GSDC
Global Skill Development Council

SITE RELIABILITY ENGINEERING (SRE)

Quick Reference Guide

www.gsdcouncil.org

1. Introduction to Site Reliability Engineering

What is Site Reliability Engineering?

Site Reliability Engineering (SRE) is a discipline that applies software engineering principles to IT operations and infrastructure management. The goal is to create highly scalable, reliable, and efficient systems that can withstand the demands of modern production environments. Originally developed at Google, SRE has since become a widely adopted practice across the industry.

Rather than treating operations as a set of manual tasks performed by a separate team, SRE embeds engineers with software development skills directly into operations workflows. This approach ensures that reliability is treated as a feature — not an afterthought — and that operational concerns are addressed with the same rigor applied to product development.

Reliability

Ensuring systems remain operational and resilient under expected and unexpected conditions.

Scalability

Designing systems to handle growing workloads without degradation in performance.

Automation

Replacing repetitive manual tasks with automated workflows to reduce toil and human error.

Monitoring

Continuous observation of infrastructure and application health to detect and respond to issues early.

Incident Management

Structured processes for detecting, triaging, and resolving service disruptions quickly and effectively.

Performance Optimization

Continuously measuring and improving system responsiveness, throughput, and efficiency.

2. Core Goals of SRE

Every SRE practice is oriented around a set of core goals that together define what it means for a system to be production-ready. These goals provide the organizing framework for day-to-day SRE work — from writing runbooks to evaluating deployment risk. Understanding them is essential for aligning SRE efforts with broader engineering and business objectives.

Reliability

Ensure systems remain operational and meet user expectations consistently across all conditions.

Scalability

Support growing workloads by designing systems that expand capacity gracefully without failure.

Automation

Reduce manual operational tasks through systematic use of tooling, scripts, and self-healing systems.

Availability

Minimize downtime by building in redundancy, failover, and proactive incident detection.

Performance

Optimize system responsiveness and efficiency to meet latency and throughput targets.

Efficiency

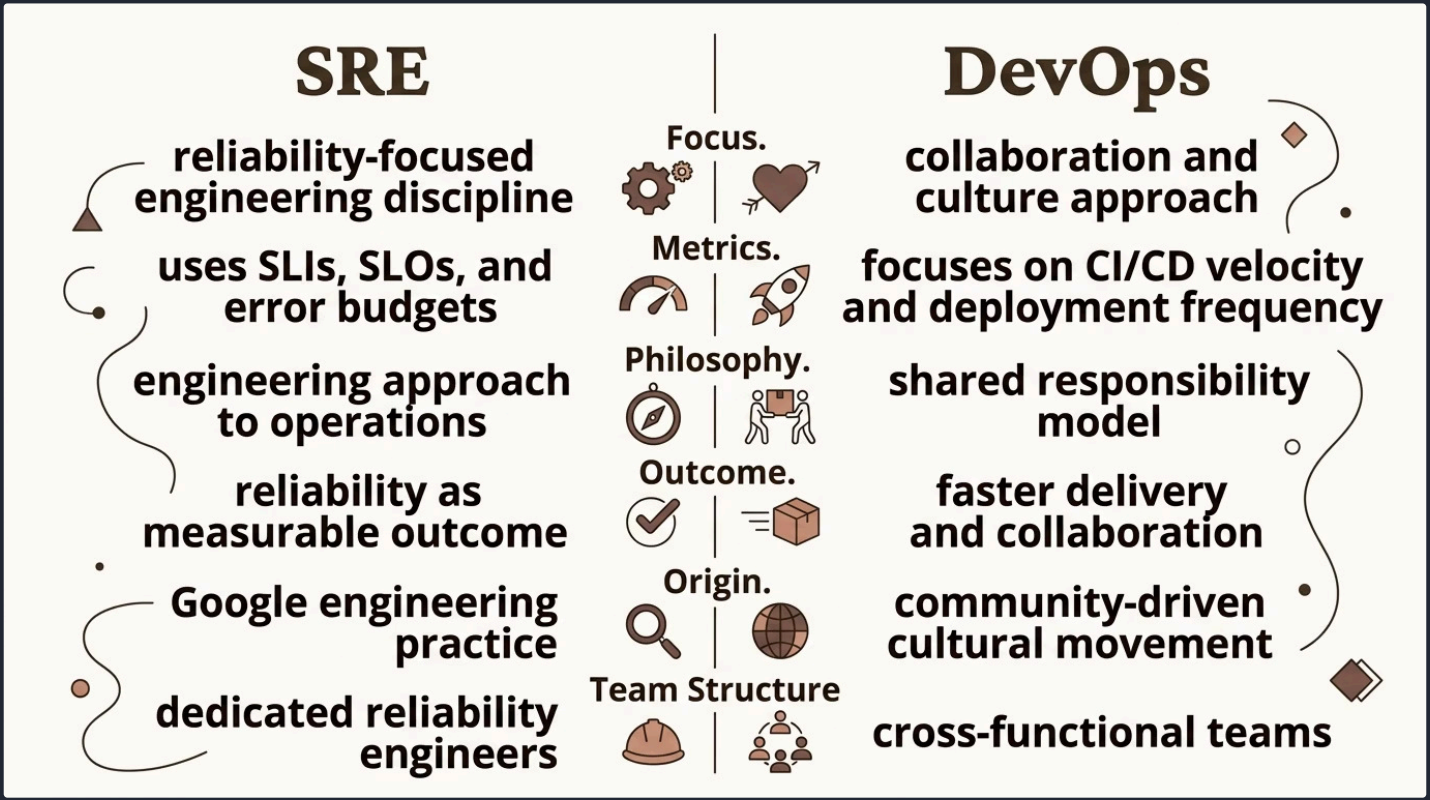
Improve operational productivity by eliminating waste, redundant effort, and unnecessary complexity.

These goals are not isolated — they reinforce one another. For example, improved automation directly enables greater reliability and efficiency, while strong monitoring underpins both performance optimization and availability. A mature SRE practice develops each of these dimensions in parallel.

3. SRE vs DevOps

SRE and DevOps are complementary but distinct approaches. DevOps is a cultural philosophy emphasizing collaboration between development and operations teams, continuous integration and delivery, and shared ownership of the software lifecycle. SRE, by contrast, is an opinionated engineering discipline with specific practices, metrics, and tooling for achieving measurable reliability outcomes.

A useful way to think about it: **DevOps defines the "what" and "why" of collaboration**, while **SRE provides a prescriptive "how"** — including error budgets, SLOs, and blameless postmortems. Many organizations implement both simultaneously, using DevOps culture as the foundation and SRE practices as the operational layer on top.



In practice, the two approaches are often used together. DevOps creates the cultural environment in which SRE practices can thrive, and SRE provides the engineering rigor that gives DevOps its measurable operational outcomes. Understanding where they differ — and where they align — helps teams make informed decisions about how to structure their reliability work.

4. Key SRE Principles

The core principles of SRE form the intellectual backbone of the discipline. They guide how SRE teams make decisions, prioritize work, and build culture. Each principle addresses a specific operational challenge and provides a lens through which to evaluate SRE maturity and effectiveness within an organization.

Automation

Eliminate repetitive work by building systems that handle routine tasks programmatically. Automation frees engineers to focus on high-value work and reduces the risk of human error in production.

Monitoring

Improve system visibility through comprehensive, proactive monitoring. What isn't measured can't be managed — monitoring data is the foundation for all operational decision-making.

Toil Reduction

Reduce manual operational effort that scales with traffic but adds no long-term value. SRE teams should spend no more than 50% of their time on toil, with the rest invested in engineering work.

Observability

Build systems that expose their internal state through structured telemetry. Observability goes beyond monitoring — it enables teams to ask novel questions about system behavior without deploying new instrumentation.

Error Budgets

Use quantified reliability targets to balance innovation with stability. Error budgets give teams a shared language for risk decisions and prevent both over-engineering and reckless deployment.

Blameless Culture

Encourage learning from failures without assigning individual blame. A blameless culture surfaces systemic problems and enables honest postmortems that actually improve reliability over time.

5. Important SRE Terminology

Mastering SRE requires familiarity with a specific vocabulary. These terms appear throughout certification materials, job descriptions, incident reports, and production reviews. Understanding their precise definitions — and how they relate to one another — is foundational to practicing SRE effectively.

Term	Meaning
SLA	Service Level Agreement — a formal, contractual commitment between a service provider and a customer defining expected performance, availability, and consequences for failure.
SLI	Service Level Indicator — a quantitative metric used to measure the performance or reliability of a service, such as availability percentage or request latency.
SLO	Service Level Objective — an internal target for a given SLI, representing the reliability goal the team is engineering toward. SLOs inform error budget calculations.
MTTR	Mean Time To Recovery — the average time required to restore a service to normal operation following an incident. A key measure of incident response effectiveness.
MTTD	Mean Time To Detect — the average time elapsed between the start of an incident and its detection. Shorter MTTD reduces customer impact.
Incident	An unplanned service disruption or degradation that affects users or system functionality. Incidents are classified by severity and managed through a defined lifecycle.
Toil	Manual, repetitive, automatable work that scales with service growth but does not provide long-term engineering value. Reducing toil is a primary SRE objective.

ⓘ These terms form the foundation of SRE communication. Using them precisely — especially in postmortems, SLO reviews, and stakeholder updates — signals operational maturity and ensures shared understanding across teams.

6. Service Level Indicators (SLIs)

Service Level Indicators are the quantitative metrics at the heart of SRE reliability measurement. An SLI is a carefully selected metric that reflects the user experience of a service — not just whether the system is "up," but whether it is performing in a way that users find acceptable. Choosing the right SLIs is one of the most important — and most nuanced — tasks an SRE team performs.

Common SLI Types

SLI	Measures
Availability	Uptime percentage
Latency	Response time (p50, p95, p99)
Throughput	Request handling rate
Error Rate	Percentage of failed requests
Durability	Data retention reliability

Good SLIs directly reflect user experience. Avoid internal metrics like CPU usage as SLIs — these are symptoms, not outcomes.

Example SLI Formula

API Availability

```
Availability =
Successful Requests
÷ Total Requests × 100%
```

This formula is the foundation for most availability SLIs. A request is "successful" when it returns a valid response within an acceptable latency threshold — not simply when it returns any HTTP response.

When defining SLIs, teams should consider the user journey end-to-end. A single service may require multiple SLIs to fully describe its reliability from the user's perspective. For example, an API might have separate SLIs for availability, latency at the 99th percentile, and error rate — each capturing a different dimension of quality.

7. Service Level Objectives (SLOs)

Service Level Objectives are the reliability targets that SRE teams commit to maintaining. An SLO is expressed as a specific value for a given SLI over a defined time window — for example, "API availability will be 99.9% over any rolling 30-day period." SLOs are internal commitments, not customer-facing contracts, which gives teams the flexibility to set them based on what is actually achievable and meaningful.

Website Availability

SLO: 99.9%

Less than 8.7 hours of downtime per year. Appropriate for most customer-facing web applications.

API Response Time

SLO: < 200ms

99th percentile latency target. Ensures fast, consistent performance for API consumers under load.

Error Rate

SLO: < 1%

No more than 1 in 100 requests may fail. Drives engineering focus on error handling and resilience.

SLO Benefits

Defines Reliability Targets

SLOs translate abstract reliability goals into concrete, measurable engineering targets. They give teams a clear definition of "good enough" — preventing both under-engineering and over-engineering.

Aligns Business Expectations

SLOs create a shared language between engineering and business stakeholders. They enable product managers, executives, and SREs to have informed conversations about reliability trade-offs.

Supports Operational Decisions

When an SLO is close to being violated, teams have an objective basis for slowing deployments, prioritizing reliability work, or escalating incidents — without relying on gut instinct.

8. Service Level Agreements (SLAs)

A Service Level Agreement is a formal, contractual commitment made to customers that defines the minimum level of service they can expect. Unlike SLOs — which are internal targets — SLAs are external promises with legal and financial consequences if they are not met. SLAs are typically more conservative than SLOs, providing a buffer that accounts for unexpected failures.

A well-structured SLA goes beyond a simple uptime number. It specifies how availability is measured, what constitutes an incident, how support requests are handled, and what remedies — such as service credits — are available to customers when commitments are missed. SREs play a key role in determining what SLA commitments are technically feasible and in monitoring compliance continuously.

Component	Description
Availability Target	The expected uptime percentage (e.g., 99.9%) over a defined measurement period, typically monthly or annually.
Support Response Time	Maximum time within which the provider commits to acknowledging and responding to incident reports based on severity level.
Penalties	Contractual consequences — such as service credits or refunds — that apply when the provider fails to meet the stated availability or support targets.

⚠ SLAs should always be set less aggressively than internal SLOs. If your SLO is 99.9%, your SLA might commit to 99.5% — preserving an operational buffer and reducing the risk of costly penalties during unexpected incidents.

9. Error Budgets

The error budget is one of SRE's most powerful and distinctive concepts. It represents the permissible amount of unreliability within a given SLO — the portion of time or requests during which a service is allowed to fail before the SLO is violated. Error budgets transform reliability from an abstract virtue into a concrete, consumable resource that teams can reason about and manage deliberately.

Error Budget Formula

Error Budget = 100% − SLO

Example:
SLO = 99.9%
Budget = 0.1%

Over 30 days:
= 43.2 minutes of
allowed downtime

Purpose of Error Budgets

Balance innovation and reliability. When budget is plentiful, teams can move fast and deploy frequently. When budget is depleted, the team must pause new feature work and prioritize reliability improvements.

Control deployment risks. Error budgets create an objective, data-driven mechanism for deciding whether it is safe to proceed with a risky deployment or whether caution is warranted.

Support release decisions. Product and engineering teams can align on release timelines using budget consumption as a shared, neutral input — removing subjective conflict from reliability conversations.

ⓘ Error budgets reset at the start of each measurement window. A team that has burned through its entire budget in the first week of a month faces a very different set of constraints than a team that has used only 20% of its budget with two weeks remaining.

10. Toil Management

Toil is manual, repetitive, automatable work that grows in proportion to service traffic and operational scale. It is tactical, interruptive, and devoid of long-term engineering value. The SRE discipline prescribes that no more than 50% of an SRE team's time should be spent on toil — the remainder should go to engineering work that permanently reduces operational burden.

Characteristics of Toil

→ Repetitive

The same task performed again and again with no variation or learning value.

→ Manual

Requires direct human intervention — cannot be completed without an engineer's input.

→ Scales Poorly

Work volume increases with traffic, users, or infrastructure scale — not with team size or engineering value.

→ Lacks Long-Term Value

Completing the task returns the system to a previous state rather than improving it.

Toil Reduction Strategies

Automation

Script and automate recurring operational workflows using purpose-built tooling.

Self-Healing Systems

Build systems that detect and recover from common failure modes without human intervention.

Infrastructure as Code

Replace manual provisioning with declarative, version-controlled infrastructure definitions.

Monitoring Improvements

Reduce alert noise and false positives to minimize unnecessary on-call interruptions.

11. Monitoring Fundamentals

Monitoring is the continuous observation of systems and infrastructure to detect degradation, failure, and anomalies before they impact users. A mature monitoring strategy covers the full stack — from physical infrastructure and network layers to application performance and user experience. Without comprehensive monitoring, SRE teams are flying blind, relying on user reports rather than proactive detection to discover problems.

Infrastructure Monitoring

Tracks the health and performance of servers, networks, and storage systems. Detects hardware failures, resource exhaustion, and network connectivity issues before they cascade.

Application Monitoring

Measures application-level performance including response times, error rates, and request throughput. Surfaces code-level issues that infrastructure monitoring cannot detect.

Log Monitoring

Analyzes structured and unstructured log data for error patterns, security events, and operational anomalies. Essential for root cause analysis during and after incidents.

Network Monitoring

Observes connectivity, bandwidth utilization, and traffic patterns across the network layer. Identifies latency spikes, packet loss, and unusual traffic behavior.

Key Monitoring Metrics

Metric	Description
CPU Usage	Processor utilization across hosts and services — high values may indicate runaway processes or under-provisioned capacity.
Memory Usage	RAM consumption — memory leaks and unexpected spikes are common early indicators of application-level problems.
Disk I/O	Storage read/write performance — I/O saturation is a frequent cause of application latency and database degradation.
Network Latency	Communication delays between services — elevated latency often precedes broader availability failures.

12. Observability

Observability is the ability to infer the internal state of a system from its external outputs. While monitoring tells you when something is wrong, observability helps you understand *why* it is wrong — even in scenarios that were never anticipated when the system was built. This distinction is critical in complex, distributed architectures where failures are often emergent and non-obvious.

Observability is built on three foundational data types, each of which provides a different window into system behavior. Together, they enable engineers to reconstruct exactly what happened during an incident and identify the root cause with confidence rather than guesswork.

Logs

Time-stamped records of discrete events within a system. Logs provide the most granular view of system behavior and are essential for reconstructing the sequence of events during an incident.

Metrics

Quantitative measurements aggregated over time. Metrics are efficient to store and query, making them ideal for dashboards, alerting, and trend analysis. They answer "how much" and "how often."

Traces

Representations of the path a request takes through a distributed system. Traces connect related events across multiple services, enabling engineers to pinpoint latency and failure at the service-call level.

3x

Faster Troubleshooting

Observability-mature teams resolve incidents significantly faster by reducing the time spent on hypothesis testing.

↓ 70%

Reduced MTTR

Structured telemetry dramatically reduces mean time to recovery by surfacing root causes directly.

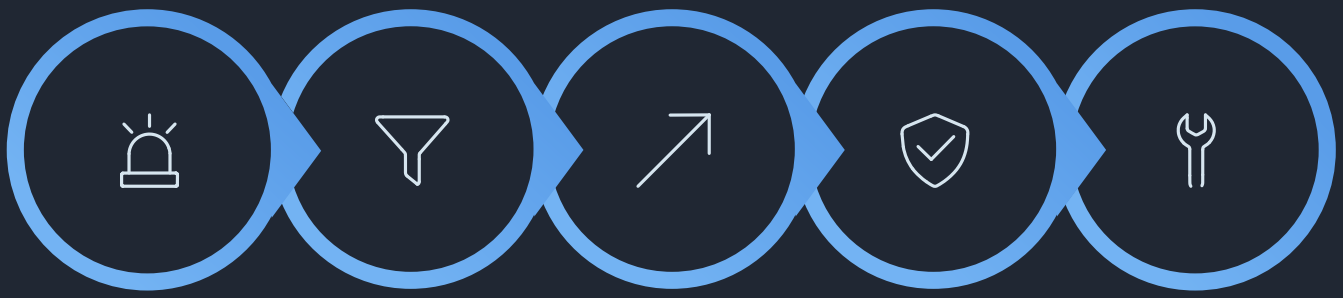
360°

Full System Visibility

Logs, metrics, and traces together provide complete coverage of system behavior across all layers.

13. Incident Management

Incident management is the structured process by which SRE teams detect, respond to, and resolve service disruptions. A well-defined incident management process minimizes customer impact, ensures consistent communication, and creates the operational data needed for continuous improvement. Every second of unstructured response during an incident is a second of avoidable damage to users and the business.



Each phase of the incident lifecycle has distinct goals. Detection focuses on identifying that something is wrong as quickly as possible. Triage determines scope and severity. Escalation ensures the right people are engaged. Mitigation stops the bleeding. Resolution addresses the underlying cause. And the postmortem ensures the team learns from the experience.

Incident Severity Levels

Level	Severity	Description
SEV-1	Critical	Complete service outage affecting all users. Requires immediate all-hands response and executive notification.
SEV-2	Major	Significant degradation or partial outage with major user impact. Requires urgent response and active incident commander.
SEV-3	Minor	Limited impact affecting a subset of users or functionality. Managed during business hours with standard response procedures.
SEV-4	Informational	No current user impact but a condition requiring monitoring or investigation to prevent escalation.

14. Blameless Postmortems

A blameless postmortem is a structured retrospective conducted after an incident with the explicit goal of understanding what happened — not who is at fault. The blameless philosophy recognizes that well-intentioned engineers operating within complex systems will make mistakes, and that punishing individuals for those mistakes discourages honesty, suppresses information, and prevents systemic improvement.

The goal of a blameless postmortem is to identify the systemic conditions that allowed the incident to occur and to produce actionable improvements that make recurrence less likely. When done well, postmortems are among the most valuable learning activities available to an engineering organization.

Postmortem Goals

→ Identify Root Causes

Surface the underlying systemic issues, not just the proximate trigger.

→ Improve Systems

Produce concrete changes to code, process, tooling, or monitoring.

→ Prevent Recurrence

Address contributing factors before they cause the next incident.

→ Encourage Learning Culture

Create psychological safety for honest reflection and information sharing.

Postmortem Components

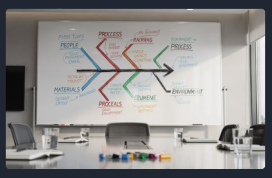
Component	Description
Timeline	Chronological sequence of events from first symptom to resolution.
Root Cause	The underlying systemic issue that enabled the incident to occur.
Impact Analysis	Quantified business and user impact including duration and scope.
Corrective Actions	Specific, assigned improvements with owners and due dates.

15. Root Cause Analysis (RCA)

Root Cause Analysis is the systematic process of identifying the underlying cause of an incident — not just the most visible symptom or the most recent contributing factor. Effective RCA distinguishes between proximate causes (the immediate trigger) and root causes (the systemic conditions that made the incident possible). Without this distinction, corrective actions address symptoms rather than causes, and incidents recur.

5 Whys

A deceptively simple technique: ask "why?" five times in succession, with each answer forming the basis of the next question. Each iteration moves deeper into the causal chain, past symptoms and toward systemic root causes. Best used for straightforward, linear failure chains in well-understood systems.



Fishbone Diagram

Also known as an Ishikawa or cause-and-effect diagram, the fishbone technique organizes potential causes into structured categories — such as people, process, technology, and environment. It is particularly useful when an incident may have multiple contributing factors that span organizational boundaries.

Fault Tree Analysis

A top-down, deductive analytical method that maps the logical relationships between a system failure and its contributing causes using Boolean logic gates. Fault tree analysis is well-suited for complex systems where multiple failure modes can combine in non-obvious ways to produce a system-level failure.

16. Automation in SRE

Automation is the cornerstone of SRE practice. Where traditional operations teams scale headcount linearly with infrastructure complexity, SRE teams use automation to decouple operational burden from system scale. The goal is not merely to save time — it is to build systems that are inherently more consistent, reliable, and resilient than those dependent on manual human intervention.

Automation Objectives

01

Reduce Manual Tasks

Replace repetitive human interventions with automated workflows that execute consistently at any scale.

02

Improve Consistency

Eliminate configuration drift and human variation by enforcing uniform, codified procedures.

03

Minimize Human Error

Remove the risk of typos, missed steps, and fatigue-induced mistakes from critical operational workflows.

04

Increase Scalability

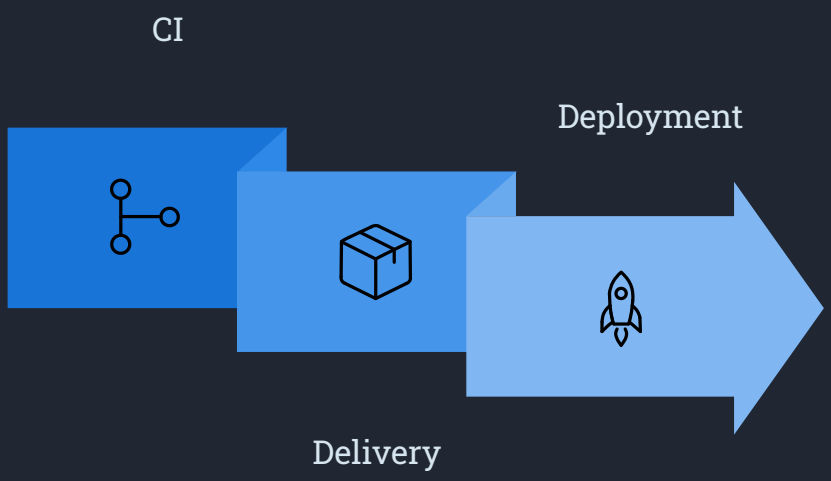
Enable systems to grow without a proportional increase in operational headcount or toil.

Common Automation Areas

Area	Example
Deployment	CI/CD pipelines that automatically build, test, and deploy code changes to production with rollback capability.
Infrastructure	Infrastructure as Code tools that provision and configure environments declaratively and repeatably.
Incident Response	Automated alerting, on-call routing, and runbook execution triggered by monitoring signals.
Scaling	Auto-scaling systems that add or remove compute resources dynamically based on real-time demand metrics.

17. CI/CD Fundamentals

Continuous Integration, Continuous Delivery, and Continuous Deployment form a pipeline that enables software teams to release changes rapidly, safely, and repeatedly. Together, they reduce the risk associated with large, infrequent releases by making deployments small, incremental, and reversible. For SRE teams, CI/CD is both a reliability enabler and an error budget consumer — the pipeline itself must be reliable and well-instrumented.



CI/CD Definitions

Term	Meaning
Continuous Integration	Developers merge code changes frequently — multiple times per day — with automated build and test validation on every merge.
Continuous Delivery	Every validated code change is automatically prepared for release to production. Human approval gates the final deployment step.
Continuous Deployment	Every validated change is automatically deployed to production without human intervention, requiring robust automated testing and monitoring.

CI/CD Benefits for SRE

Faster Releases

Smaller, more frequent deployments reduce the time from code completion to user value.

Reduced Deployment Risk

Small changesets are easier to test, review, and roll back than large batch releases.

Improved Quality

Automated test gates catch regressions before they reach production, protecting SLOs.

18. Infrastructure as Code (IaC)

Infrastructure as Code is the practice of managing and provisioning computing infrastructure through machine-readable configuration files rather than through manual processes or interactive configuration tools. IaC applies the same principles to infrastructure that software engineering applies to application code — version control, peer review, automated testing, and repeatable deployment.

For SRE teams, IaC is a foundational toil-reduction strategy. Manual infrastructure provisioning is slow, error-prone, and impossible to audit. IaC replaces it with a declarative, auditable, reproducible workflow that enables teams to rebuild entire environments from scratch in minutes rather than days.

Consistency

Every environment — dev, staging, production — is built from the same code, eliminating the "works on my machine" problem and configuration drift between environments.

Repeatability

Infrastructure can be torn down and rebuilt identically at any time. Disaster recovery scenarios that once took days can be executed in minutes with IaC.

Scalability

Infrastructure changes that once required manual effort on dozens of servers can be applied uniformly across thousands of resources with a single code change.

Faster Provisioning

New environments, services, and resources can be stood up in minutes rather than hours, accelerating development velocity and incident response.

Common IaC Tools

Tool	Purpose
Terraform	Cloud-agnostic infrastructure provisioning using declarative HCL configuration. Manages resources across AWS, Azure, GCP, and hundreds of other providers.
Ansible	Agentless configuration management and application deployment using YAML playbooks. Ideal for post-provisioning configuration and software installation.
CloudFormation	AWS-native infrastructure provisioning using JSON or YAML templates. Tightly integrated with AWS services and IAM for secure infrastructure management.

19. Containerization Basics

Containers are lightweight, isolated execution environments that package an application together with its dependencies, runtime, and configuration. Unlike virtual machines, containers share the host operating system kernel, making them significantly faster to start and more efficient in resource utilization. For SRE teams, containers provide a consistent, portable unit of deployment that reduces environment-specific failures and simplifies scaling.

Docker Image

An immutable, read-only template that contains everything needed to run an application — code, runtime, system tools, libraries, and settings. Images are built from Dockerfiles and stored in registries for distribution and reuse.

Docker Container

A running instance of a Docker image. Containers are isolated from each other and from the host system, providing predictable behavior regardless of the underlying infrastructure. They can be started, stopped, and destroyed in seconds.

Dockerfile

A text file containing the step-by-step instructions used to build a Docker image. Dockerfiles are version-controlled alongside application code, enabling reproducible builds and auditability of exactly what is running in production.

Portability

Run anywhere — laptop, staging server, or cloud — with identical behavior. Eliminates environment inconsistencies.

Scalability

Spin up dozens of container instances within seconds to handle traffic spikes without manual provisioning.

Isolation

Each container operates independently, preventing dependency conflicts and limiting the blast radius of failures.

Faster Deployment

Container startup times measured in milliseconds enable rapid scaling and near-zero-downtime deployments.

20. Kubernetes Basics

Kubernetes (K8s) is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications. For SRE teams managing production workloads at scale, Kubernetes provides a self-healing, declarative runtime environment that dramatically reduces the operational overhead of running containers in production.

Pod

The smallest deployable unit in Kubernetes. A pod encapsulates one or more containers that share network and storage resources and are always scheduled together on the same node.

Node

A worker machine — physical or virtual — that runs pods. Each node is managed by the control plane and runs the kubelet agent, container runtime, and kube-proxy.

Cluster

A group of nodes managed by a single Kubernetes control plane. The cluster is the fundamental unit of deployment for Kubernetes workloads in production.

Deployment

A Kubernetes object that declaratively manages the desired state of an application, including replica count, update strategy, and rollback behavior.

Auto-Scaling

Horizontal Pod Autoscaler dynamically adjusts pod count based on CPU, memory, or custom metrics.

Self-Healing

Kubernetes automatically restarts failed containers, replaces unhealthy pods, and reschedules workloads from failed nodes.

Load Balancing

Built-in service discovery and load balancing distribute traffic evenly across healthy pod replicas.

Rolling Updates

Zero-downtime deployments by gradually replacing old pod versions with new ones while maintaining availability.

21. Reliability Engineering

Reliability engineering encompasses the design patterns, operational practices, and architectural decisions that make systems capable of withstanding failure. In distributed production environments, failure is not a question of "if" — it is a question of "when" and "how much." Reliability engineering ensures that when failures occur, their impact on users is minimized and recovery is rapid and systematic.

Reliability Practices

Redundancy

Deploy duplicate systems so that no single component failure can take down the entire service. Redundancy is the foundational reliability primitive.

Failover Mechanisms

Automated systems that detect primary component failure and switch traffic to backup systems without human intervention or user-visible impact.

Disaster Recovery

Documented and regularly tested plans for restoring full service capability following catastrophic failures, data loss, or regional outages.

Backup Management

Regular, automated backups with tested restore procedures. A backup that has never been tested is not a backup — it is a hypothesis.

Capacity Planning

Proactive forecasting of resource requirements to ensure infrastructure can absorb expected and unexpected demand spikes without degradation.

High Availability Concepts

Concept	Description
Redundancy	Duplicate systems eliminate single points of failure across compute, storage, and network layers.
Failover	Automatic switching to backup systems when the primary fails, maintaining continuity for users.
Load Balancing	Distributes traffic evenly across healthy instances, preventing any single instance from becoming a bottleneck.

22. Capacity Planning

Capacity planning is the process of forecasting future infrastructure demand and ensuring that sufficient resources are available to meet that demand before users experience degradation. It bridges the gap between current operational state and future growth — preventing both the waste of over-provisioning and the reliability risk of under-provisioning. For SRE teams, capacity planning is a proactive, data-driven discipline that is central to maintaining SLOs as systems scale.

Effective capacity planning requires understanding not just current resource utilization, but growth trends, seasonal patterns, and the potential impact of product changes on infrastructure demand. Capacity plans should be revisited regularly and updated as traffic patterns evolve.

CPU

Processing Demand

Track CPU utilization trends across services to identify compute bottlenecks before they impact latency or availability.

RAM

Memory Usage

Monitor memory consumption patterns to forecast when additional capacity will be required and prevent OOM failures.

NET

Network Traffic

Measure ingress and egress bandwidth to identify connectivity saturation risks as traffic grows.

DISK

Storage Growth

Project data volume growth to ensure storage capacity is provisioned well ahead of saturation thresholds.

Capacity planning metrics should be tracked with sufficient historical depth — typically 3–6 months — to identify meaningful trends and seasonal variations. Point-in-time snapshots are insufficient for reliable forecasting.

23. Security in SRE

Security is an integral dimension of reliability, not a separate concern. A system that is compromised, misconfigured, or exposed to unauthorized access is inherently unreliable — even if its uptime metrics appear healthy. SRE teams share responsibility for the security posture of the systems they operate, and security considerations should be embedded into every aspect of SRE work: from infrastructure provisioning to incident response.

Security Best Practices

01

Least Privilege Access

Grant only the minimum permissions required for a system or user to perform its function. Over-provisioned access is one of the most common sources of security incidents.

02

Patch Management

Maintain a systematic process for applying security patches to operating systems, runtimes, and dependencies in a timely manner.

03

Secrets Management

Store API keys, credentials, and certificates in dedicated secrets management systems — never in code repositories or environment variables.

04

Encryption

Enforce encryption at rest and in transit for all sensitive data. Default to encrypted connections for all service-to-service communication.

05

Vulnerability Scanning

Continuously scan container images, dependencies, and infrastructure configurations for known vulnerabilities as part of the CI/CD pipeline.

Security Monitoring Areas

Area	Example Signals
Access Logs	Unusual login times, failed authentication attempts, access from unexpected IP ranges indicating unauthorized access attempts.
Network Traffic	Unexpected outbound connections, large data transfers, port scans, or traffic from known malicious IP ranges.
System Changes	Unexpected configuration modifications, new processes, changed file permissions — indicators of configuration drift or compromise.

24. Chaos Engineering

Chaos engineering is the discipline of intentionally introducing controlled failures into production or production-like environments to test system resilience and validate that reliability mechanisms — such as failovers, auto-scaling, and circuit breakers — behave as expected under real failure conditions. The philosophy originates from Netflix's Chaos Monkey project and has since become a core practice in mature SRE organizations.

The key word is "controlled." Chaos engineering is not random destruction — it is a scientific approach to failure injection with clearly defined hypotheses, blast radii, and rollback procedures. The goal is to discover weaknesses before they manifest as unplanned incidents in production.

Validate Resilience

Confirm that redundancy, failover, and self-healing mechanisms function correctly when subjected to realistic failure conditions — not just in theory or in documentation.

Identify Weaknesses

Surface single points of failure, hidden dependencies, and resilience gaps that would otherwise only be discovered during a real, high-stakes production incident.

Improve Recovery Processes

Test and refine runbooks, escalation paths, and recovery procedures under controlled conditions — building muscle memory for effective incident response.

Common Chaos Testing Scenarios

Server Shutdowns

Terminate instances without warning to verify that load balancers reroute traffic and autoscalers replace capacity automatically.

Network Latency Simulation

Inject artificial latency into service-to-service calls to test timeout handling, retry logic, and circuit breaker behavior.

Database Failures

Simulate primary database unavailability to verify that read replicas promote correctly and applications handle connection failures gracefully.

25. Cloud Reliability Basics

Cloud platforms provide a rich set of managed services and architectural patterns that SRE teams can leverage to build more reliable systems with lower operational overhead. However, using the cloud effectively for reliability requires deliberate architectural choices — availability zones, multi-region deployment, managed services, and auto-scaling are tools that must be consciously designed into a system, not inherited by default.

Auto Scaling

Cloud-native auto-scaling dynamically allocates and releases compute resources based on real-time demand signals. Properly configured, it ensures systems maintain performance during traffic spikes without over-provisioning during quiet periods.

Multi-Region Deployment

Distributing workloads across multiple geographic regions provides disaster resilience against regional outages — whether caused by infrastructure failures, natural disasters, or cloud provider incidents.

Managed Services

Cloud-managed databases, queues, and caches offload operational overhead — patching, backups, failover — to the cloud provider, allowing SRE teams to focus on application-level reliability rather than infrastructure maintenance.

Major Cloud Platforms

AWS

Amazon Web Services — the largest cloud platform, offering the broadest set of services and global infrastructure footprint.

Azure

Microsoft Azure — strong enterprise integrations, hybrid cloud capabilities, and deep Microsoft ecosystem alignment.

Google Cloud

Google Cloud Platform — where SRE originated. Particularly strong in container orchestration, data analytics, and AI/ML services.

26. SRE Metrics and KPIs

Measuring SRE effectiveness requires a well-chosen set of key performance indicators that reflect both the current reliability state of systems and the efficiency of the engineering team that operates them. These metrics serve two audiences: engineering teams that use them to guide technical decisions, and business stakeholders who need to understand reliability in terms of customer impact and organizational risk.

KPI	Purpose	Interpretation
Uptime Percentage	Availability tracking	The fraction of time a service is fully operational. Expressed as a percentage of total measurement window duration. Directly related to SLO compliance.
MTTR	Recovery speed	Lower MTTR indicates faster, more effective incident response. Target reductions through better tooling, runbooks, and on-call training.
MTTD	Detection efficiency	Lower MTTD reduces the window of user impact before response begins. Improved by better alerting, reduced alert fatigue, and comprehensive monitoring coverage.
Incident Count	Reliability trends	Track incident frequency over time by severity level. Increasing incident counts signal systemic reliability degradation requiring engineering investment.
Deployment Frequency	Delivery efficiency	How often production deployments occur. Higher frequency with stable reliability indicates a healthy CI/CD pipeline and mature engineering culture.

Track these KPIs over time rather than as point-in-time snapshots. Trends are more meaningful than absolute values — a rising MTTD is a more important signal than a specific MTTD number, regardless of whether it crosses a threshold.

27. SRE Team Responsibilities

SRE teams occupy a unique position at the intersection of software engineering and operations. Their responsibilities span a broad range of technical and organizational domains, requiring deep expertise in distributed systems, software development, and operational practices. Understanding the full scope of SRE responsibilities is essential for staffing, evaluating, and growing an effective SRE organization.

Effective SRE teams balance reactive work — incident response and on-call duties — with proactive engineering investment in reliability, automation, and observability improvements. The 50% toil ceiling is not merely a guideline; it is a structural safeguard that prevents operational work from crowding out the engineering work that makes systems more reliable over time. Engineering managers should monitor toil levels closely and protect engineering time aggressively.

28. Common SRE Tools

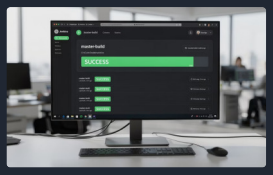
The SRE toolchain spans the full operational lifecycle — from infrastructure provisioning to incident response. While specific tool choices vary by organization and cloud environment, the categories remain consistent across mature SRE practices. Understanding the purpose and trade-offs of tools within each category is as important as knowing how to operate any individual tool.

Monitoring

Prometheus is an open-source metrics collection and alerting system with a powerful query language (PromQL) and native Kubernetes integration. **Grafana** provides rich dashboarding and visualization on top of Prometheus and other data sources. Together they form the most widely adopted open-source monitoring stack in SRE.

Logging

The **ELK Stack** (Elasticsearch, Logstash, Kibana) provides scalable log ingestion, storage, and search. **Splunk** offers enterprise-grade log analysis with powerful search, alerting, and compliance features. Both are widely used for operational log analysis and security monitoring.



CI/CD & Orchestration

Jenkins is the most widely deployed open-source CI/CD automation server. **GitLab CI** provides integrated pipelines within the GitLab platform. **Docker** standardizes container builds and distribution. **Kubernetes** orchestrates container deployments at scale with self-healing and auto-scaling capabilities.

29. SRE Best Practices

The following best practices distill the core guidance from the SRE discipline into operational and strategic principles that teams can apply regardless of organization size, cloud environment, or technology stack. They represent the accumulated wisdom of mature SRE organizations — the practices that consistently differentiate high-performing reliability teams from those struggling to keep pace with operational demand.

Operational Best Practices

01

Automate Repetitive Tasks

Every recurring manual task is a toil debt that compounds with scale. Automate aggressively and track toil reduction as an engineering metric.

02

Monitor Proactively

Build monitoring coverage that surfaces problems before users experience them. Reactive monitoring is insufficient — aim for detection before impact.

03

Define Clear SLOs

Every production service should have documented SLOs that are reviewed regularly and used to drive engineering prioritization decisions.

04

Conduct Postmortems

Every significant incident is a learning opportunity. Blameless postmortems with tracked action items are the primary mechanism for improving reliability over time.

05

Improve Observability

Continuously invest in logs, metrics, and tracing. The ability to understand system state in novel failure scenarios is built incrementally over time.

Strategic Best Practices

01

Build Reliability Culture

Reliability is a shared organizational value, not the exclusive responsibility of the SRE team. Foster a culture where all engineers consider reliability in their daily work.

02

Balance Innovation With Stability

Use error budgets as a shared decision framework for balancing feature velocity with reliability investment. Avoid both over-engineering and reckless deployment.

03

Reduce Operational Complexity

Complexity is the enemy of reliability. Actively work to simplify architectures, eliminate unused services, and reduce the cognitive load of operational systems.

04

Encourage Collaboration

The most effective SRE organizations work in close partnership with product and development teams — embedding reliability considerations throughout the software development lifecycle.

Applying these best practices consistently — not just during crises — is what separates organizations that are resilient by design from those that are perpetually reactive. Start with the practices that address your team's most pressing gaps and build from there.

GSDC
Global Skill Development Council

SITE RELIABILITY ENGINEERING (SRE) FOUNDATION CERTIFICATION (CSREF)

ABOUT GSDC CERTIFICATION

EBOOK

Extensive and exclusive Ebook created by world's experts to help you with understanding core concepts.

LEARNING MATERIALS

Get access to learning materials such as videos, ebooks, templates, and practice exams, which will help you clear the certification exam.

CREATED BY EXPERTS

GSDC certifications are created and authored by world's leading experts in the field.

LEARNING OBJECTIVE

- Gain insights into autonomous decision-making processes
- Apply knowledge using ready-to-implement templates
- Demonstrate ability to work with Agentic AI models
- Validate your skills wit

Enroll now with the code **LEARN20** To avail **20%** discount

Enroll Now

www.gsdcouncil.org