fast burn rate alerts. These mean the error budget is being consumed at 14x+ the sustainable rate. Something is actively broken and users are affected.
- **Ticket (Slack, Jira):** For slow burn rate alerts. These mean reliability is degraded but not critically. The team addresses them during business hours.
- **Dashboard only:** Infrastructure metrics (CPU, memory, disk) become dashboard-only. They are useful for diagnosis during an incident but should not trigger pages.

This typically reduces page volume by 70-80%. The alerts that do fire are meaningful and actionable.

SLO Review Meetings

We run monthly SLO review meetings with every team we work with. The agenda is fixed:

1. **Error budget status for each SLO.** How much budget was consumed this month? How much remains?
2. **Incidents that consumed budget.** What happened, what was the root cause, and what is the remediation status?
3. **Burn rate trends.** Is reliability improving or degrading month over month?
4. **Action items from last month.** Were they completed? Did they have the expected impact?
5. **SLO target review.** Every quarter, evaluate whether the target is too aggressive or too lenient based on actual data.

The meeting should include the engineering lead, product manager, and on-call engineer. Product needs to be in the room because error budget consumption directly impacts feature velocity. This is the meeting where "we need to slow down and fix reliability" gets backed by data instead of gut feelings.

Adjusting SLO Targets

Your first SLO targets will be wrong. Accept this. After 30 days of baseline data, review:

- If you never burned any error budget, your target is probably too lenient. Tighten it. A target you never risk violating is not driving behavior.
- If you exhausted your error budget in the first week, your target is too aggressive for your current system. Loosen it to something achievable, then improve your way to a tighter target.
- If the error budget drives the right conversations and decisions, the target is correct. Leave it alone.

The goal is not perfection. The goal is a target that creates the right tension between shipping features and maintaining reliability. SLOs are a tool for decision-making, not a vanity metric for uptime pages.

Getting Started

If your team has no SLOs today, here is the path we recommend:

1. Pick one critical user journey. Just one.
2. Instrument an availability SLI for it using your existing metrics.
3. Measure for 30 days without setting a target.
4. Set a target slightly above your baseline.
5. Configure multi-window, multi-burn-rate alerting.

6. Write an error budget policy and get it signed off by the engineering lead and product manager.
7. Run your first monthly SLO review.
8. Expand to additional journeys only after the first one is working.

Do not try to implement SLOs for everything at once. One well-implemented SLO that drives decisions is worth more than twenty that sit in a dashboard nobody opens.

If your organization is looking to implement an SLO program or improve an existing one, we have done this across teams of every size. Get in touch and we will share what has worked.

`sre` `slos` `observability` `prometheus` `grafana`

← Back to all articles