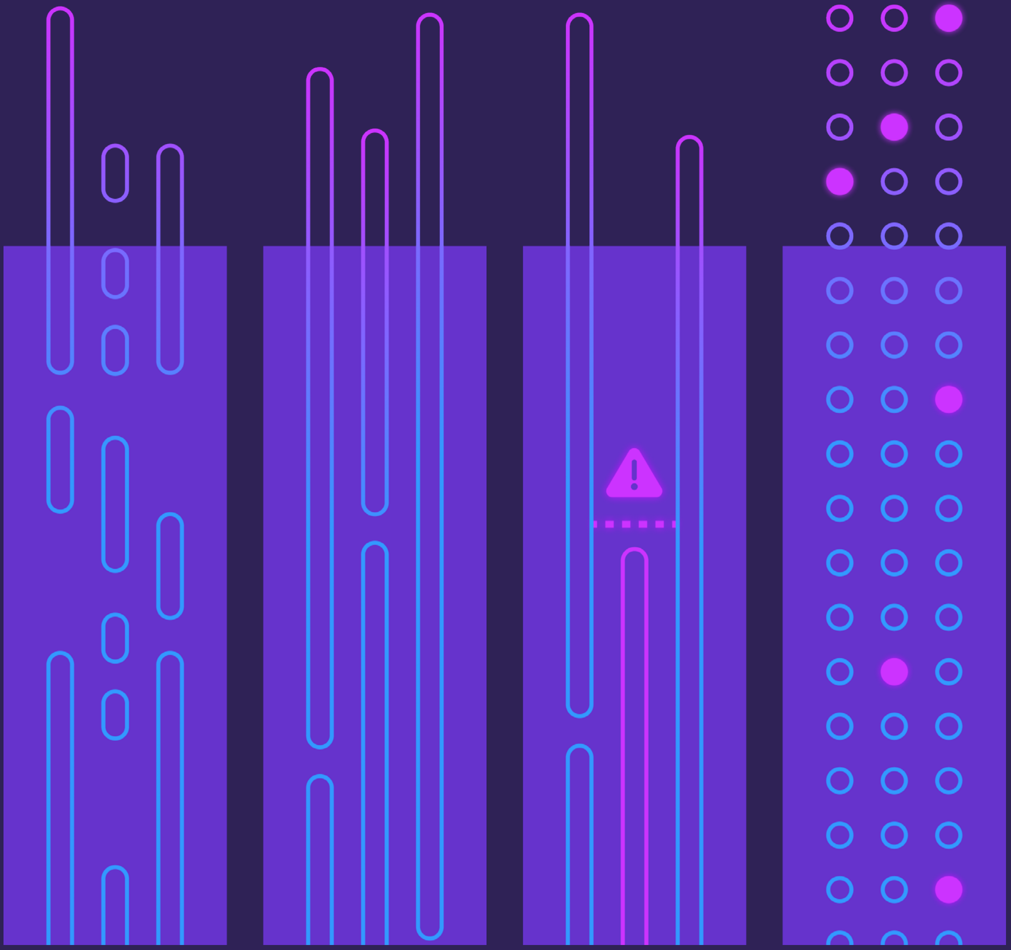
Gremlin

# Achieving SLO Success

## with Golden Signals & Reliability Testing



RELIABILITY MANAGEMENT

WHITEPAPER

## TABLE OF CONTENTS

# Executive Summary

Golden Signals represent the most important attributes of a system from the user's perspective and should be the foundation of any observability practice. This is especially true for organizations that provide Service Level Agreements (SLAs) to their customers, as SLA violations can have financial and legal consequences. In this white paper, we explain what the four Golden Signals are, how they fit into your organization's reliability goals, and how you can use them to become more successful at meeting your SLAs and providing the best possible customer experience.

# What are the four Golden Signals?

Having visibility into the health of your services is critical, but collecting too much data can be counterproductive. There are countless metrics to track, including CPU usage, network latency, processes, and uptime. Even for experienced SREs, identifying the right metrics to track isn't always straightforward, especially when working with large, complex systems.

Google provides guidance on key metrics through the four Golden Signals: ***latency, traffic, error rate,*** and ***resource saturation***. These represent the most important attributes of a system by telling us how well the system is performing, how close it is to its maximum capacity, what errors it's generating (if any), and how many more users it can serve before problems start to arise.

**Golden Signals originated from the Google Site Reliability Engineering Handbook. Many teams have adopted Golden Signals into their monitoring and alerting practices.**

## LATENCY

***Latency* is the difference in time between when one part of a system (such as a service) sends a request and when it receives a response. In a distributed system,** latency between services has a significant impact on performance. For example, imagine an online banking application consisting of a front end and a database. When a user submits a deposit, the front end has to perform multiple steps:

**1** – Retrieve the user's account info from the database.

**2** – Verify and process the deposit.

**3** – Store the transaction in the database.

**4** – Display a confirmation (or error) message to the user.

Each of these steps introduces latency, and the longer they take or, the more steps that are involved, the slower the experience becomes for the user.

The challenge with distributed system latency is that small amounts can lead to an exponentially slower experience. Even under tightly controlled conditions, adding just 100 milliseconds of latency can **reduce the performance of a web service by up to 140%.** Imagine if your users were accessing your service over a slow network connection, or if an unusually high amount of traffic in your data center is causing network congestion. This is why latency is important to users.

## TRAFFIC

***Traffic* is the amount of demand being placed on your systems and services.** Traffic is best measured in a way specific to your services and how your users interact with them. For example:

- Search engines might track queries per second.
- Web applications might track HTTP requests or page loads per second.
- Backend services or public-facing APIs might track API requests per second.
- Banks might track transactions per second.
- Databases and file storage services might track data throughput in gigabytes or terabytes per second.

## ERROR RATE

Not every request can be processed successfully. Maybe a user sent a request that was improperly formatted, an undiscovered bug in our backend code caused a crash, or the user's network connection dropped out mid-transaction. **A certain number of requests will fail, and when measured as a percentage of all requests, this is called the *error rate*.**

Tracking and troubleshooting errors ranges in difficulty depending on where the error occurred. Errors generated by our services are relatively easy to detect if we have observability since we can pull up logs, metrics, and traces. Errors generated by a user's system are more difficult since we don't have direct access to those systems. Errors can also masquerade as successes: for example, if a user can successfully access a website's login page but enters an incorrect password, should that count as an error?

To accurately reflect the user experience, engineers should differentiate between errors generated by infrastructure (e.g. network disconnections, failed requests, and hardware failures) and errors generated by services (e.g. bad user data, security controls, and invalid requests).

## RESOURCE SATURATION

***Saturation* measures how much of a given resource is being consumed. At an infrastructure level, this includes CPU, memory (RAM), disk space, disk bandwidth, and network utilization.** Reaching 100% utilization on any resource could cause performance drops, increased latency, reduced throughput, and a higher error rate.
It's also possible to measure saturation per service using a resource management system like **Kubernetes' pod resource limits.**

A key benefit of distributed systems is the ability to add or remove capacity on demand or in response to saturation. If our systems become too saturated due to heavy traffic, we can scale up by adding capacity via additional computing nodes, then re-deploy our services across those nodes so they can access those extra resources. Likewise, if saturation gets low, we can scale down by removing underutilized nodes and migrating our services back to a smaller set of nodes. This ensures our systems are always performant enough to meet user demand, while also saving costs during periods of little to no demand.

## What are Service Level Objectives (SLOS), AND WHY ARE THEY SO HARD TO IMPLEMENT?

When customers rely on a third-party service, they often want a guaranteed level of service. In enterprise applications, this comes in the form of a contract called a Service Level Agreement (SLA). An SLA describes the quality of service that a provider promises its customers. For cloud services, this typically includes service uptime, data integrity, and data retention.

An SLA is a high-level promise typically defined by business leaders. To tie the SLA back to technical systems and services, technical teams must identify the system and application-level metrics that best represent the goals of the SLA. These are called Service Level Indicators (SLIs). Teams must also identify the range of values that those metrics must fall within, which are called Service Level Objectives (SLOs). Engineers use SLIs to monitor the health and performance of a system or service, and if those SLIs fall outside of the range specified in the SLO, then the business is at risk of breaching its SLA.

SLAs are first and foremost business tools. They're legal contracts between business leaders and customers, but they're ultimately based on technology. The most difficult part of implementing an SLA isn't necessarily defining the quality of service or the amount of compensation, but translating business objectives into technical requirements. Teams need to convert business language into technical SLOs and SLIs, and they need to do this in a way that accurately represents and measures business goals. Doing this takes ongoing collaboration across the organization, a mature observability practice, and a strong understanding of how the behavior of technology systems influences the user experience.

Even when there's alignment on what the SLA should promise, there's still the challenge of implementation. SLIs require observability, and if teams aren't already monitoring their services, setting up observability adds development time and effort. SLAs raise many other questions, including:

**Engineers use SLIs to monitor the health and performance of a system or service, and if those SLIs fall outside of the range specified in the SLO, then the business is at risk of breaching its SLA.**

- Which system(s) or service(s) are relevant to my SLA? Which ones do I need to monitor, and which ones are ancillary?
- How do we determine when an SLO has been violated? Who do we need to notify?
- How frequently should we sample our SLIs and compare them to our SLOs? How much will it cost to collect, analyze, and retain this data, and how long should we keep it?
- How do we set up proactive monitoring and alerting to know when an SLO might be violated soon? Do we need to set up an on-call rotation, so someone is always available in case we exceed an SLO?
- Do we need to create incident response runbooks or disaster recovery plans for when an SLO is violated? What should those runbooks and plans look like, and where will we store them so on-call engineers can quickly access them? Will we need to train engineers on these plans?
- How will we keep track of your SLA(s) once it becomes part of our business processes? How can we encourage engineers, project managers, etc. to track their SLA adherence continually?
- What time periods should your SLA apply to? Should it be 24/7, or only cover normal business hours?
- How much compensation should you provide? Do you offer a fixed financial discount (e.g. 100% refund or credit for any downtime above a certain amount) or a sliding scale like the **Amazon Compute SLA**? Should you build this compensation method into your billing process, and if so, who would need to be involved, and how long would it take?
- Should you offer customer support when your SLA is violated, and if so, what would that look like?

# How to integrate Golden Signals into your SLOs

Recall that SLAs are contractual agreements promising a certain level of performance to customers, and Golden Signals measure how well a system or service is performing. At a high level, integrating Golden Signals into your SLOs involves three steps:

1 – **Identify your most critical services.**

2 – **Create monitors and/or alerts in your observability tool.**

3 – **Run tests to ensure your Golden Signals are tracking to your SLOs.**

## IDENTIFY YOUR MOST CRITICAL SERVICES

A critical service is any service that will bring your application down if it fails. If you're unsure which of your services are critical, consider your application's **critical path**. The critical path is the set of components (in this case, services) required for your application to serve its core function.

For example, imagine we're responsible for the reliability of a microservice-based banking application. Each function of our bank is a distinct service.

Each function of the website is a distinct service, such as the website frontend, user account management, and transaction ledger services. As a bank, our core workflow is allowing customers to log into their accounts, check their balances, and transfer money. This means ensuring that our website frontend, user account management, and transaction ledger services are part of our critical path. Therefore, we should focus our initial reliability efforts on them.

## CONNECT YOUR GOLDEN SIGNALS TO YOUR SLIS

Once we've chosen one or more services, we need to identify their SLIs. Remember—an SLI is a metric that you measure to ensure adherence to your SLOs. Think about the Golden Signals (latency, traffic, error rate, and resource saturation), what they represent, and how that relates to your SLIs. Sometimes this is a one-to-one relationship: for example, if one of your SLIs tracks the response time of each HTTP request that your frontend receives, then you're already tracking latency. If this isn't the case, create a new SLI by using your monitoring tool to collect and aggregate response times from your service. Then, based on the parameters of your SLO, create a monitor to continually check this SLI and set the appropriate warning and alert levels.

Keep in mind that Golden Signals aren't necessarily independent of each other. An increase in latency might've been preceded by an increase in resource saturation or traffic. Likewise, an increased error rate could be caused by increased saturation of a critical dependency causing it to fail. Correlations like these won't always be obvious, but tracking each Golden Signal will make it easier to recognize them.

# How to ensure you continually meet your SLOs

Now that we've set our SLOs and identified our SLIs, how do we make sure we're meeting them? The obvious answer is "keep our systems reliable," but there are a lot of pieces to this simple statement. Reliability can mean different things to different teams, even teams within the same organization. That's why it's important to:

1 – **Create a standard definition of reliability and have a way to track it centrally for all teams.**

2 – **Use a consistent method of tracking your adherence to your SLOs.**

3 – **Run regular, proactive reliability tests to ensure you can always meet your SLOs.**

## CENTRALIZE AND STANDARDIZE RELIABILITY

Although teams deploy and manage services independently of each other, we should still have a centralized place to observe and track the reliability of every service. For example, allowing each team to use its own observability tool will likely result in several tools with data silos and different dashboards that teams need to spend time exporting and sharing. This makes it difficult to correlate SLIs and SLOs, which is especially dangerous since SLOs are primarily driven by the organization and not technology teams.

Centralization has another benefit: standardization. For SLAs to work effectively, engineering teams need to agree on using the same standards of measuring and tracking reliability. Otherwise, what one team considers degraded service might be considered a critical failure by another team. Standardization ensures that different teams are aligned on the same best practices for measuring and testing their services to a consistent organizational standard.

## USE MONITORS TO TRACK YOUR ADHERENCE TO YOUR SLOS

In your observability tool, make sure to create monitors using your SLIs. Set an alert threshold based on your SLOs, and if your tool supports it, set a warning threshold to give your team advanced notice before your SLO is breached. Many enterprise observability tools like Datadog and New Relic can trigger monitors based on gradual trends in SLIs and sudden rapid changes. Even a few seconds of advance warning can give your team the time needed to avoid a large-scale outage.

## USING RELIABILITY MANAGEMENT TO ENSURE YOU'RE CONSISTENTLY MEETING YOUR RESILIENCY TARGETS

Setting up monitors and alerts gives you fast feedback on changes to your systems, but what if you want to be more proactive? What if you'd like to use your Golden Signals to detect potential problems with your systems before they happen? That's where Reliability Management becomes useful.

**Reliability Management is a practice that helps teams automate and standardize reliability. This includes testing the reliability of systems and services, measuring reliability in an objective way, and standardizing those tests and measurements across the organization.** As the leading Reliability Management platform, Gremlin lets you do this in a safe, simple, and secure way.

How do Golden Signals and SLOs fit into Reliability Management? When running reliability tests, Gremlin monitors your Golden Signals to determine whether a service is healthy. If the Golden Signals remain in a healthy state (e.g. within your SLOs) throughout the test, then your service is reliable. If not, then you now have a clear direction on how to address those shortcomings.

### The way this works is simple:

1 – **Define your services in Gremlin.**

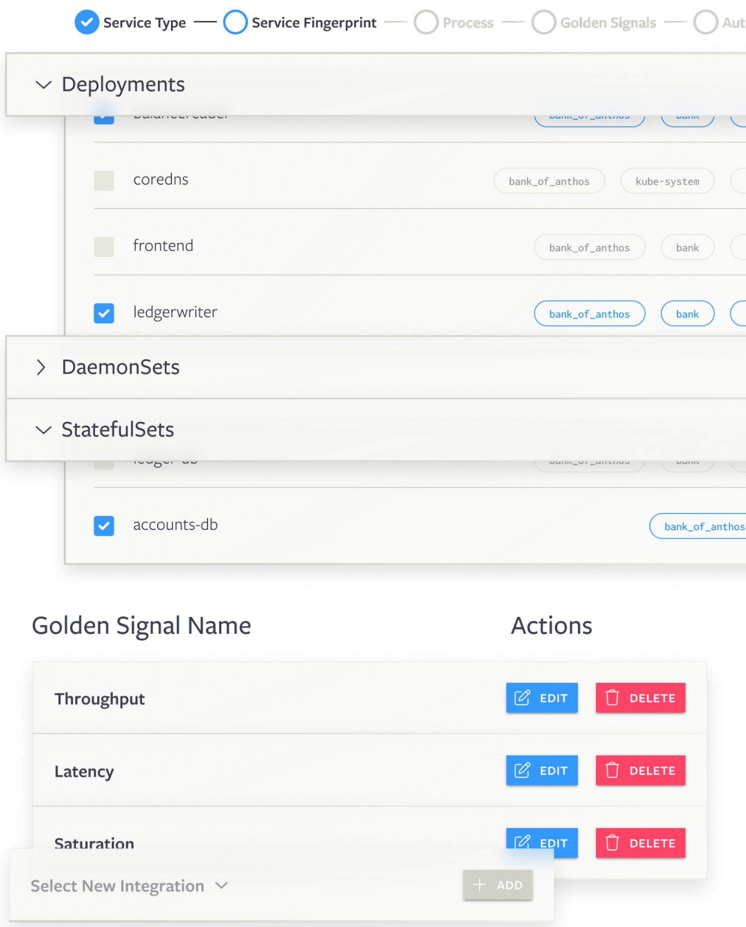
2 – **Run reliability tests to identify failure modes.**

3 – **Run regular automated reliability tests to maintain your reliability posture.**

## 1. Set up your services in Gremlin

Gremlin uses services as the unit of reliability measurement and improvement. A service is a specific set of functionality with clear interfaces provided by one or more systems within an environment, like a checkout or authentication service. Services map to infrastructure running in your environment, including hosts, containers, and Kubernetes resources. Gremlin automatically detects your infrastructure, so adding a service is as simple as selecting the relevant components from a list.

While creating your service, you'll be able to link your Golden Signals to that service by integrating Gremlin with your observability tool. All you need to do is enter the URLs of the monitors corresponding to your Golden Signal, provide credentials to your observability tool, and click Add. Once Gremlin successfully verifies the connection, it will automatically check those monitors before, during, and after each reliability test. If any of your monitors becomes unhealthy, Gremlin immediately halts the test and rolls back its impact to prevent unintended outages.

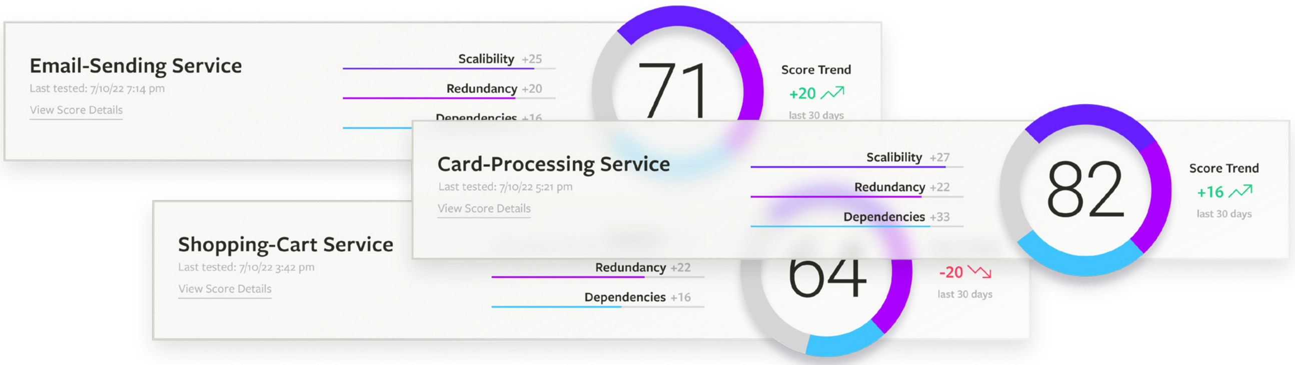


## 2. Proactively run reliability tests

Once you've added a service, Gremlin provides a suite of reliability tests that you can run on it right away. These tests identify potential failure modes that teams may run into, such as poor (or zero) redundancy, limited scalability, and tightly coupled dependencies. By running each test and tracking any failures, you'll gain a clear picture of the possible vulnerabilities in your service and how addressing them will improve your overall reliability posture.

There's more you can do with reliability tests than checking the health of a service. Reliability tests simulate real-world scenarios such as region or zone outages. Some of these scenarios require intervention from engineers (i.e. incident response). Running reliability tests lets you simulate real-world incidents so you can test your incident response procedures, ensure your playbooks are up-to-date, and train your engineers on how to handle real-world failures.

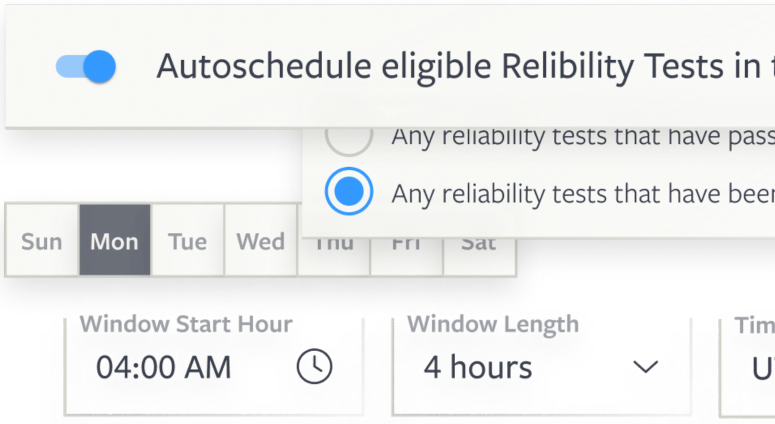
Lastly, you can use these tests to verify that your monitoring and observability are set up correctly in the first place. For example, if you run a reliability test that you know will cause one of your Golden Signals to enter a failure state, yet that Golden Signal never reports as unhealthy, you know your monitor isn't configured correctly. This allows you to fix your monitors so that they're reporting accurately.



## 3. Maintain your reliability posture with regular automated testing

After you've worked to improve your reliability posture, it's important to ensure you're maintaining it. Just because you ran your reliability tests doesn't mean your service will always remain reliable; code deployments, infrastructure changes, and other variables can affect the reliability of your services. Regular testing lets you validate that these changes don't negatively impact your reliability, and if they do, then you have advanced warnings to address them.

Gremlin makes it easy to auto-schedule reliability tests for each of your services. You can set your tests to run on specific days of the week and during a time window of 2–24 hours. You can also specify whether to run tests that have been run at least once or only to run tests that have passed at least once.


Autoschedule eligible Relibility Tests in t
Any reliability tests that have pass
Any reliability tests that have been
Sun
Mon
Tue
Wed
Thu
Fri
Sat
Window Start Hour
04:00 AM
Window Length
4 hours
Tim
U

## Conclusion

Understanding reliability is already difficult, and will only become more difficult as systems become more complex. The four Golden Signals are an easy and effective way to measure the most important aspects of a system, and when paired with a reliability management platform like Gremlin, they help you proactively meet your SLOs so you can meet your legal obligations and deliver the perfect customer experience.

Gremlin

Gremlin is the enterprise Chaos Engineering platform on a mission to help build a more reliable internet. Their solutions turn failure into resilience by offering engineers a fully hosted SaaS platform to safely experiment on complex systems, in order to identify weaknesses before they impact customers and cause revenue loss.