⭐ Scalability + cost + resk + complexity → impln of SRE

⇓ biz. applns

## SRE Foundation 07Aug2021

⭐ Bring dev operations teams into the ecosystem earlier

DevOps: Velocity + stability

CALMS

↓

culture change

Reactive → Incident management

Proactive → Event monitoring

predictive

Reliability → do whatever you do, without any issues.

Resilience → how soon can you come back from an unplanned outage (reactive)

preventing incidents and addressing at the earliest if it occurs

Water molon syndrome: looks green from outside, red from inside

SRE: → permanent fix to user issues

is a discipline ⭐ apply soft. engg → infra + opers problems

→ how can we reduce Problems?

→ In companies, L1 → removed / incorporated chatbots.

Goal of SRE: create ultra-scalable and highly distributed S/W systems

→ Preempt → better user experience.

SREs → journey / guide Dev. & Platform team to spread awareness about SREs and build the mindset

Goal: Create ultra-scalable and highly reliable distributed software systems

50% (ops) + 50% (dev.)

(ops) → issue resolution, On-call & manual interventions

(dev.) → new features, scaling or autom^n

Monitoring + Alerting + Automation → Large part of SRE

→ Reduce "Toil" → no manual stuff → autom^n

- SRE: class that implements DevOps
  - * SRE and Devops coexist.
  - * SRE can work without Devops also.

part of → Platform Engg. Team

- SRE – Engg. discipline – achieve appropriate level of reliability
  Biz. + performance + ops insight with monitoring → Key to growing SRE role

- Scalability
  availability
  incident response
  autom^n

- Google: 50% → on the aggregate 'ops' work.
  50% → actually doing dev.

* Auto-healing → reduction in the volume of incidents

SRE ↔ * Value stream mapping (VSM) + awareness
↳ collective efforts

* SRE → experimentation

- Experience Level Agreement (Ex-LA) : new dimension to SLA.

- SRE - specific implementation of DevOps with some extensions (Google)

Six Principles: 1) operations - S/W problem
↳ S/W engg. as a discipline focusses on designing and building rather than operating and maintaining.
↳ Oversee the NFR during S/W dev.
↳ estimates: 40-90% of the TCO are incurred after launch.

"Slowness is the new love".

2) Service Levels

SLO → service level obj.
↓
SLOs need consequences if they are violated.

availability target for a product or service (never 100%)

- In SRE, services are managed to the SLOs.

Tasks can provide the "wisdom of prodn".

3) Toil:

- Kind of work tied to running a prodn service that tends to be manual, repetitive
- reduce toil → automate whatever can be automated.
- → system thinker
- → level of understanding

4) Autom: → reduce manual tasks.
+
engineering-based approach to problems
↳ by
KISS: Keep it simple, stupid.

5) reduce the cost of failure
↳ look to improve MTTR (mean time to repair)

(proactive)
MTTD: mean time to detect.
↳ detecting before incident occurs.

⭐ late problem (defect) discovery is expensive so SRE looks for ways to avoid this.

6) shared ownership
↳ (boundaries between app. dev and prodn should be removed.)

SRE's shift left and provide wisdom of prodn to devp. teams. ← SLOs → element of shared ownership

→ Greg McKeown

Essentialism → Book

Get up early → cognitive bandwidth

⇓

no gadget-day

1% improvement per single day

---

Module-II

- SLOs → service level objective: → binding target for a collⁿ of SLIs.
- tightly related → user exp.
- weekly tracked SLO → availability → goal for how well a product or service should operate.
- products / services | could have several SLO's.
- SLO is a subject of SLA.
- 100% reliability → expensive, technically complex

---

* customer journey exp. (going to Amazon and purchasing)
* functional reqs (feature teams)
* NFRs (oper$^n$/infra. teams)
* SLOs: → defined by SRE and product mgmt.
  - → internal target
  - → improvement over own perf. and enhance the benchmark.

moving target

SLO → become the new SLA. with increased user-experience.

depends on
SLO's → business
↳ set according to customers.

SLA – 95% (Contractual agreement)
SLO – 96% (Business)
Error Budget = 100 – SLO
or
budget of unreliability (quantitative measure shared between product and SRE teams to balance innovn & stability)

SLO – 95%
Error budget – 100 – 95% = 5%

breach SLO → increase to error budget → Consequences

calculated every day.

Consequences → (stop deployments.) → Biz. profit period

Dev. Team + SRE Team meet daily to outline their improvements.

Bias to be exhausted.

Error budget → balance between velocity & stability

→ assigned every month or quarter.

→ difficult to achieve.

quantitative risk measure to understand sys. behaviour.

most popular SLOs – Availability (1%) → error-budget
Latency (2%)
Error rate (1%)

does the service deliver in a user-acceptable period of time?

Missed SLO: lost revenue, drop in employee productivity, dissatisfied customers, social media backlash
impact on biz. perf.

SLI → (Service level indicator) quantitative measure of an attribute of the service
eg throughput
latency
failures per request

Error-budget:

Bad
( Error-budget as going over budget
↓
work overtime or respond to out of hours issues.

Not hitting 99.9% of HTTP req. in a month → scalability issue

Good
↓↓
strategically burn the budget to zero every month
↳ good velocity without comprising availability

whether it's for feature launches or arch. changes

← Fixed Error Budget:
↳ encourage the Lean preference for small changes to stay within the error budget.

High-risk deployments or large big-bang changes have more likelihood of issues and more chance of the error budget being blown.

Consequences of missed SLO's: no new deployments → sprint planning may only pull post-mortem action items from the backlog.
Dev Team + SRE Team ↳ outline their improvements.

VALET Dim's of SLO
(Volume / availability / latency / errors / tickets)

- SRE is fundamentally more about people and processes than tech.
- purpose of SRE: bridge the gap between platform design, development and operational execution.
- Earlier we catch issues, the cheaper they are to fix.

- SRE implements DevOps → (class)

(Google) 5 pillars:
1. reduce org^n silos
2. accept failure as normal
3. implement gradual changes
4. leverage tooling & autom^n
5. measure everything.

DevOps
↳ set of principles & culture guidelines that helps to breakdown the silos between devp. & operns

SRE
↳ a set of practices with an emphasis of strong engineering capabilities that implement the Devops practices

Google: SRE is a specific impl^n of DevOps with some extensions.

- measuring reliability:
  availability = actual system uptime / anticipated sys. uptime

---

08/08:

- SRE focusses mostly on post-prod^n responsibilities
- Goal of SRE: to create ultra-scalable and highly reliable distributed s/w systems.
- Wisdom of prod^n → wisdom gained from something running in prod^n

---

Toil → manual → constant password resets
repetitive → dealing with interrupts
automatable → phy. meetings to approve prod^n deployments
tactical ← Known workarounds
devoid of enduring value : constantly review the tasks to check the value.
linear scaling → (manual infra. scaling)
* on-call responses

Toil → Bad → slow progress
poor quality
career stagnation
attrition
unwinding
burnout

extracting some data

impact of high toil

→ Engineers doesn't work 100% on toil.

* Engineering Bankruptcy: → The level of toil in an org. can increase to a point where the orgn won't have the capacity needed to stop it.

What can be done about Toil?
- ext. automn (scripts, automn tools)
- int. automn (automn delivered as part of the service) enhancing the service to not require intervention
  - need fresh.
  - process engineering

- identify toil / effort estimation (operational work)
- Add Toil to the product backlog

- making engg. time available

50% (ops)

50% (engg. tasks) → reduce future toil, add need features

→ one team or person does not become the "ops".

- Slack: committed to the imp. of reliability over feature velocity

---

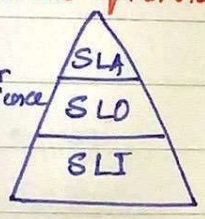
Monitoring & SLI: → ways for engg. to communicate quantitative data about systems.

System monitoring is the use of H/W or S/W component to monitor the sys. resources and perf. of a computer system.

SLO → target (internal) based on user experience

SLI → parameter to measure
  → actual quantitative data

→ ~~estimated~~ expected




- Error Budget → boundary
- Monitoring SLI's → will tell if we are meeting a SLO or not
  → also tell how much of our error budget is left.

SLI Measurement:

$$SLI = \frac{\text{no. of good events}}{\text{total no. of events}}$$

may need client-side data coll$^n$ along with server-side metrics. (Prometheus) → miss range of problems

beyond (consider other sources) → load balancer determined by the app. front-end monitoring/test infra.

- SLI —measured→ per month

- SLI —tells us→ whether we are meeting SLO.

- Mikey Dickerson's hierarchy of service reliability:

Product dev.
capacity planning
test/release process
post mortems
incident response

Monitoring → identify what needs urgent attention and helps prioritization:

urgency
trends
planning
improvements

- Telemetry: highly automated comm$^n$ process by which measurements are made at remote points and transmitted to receiving equipment for monitoring.

remote measure : agent monitoring

→ strives to detect and diagnose appln perf. problems to maintain an expected level of monitoring and mgmt. of s/w

APM: Appln Perf. monitoring → perf. and availability of s/w applns

DynaTrace  AppDynamics

- Monitoring Anatomy:

  Graphing: Kibana, Graphana

  Agent & Agentless

  to probe, beyond OS level parameters,

  agents → installed on h/w hosts to be monitored

  UI → dashboards of SLO's and SLI's

  alert anamoly det$^n$

  graphing Core (config$^n$ of hosts)

  Check threshold against metrics collected

  aggr$^n$ across a time horizon

- SLI Supporting Tools:

  Monitoring - Prometheus, Splunk, Catchpoint, Nagios

  Graphing - Grafana, Collectd

  Logging - Logstash, Rsyslogd, Collectd

  Alerting - Pagerduty

- Monitoring & Observability: → debugging / trace analyses (reactive)

  (reactive) all

monitoring myth: We can't anticipate these wrong cases coolb distributed, complex services running at scale

set of symptoms → what's broken

to di:

→ alerting
→ escalation

Ops Team

Externalizing all the o/ps of a service allows us to infer the internal state of that service. (observable)

→ regular feedback or probing is a must.

→ observability

Data collected by monitoring is used in diagnostics/RCA

Monitor applns to detect prob. & anamolies.

→ (causes - why?)

observability → should be pro-active
↳ goes with monitoring.

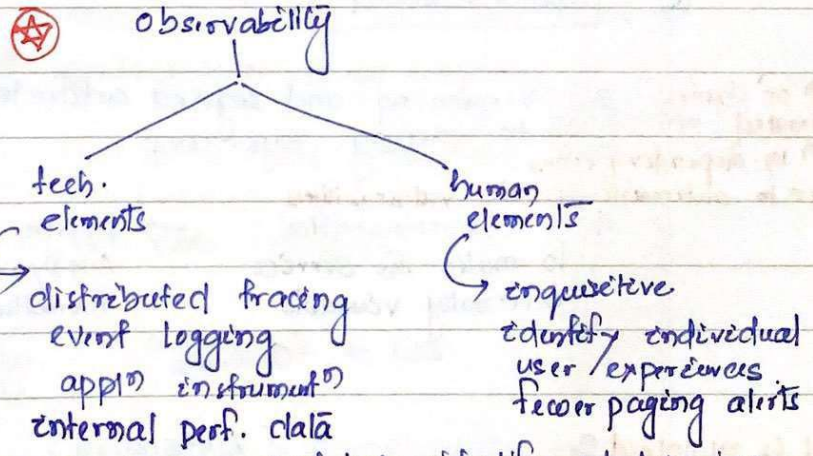
↓
(on the fly)

Why observability is important?

1) rapid rate of service growth
2) dynamic arch.
3) Container workload
4) dependencies between services
5) customer exp. matters more

- Bi-model IT
  - Legacy
  - Modernised.

- observability = better alerting
  ↳ improve signal/noise ratio to focus alerts on key issues




- SLO's → user perspective and help identify what is imp.

SLI's → give detail on how we are currently performing

Observability → gives use the normal state of the service

General Automat$^n$ Focus: Shift right, DevOps

SRE Tools and Automat$^n$:

For SRE, autom$^n$ is a force multiplier, not a panacea.

eliminate toil

improve SLO's

→ Consistency
fastest act$^n$ time savings
a platform upon which to build

big Dev

small ops

* appr. tooling
* engg. effort
* measurable outcomes

SRE-Led: Service Autom$^n$: (Shift Left)

Env:s

1. Infra/Config as code → Terraform, Azure Res. Mgr.
   ↓
   Ansible, Puppet, Docker

Consistent, repeatable, prod$^n$ ready
easy to test and audit changes
easy to reproduce errors in test env.

2. Automated funct$^n$al & non-funct$^n$al tests in pipel$^n$
   ↓
   extended build pipeline → (funct$^n$al) → Selenium, cucumber
   " test "
   ↓
   (non-funct$^n$al) → JMeter, SoapUI, Nagios

Systems are well-tested but issues can be found in prod$^n$.
cost of change is lowered
risk of regn is reduced

Spotify model

prom$^n$ of change automated
regn in dependency errors easier to determine

3. Versioning and signed artifacts to deploy sys. comp. → Artifactory

security vulnerabilities

4. Instrumentation to make the service externally viewable → Dynatrace, AppDynamics, Prometheus.
   SLI → OpsGenie

sec. & audit events can assist with proactive monitoring
reduce mean time to by granting dev. → read only access to logs

toil is minimized
upfront
reduced rework
reduced service TCO

5. future-growth Envelope (Autoscaling databases)

6. anti-fragile strategy: → adm. activities

availability & integrity risks to sys. are appropriately mitigated.

DR – Fire Drills
Chaos Engg. – Chaos monkey
on call mechanism – PagerDuty, Squadcast

mitig$^n$s evidenced and tested reducing risk

can't be implemented

Chaos Engg. → without preprod same as prod.

↓

Netflex

↓

limiting the blast radius

- Hierarchy of Autom$^n$ Types:
  1. None
  2. Ext. maintained sys. specific autom$^n$
  3. Ext. maintained generic autom$^n$
  4. Internally maintained system-specific autom$^n$
  5. Systems that require no intervention

- Secure Autom$^n$: (DevSecOps) → must for SRE.
  - ↳ removes the chance of human error (or willful sabotage)
  - → build-test-deploy pipeline.

→ Pipeline can be validated and checked for compliance

- SRE places extra emphasis on security of prod$^n$.

- Secure Build: digitally signing build artefacts avoids possibility of "fake" code.
  - ↳ proper conf$^n$ mgmt → Huge role in compliance.

- Secure Test: (does not change)
  Test envs are immutable and are created, guaranteeing compliance with code repository.

- Secure Staging: → (immutable)
  - ↓
  - data security considers: GDPR, PCI
  - dedicated tool and uncover security scanning → (vulnerabilities) or Proxy security req.
  - ✱ dependencies and integr$^n$ to other services may introduce vulnerabilities.

Hard Work Beats Talent - Book

- Secure Prodⁿ: → immutable
  dedicated security scanning → try and uncover ~~remediate~~ vulnerabilities.

- (172-195): Not reqd. for exam

---

## Anti-Fragility and Learning from Failure

Fire-fighting: requires lot of training → plan for the unexpected in training

Embracing failure will help improve → optimize monitoring

MTTD and MTTR metrics. (components)

mean time to (failure/incidents) detect

MTRS — mean time to recover (service)

RPO - recovery point obj.

Chaos Engg. → identify key components & dependencies across services.

The Third Way:

* 1) continuous experim$^n$ & learning
  2) understanding that repetition and practice is the prerequisite to mastery.

Introduce faults into the sys. to increase resilience

* Safe experim$^n$ and innovation (Hackathons)

* Pathological (Power-oriented) → failure is covered up.
  Bureaucratic (Rule-oriented) → org$^n$ is just and merciful.
  Generative org.: (Perf. oriented) → Learning (failure causes enquiry)

Introduce Fire Drills → annual data center failover test → good 1st step towards C.E

↓

loss of facility / tech. / resources / critical 3rd-party vendors or region

Chaos Engg. → discipline of experimn on live prodn system. in order to build confidence in the system's capability to withstand turbulent and unexpected cond^n's

e.g. Simian Army

↓

minimize the blast radius → outage impact → very minm.

* minimize impact on customer exp. * automated self-healing due to less on-call incidents

## Typical SRE Adoption Steps:

### Orgn Impact of SRE

SRE approaches (shift left)

- Consulting → specialist advice / no-hands-on delivery involvement
- Embedded → some movement towards shared responsibility, ownership of SRE → service team
- Platform → SRE takes critical on-call
- Slice & Dice → one size fits all approach
- Full SRE → SRE "owns" the deployment platform and provide on-call

↓

Reliability (first class citizen)

shared respons. & shared "on-call"

↳ owns sect^n of service typically appln & infra. on-call responsibility → shared

Google → recommends → 25% on-call.

Blameless Post-mortem: → P1 or P2 incidents → Self-healing

Reasons for Blameless PM: monitoring failure, data loss of some kind, etc. on-call engineer contention

## SRE & other frameworks

SRE/Agile ← SRE does not stand alone: Tool, Backlog, def^n of done → more prodn focussed.

SRE/DevOps → Pipelines of delivery go further. | custom^n: more widespread & consistent

Agile: Biz. + del.

DevOps: CICD

SRE: Stability + Reliability

ITSM: orgn learning across value stream

* SRE can help with ITSM compliance activities through automn & integrate
  - Failure → learning opp.
  - cont. learning

* ITSM Process models support SRE.

Trends in SRE:

1) Failure as the new normal
2) Automn as a service
3) Cloud is King
4) observe and learn
5) evolution of Network Engineer

NRE: → (applies an engg. approach to measure and automate resiliency of networks)
→ uses Chaos Engg. to test the reliability of networks

DBRE: → applies C·E to the database to confirm failover and restore responses → moves to managed DB services

CRE: → creats shared resp. across supplier and customer
applies an engg. mindset to cust. support

HRE: → establishes SLO's focussed on legacy processing eg batch to RE-platforms onto modern arch.

---

* On-call must be undertaken by Opern's and engg. teams.
* incident respone must be 'sustainable'.
* Change
  Config
  release
  incident
  problem
  } mgmt. (ITIL processes) → SREs are involved.
* Providing 24/7 support → req. 08 SREs
* General Autom Focus: 1) Features are being pushed to those supporting prodn on ever increasing numbers.
  2) developers assume that envs are consistent.

3) Testing step introduce false confidence as prodⁿ is always unique.

4) monitoring & alerting → focussed on things that are known to go wrong.

* In SRE-Led service automⁿ, overall there is focus on prod.
DevOps gains "wisdom of prodⁿ".
SRE can say "No".

* Synthetic monitoring: ability to monitor service behaviour by creating scripts to simulate the actⁿ or path taken by a customer/end-user and the associated outcome.

* C.E:
1) segregate sys. into key comp.
2) Test the sys. without key comp. available
3) Break the sys. (Non-prod env.s first)
4) introduce failure of key components (in-prod)
5) introduce DB failure
6) introduce total sys. failure in prod

Next Steps:
1) Holistic logging
2) identify dependencies
3) improve by error-handling and recovery
4) learn from real failures

Automated Recovery:
1) Create immutable IAC
2) system state & functⁿal code with automated tests
↓
do not allow ssh in prodⁿ

Kubernetes / AWS auto-scaling
- detect compromised instances of servers or containers
- automatically scale infra/appⁿs up and down
- integrate with monitoring services.

← self-healing capabilities

- True North Vision: impossible to achieve 100% up-time

- Blameless postmortems: making sure that the same issue doesnot repeat.

↓

those involved on the failure can give a detailed acct. of ... without punishment.

- Postmortem Outputs:

details of the incident
follow-up actⁿ
lessons learnt
a time-line of what happened
any supporting infoⁿ

- key success factors for SRE Adoptⁿ:

i) Exec support and buy in
ii) Funding
iii) Good relⁿship across spectrum
iv) an growing org.

- SRE approach to platform growth: Containⁿ + cloud + Nw + AS-a-service

- SRE approach to scope growth: Platform SRE
shift-left
automating toil

- SRE approach to tecket growth: Tool req$^n$ (auto-ticket response) + DRY (don't repeat yourself)
  ↓
  Sol$^n$ to eliminate repet$^n$ of prob.

# Chaos Engineering

- 2017: 01 hr of downtime → cost over $100,000 to biz.
  Rise of microservices & distributed cloud arch. → web has grown increasingly complex.
  → companies need a soln. and want to avoid next costly outage.
  → Chaos Engg.

- Chaos Engg: disciplined approach to identify failures before they become outages.
  ↓
  * By proactively testing how a sys. responds under stress, you can identify and fix failures b1fore they end up in the news.
  * lets you compare what you think will happen to what actually happens in your systems.
  * 'Break things on purpose' → to learn how to build more resilient sys.

- 2010: Netflix Engg. Tools Team → created → Chaos Monkey
  ↓
  created to move
  On-prem infra. → cloud infra (AWS)
  and the need to be sure that loss of an Amazon instance wouldn't affect the Netflix streaming experience.

2011: Simian Army —added→ additional failure inj$^{n}$ modes on top of Chaos Monkey

⇓

design a cloud infra. where individual components can fail without affecting the availability of the entire sys.

2012: Netflix shared the source code for Chaos Monkey on GitHub.

Failure Inj$^{n}$ Testing (FIT) —based on→ Simian Army but provided developers more granular control over the blast radius of their failure inj$^{n}$.

2016: Gremlin: (Founded) by Kolton Andrus

2020: AWS —C.E adds→ AWS Well-architected Framework (WAF)

AWS → Fault Inject$^{n}$ Simulator (FIS)

⇓

fully managed service for natively running chaos exp. on AWS services

2021: Gremlin: publishes 1st ever State of Chaos Engg. report

⇓

growth of C.E

benefits

frequency of running chaos exp.

C·E runs thoughtful & planned experiments that teach us how our sys. behave in the face of failure.

Principles:

These exp·s follow 03 steps.

1. Plan an experiment: Create a hypotheses. What could go wrong?
2. Contain the blast radius: Execute the smallest test that will teach you something.
3. Scale or squash: Find an issue? Job well done. Otherwise increase the blast radius until you are at full scale.

- Companies: National Australia Bank
  On-prem ... ▶ ... cloud
  reduce incident counts

- C·E is a tool to build immunity in our tech. sys. by injecting harm like latency, CPU failure or network black hole to find and mitigate potential weaknesses.

Breaking things on purpose → enable detect unknown issues that could impact our sys. & cust.

2021 State of C·E Report: Most common outcomes of C·E

↓

- increased availability (>99.9%)
- low MTTR
- low MTTD
- fewer bugs shipped to product
- fewer outages

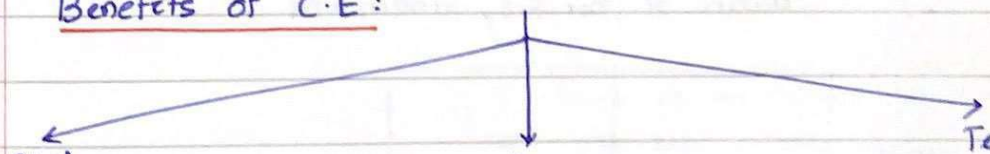
* C·Exp are more likely to have >99.9% availability.

- 8 Fallacies of distributed sys.:

1) Network – reliable
2) Latency – 0
3) Bandwidth – infinite
4) Network – secure
5) Topology – doesn't change
6) 01 Admin
7) Transport cost – zero
8) Network – homogenous

These fallacies drive the design of C·E.

- Benefits of C·E:



**Customer**

↓

increased availability
durability of service

**Biz.**

↓

prevent large loss in revenue & maint. costs

create happier and more engaged engineers

improve on on-call training for engg. teams

improve the Incident Mgmt. prog. for the entire comp.

**Technical**

↓

Redⁿ in incidents
Redⁿ in on-call burden

increased understanding of sys. failure modes

improved sys. de

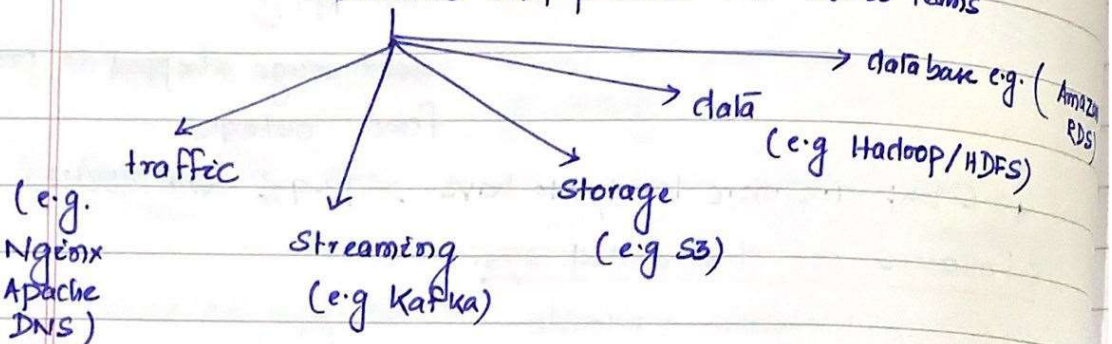
redⁿ in repeated SEVs (incidents)

- C·E For Service Teams:

dedicated Chaos Engg. Teams → small team

↓

awareness and practice C·E across teams



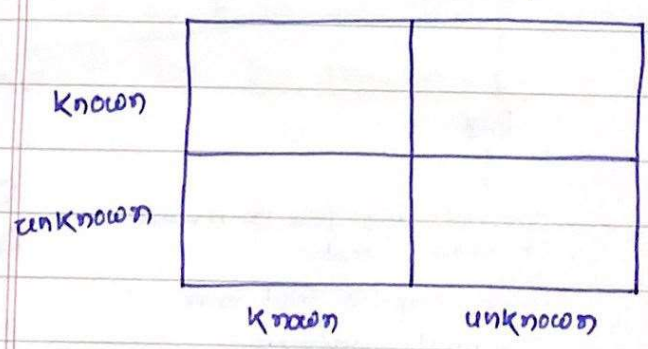

- C·E Experiments:

Experiments should be performed in the following order.

i) Known Knowns → things you are aware of and understand
ii) Known Unknowns → things you are aware of but don't understand
iii) Unknown Knowns → things you understand but are not aware of
iv) Unknown unknowns

↓

things you are neither aware of nor fully understand



- Planning for First C-Exp: 'What could go wrong'?

  - Services
  - dependencies
  - data stores
  - env.s

Create a hypotheses. → Chosen the exact failure to inject
+
What happens next?

discuss the scenario and hypothesize on the expected outcome. before running it live.

measure the impact → sys. availability
durability

⇓

KPI → relate customer success such as

orders/min

stream starts/sec

⇓

If there is an impact to these KPIs, halt the exp. immediately.

⇓

measure the failure where you want to verify your hypotheses.

impact on latency, req/sec or sys. resources.

RollBack Plan: → Backup plan (but sometimes backup plan can fail)

Go fix it: → Running your 1st C-Exp (02 outcomes)

Sys. – resilient

Problem (Fix it)

## Reducing MTTD for High-severity incidents:

- C.E: disciplined approach to identify failures before they become outages.

  ↓

  Using thoughtful experiments, you 'break things on purpose' to learn how to prevent outages and build more resilient sys.

- C. Experiments | 4 steps

  - define the sys. normal behavior - its steady state - based on measurable o/p like overall throughput, error rates, latency and so on.
  - hypothesize about the steady state behavior of an experimental group, as compared to a stable control group.
  - expose the experimental group to simulated real-world events such as server crashes, malformed responses or traffic spikes.
  - test the hypothesis by comparing the steady state of the control group and the experimental group.

- Gremlin's Failure-as-a-Service (FAAS) platform was built by Amazon and Netflix Engineers.

- High-severity incident (SEV) → used Amazon, Dropbox, Gremlin

  ↓

  availability drops
  product feature issues
  data loss
  revenue loss
  security risks

TTD - Time to detⁿ ⇒ interval from when an incident starts to the time it was assigned to a tech. lead on call who is able to start working on resolution or mitigation.

* SEV 0s (highest and worst kind of severity)
  ↓
  industry-recommended (able to detect and resolve within 15 mins)

  ⇒ SREs has to improve the MTTD.

* High-impact methods to reduce MTTD:
  - Incident classfⁿ and leveling
  - Tooling
  - Monitoring
  - KPIs
  - Alerting
  - Observability
  - C.E

Step 0: incident classfⁿ, including SEV descriptions and levels, SEV timeline and TTD timeline

Step 1: critical-service monitoring including key dashboards and KPI metric emails

Step 2: service ownership and metrics, including measuring TTD by service, service triage, service alerting

Step 3: on-call principles, including the pareto principle, rotation structure, alert threshold maintenance and escalation practices

Step 4: chaos engg., including chaos days and continuous chaos

Step 5: self-healing sys., including when automation incidents occur, monitoring and metrics for self-healing sys. automn

Step 6: listening to your people and creating a high-reliability culture

---

STEP 0: Incident Classification:

SREs → determine the SEV level of an incident and then triage and route the incident to an appropriate TL.

Industry-standard (15 mins) — SEV 0s

- 5 mins (MTTD for critical services)
- 10 mins (Resolution or mitigation for SEV 0s)

SEV 0 → catastrophic service impact
SEV 1 → critical device impact
SEV 2 → high service impact

Incident-Mgmt. Program

IM
↓
* resolve SEVs in a safe and fast manner (call leader)
* focussed on TTR

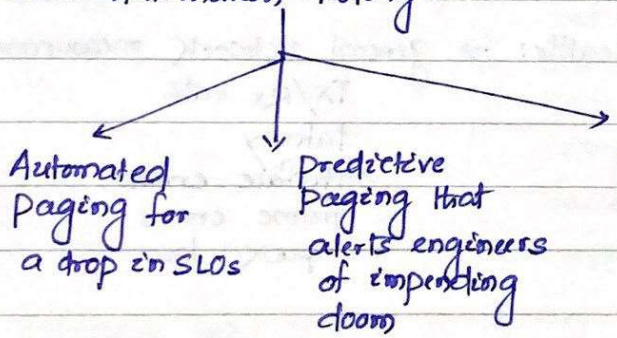
TL
* diagnose and resolve SEV as quickly and safely as possible
* TL leads → C. Exp
↓
improves mean time between failure (MTBF) and mean time to prevention (MT

* One primary IM, One secondary IM } on-call per week

- TTD: Time to detection ⇒ time interval from when the incident started to the time it was acknowledged by an engg.

TTR: time to resolution (mitigation) ⇒ introduce fix and mitigate impact of fix

- SEV Automation Tooling:



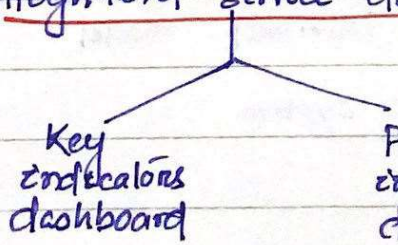

(k) Step 1: Org-wide critical service monitoring:

Monitoring → reduce MTTD for high-severity incidents

Critical-service dashboard —create→ Datadog / Grafana

Google's SRE's 'Four Golden Signals'
Tom Wilkie's RED Method
→ good baseline standards for what every critical service must have.

High-level service dashboard: ⇒ quickly drill down into health of the services.



System services health: → ancillary services which are reqd. for the infra. to be considered healthy.

- SSH
- DNS
- Puppet
- LDAP

Host General Health:

- ceptime
- CPU load
- memory usage
- I/O latency

Host Network Health: → general network measurements

- Tx/Rx rates
- latency
- interface errors
- frame errors
- packet loss

- KPI service excel: contain rolled-up daily metrics for each critical service

(*) Step 2: Service Ownership and Metrics

increasing microservices → managing services has grown increasingly complex.

Service ownership — adv. SRE practice.

RACI (matrix) → Responsible, Accountable, Consulted and Informed model

SOS → Service ownership system

* Step 3 - Uncall principles

SREs → new questions
+
review alert architecture once per quarter

* Step 4 - Chaos Engineering
↓
Series of thoughtfully planned experiments designed to reveal the weakness in our systems.

Insights from chaos exp. → red$^n$ in incidents
red$^n$ in on-call burden
increased understanding of system failure modes
improved system design
faster MTTD for SEVs and
red$^n$ in repeated SEVs.

* Effective way to start practicing C.E → Run a Chaos Day

* Steps:
Detecting Incidents Caused by Self-Healing Systems:

e.g. Restarting a stopped service

dangerous and can cause cascading failure and SEV Os without adequate testing.

Real-time eg: FaceBook's FBAR
LinkedIn's Nurse
Netflix's Janitor

Self-healing workflows: "Just like IF statements and should not be testing regularly.

* Steps: Listening to your people and creating a high-reliability culture

↑ Preoccupation with failure
Reluctance to simplify interpretations
Sensitivity to ops
Commitment to resilience
↓ deference to expertise

---

## Testing Disaster Recovery with CE:

- DR is the subset of BCP.
  - ↓ focusses on restoring the essential functns of an orgn. as determined by a BIA (Biz. impact analysis)
- DRPs (disaster recovery plans) → playbooks or run books

Black swans: outlier events that are severe in impact.

→ incidents that trigger problems that you didn't know you had, cause major visible impact and can't be fixed quickly.

- by defn, can't be predicted.
- e.g natural disaster

- C·E → practice of systematically testing sys. for failure by injecting small amount of harm.

  ↓

  Run chaos exp → running an attack against a specific system, appln or service.

  e.g. latency to network calls
  dropping packets on a specific port
  shutting down a server

* Run Books (Playbooks) behave much like disaster recovery plans by providing engineers with detailed instructions on how to restore service after an incident.

* Teams validate their runbooks by performing Fire Drills.

  ↓

  deliberately created outages meant to test runbooks under realistic but controlled circumstances.

* Game Days: Teams reproduce past incidents in order to test the resilience of their systems.

* DR: use → C·E to recreate or simulate Black swan event

* Create and validate Disaster Recovery Plan

- identifying critical assets
  - * assigning priority level to each asset
  - * dependent services
  - * identify sys. that don't have any DR or HA on place
- assessing risks and threats posed to those assets
  - * Natural disasters
  - * ext. sys. failures (widespread internet outage)
  - * 3rd-party provider outage (loss of infra. owned by a 3rd party cloud provider)
  - * human-created failures (cyber-security breach)

  Effectiveness of DRP is dependent on the recovery time.

  * RTO (Recovery Time Objective): → measure of how quickly systems become operational after a failure

  * True recovery time after an event is measured by the recovery time actual (RTA).

  * SRE, → Known as TTR.
- detailing the steps needed to replace assets after a failure
  - * DRP - comprehensive keeping RTA as low as possible
  - * Ideal Backup arch.
    - Active: Active arch. where we operate an identical replica of our prod$^n$ system on an offsite env.

    ⇓

    add. tech. complexity & double the operating cost
  - * Active-passive arch. (red$^n$ cost increase to RTA & RPA as engg. need time to make the sec. env. up
- testing and applying the plan to ensure it works as expected.
  - * Test our DRP → CE to recreate conditions such as server outages & network failures
  - * Running FireDrills → DRPs effectiveness → risky step if you don't need to CE
    - Start with small CEXP and plan incremental increase the tests

* A real incident is a true test and the best way to understand if something works. However, a controlled testing strategy is much more comfortable and provides an opportunity to identify gaps and improve.

  * In an ideal scenario, RTO & RTA will be zero.

  * RPO (Recovery Point Objective) → amount of data loss or data corruption that we are willing to tolerate due to an event.

    RPO = Last Backup Time − Time of failure (at which we assume that any data stored on the failed system is lost)

  * Actual amount of data loss is measured by the recovery point actual (RPA).

  * GitLab:
    ↓
    accidentally deleted data from their prod DB.

    - 6-hour outage (RTA)
    - 5000 proj. 5000 comments 700 user acts. (RPA)

  * Red$^{n}$ in RPO → more freq. backups or replicating continuously to an offsite loc$^{n}$ → increasing operating cost and complexity

- Validating the plan by running a FireDrill:

i) create a scenario in Gremlin (Scenarios let us accurately simulate real-world disaster conditions by incrementally increasing both the blast radius and the magnitude of an attack.

03 stages

one of the servers is immediately taken offline.

after 05 mins, the blast radius is expanded to a second server.

after another 5 mins, blast radius is expanded to the final server.

taking all three servers offline.

e.g. Simulate losing a DC or region consisting of 03 servers

BlackHole Attack:

Block all incoming and outgoing network traffic on each server for the duration of the scenario.

Shutdown Attack:

(more realistic simulation)

The Shutdown Gremlin issues a sys. call to shut down or reboot the OS on which the target is running on.

* Once 30 mins has passed, the scenario ends and all 03 servers resume normal service.

* During the scenario, track scenario start time and end time.

Gremlin

TTD / TTE (time to execute)

RTA

- Post-mortem: analyses of key metrics (RTA & RPA)

* DRP - living document that requires freq. updating and testing

---

## How to Install and Use Gremlin locally with Docker Desktop

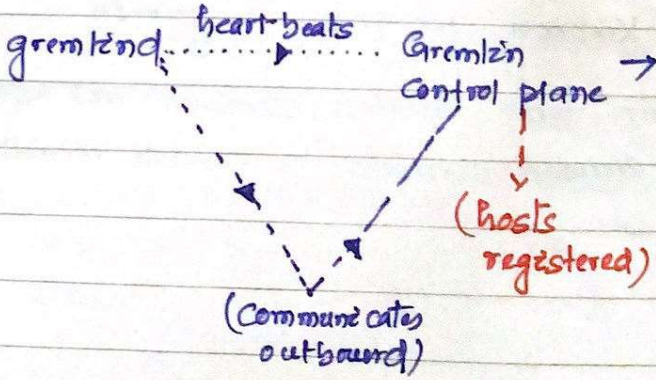
1) Install Docker Desktop
2) Verify Kubernetes cluster is up and running
   ↓
   Kubectl get nodes
3) Add the Gremlin Helm repository.
4) Create a Gremlin namespace
   ↓
   Kubectl create namespace gremlin
5) Fetch your Gremlin Team ID and Team Secret
6) Install the Gremlin Helm chart
7) Verify that your clusters are now up and running
   ↓
   Kubectl get pods -n gremlin
8) Run your first Gremlin scenario with Docker Desktop.
9) Run a Gremlin scenario to validate reliability

## Gremlin Scenarios Walk Through:

* Gremlin performs CE ⟶ through SaaS based platform
* Scenarios → Gremlin feature that allows you to map real world outage scenarios to CE exp.
* Pre-requesities:
  - i) Gremlin account
  - ii) AWS Elastic Kubernetes Service (EKS) cluster
* 1) Create and run a custom scenario → allows users to define their own scenarios.
  2) Run a recommended scenario → map real-world scenarios to CE attacks.

## Daemon & Client Overview:

* Architecture (Client <--> Daemon --> Control Plane)
* Gremlin Daemon (gremlond) → binary installed on the OS or available inside the Gremlin container.




→ to let Gremlin know that host is active and able to receive attack orders.

* Gremlin client (gremlin) → Gremlin CLI that is responsible for creating the local impact with the host

* gremlind + gremlin (daemon) (client) → this unit is referred to as a targetable client to the platform.

* Client Lifecycle:
  - → Gremlin clients (infrastructure and appln) that have been authenticated to the Gremlin Control Plane appear in the infra. clients and appln clients lists.
  - → You can run attacks on active clients.
  - → Idle client: if there is no activity for the past 5 mins.
  - → If Gremlin doesn't hear from these idle clients for a period of 24 hours, clients are removed from the list.
  - → If a client starts communicating with Gremlin again while still within the 24 hour idle window, the client is reactivated and returned to the active state.

* Logs → /var/log/gremlin
  daemon log entries → daemon.log file
  - log entries in this file may indicate events where the daemon is not able to communicate with the control plane and eventually trigger the Dead Man Switch.
  - Each attack on the host logged → /var/log/gremlin/executions
    - unique attack execution ID

* du -sh /var/log/gremlin → log size

* Bandwidth usage

Idle state
↓
daemon uses very little bandwidth.

Attack state
↓
slight increase in overall bandwidth consumptn during attacks.

The bandwidth used is not affected by the type of attack being run.

* When services discovery is enabled, the Gremlin daemon will send addn'al data and the bandwidth consumed will depend on how many services are discovered.

Measurement of actual bandwidth consumed by Gremlin for particular install$^{n}$ → use a tool → iptraf or nethogs

Command Line Interface:

* attack → starts a localized gremlin attack

e.g. gremlin attack cpu -l 60 -c 1 → attack on cpu-1 completed

* attack-container → starts an attack against a locally hosted Docker Container

→ In order to run attack-container, the container must reside on the host and

gremlin attack-container a1a9ee7eb256 -1 60 -c1

* check → inspects the state of your Gremlin install
  e.g. gremlin check os → o/p is not sent to Gremlin. Helpful to support team if we need to triage an issue with install.

* **init** → sets up the client config.
  A prompt will appear to enter the Team ID
  Team Secret
  → registers client to control plane.
  e.g. gremlin init

* logout → removes the Gremlin client from the Gremlin Control Plane.

* rollback → interrupts an active attack. It can also revert the last impact.
  e.g. gremlin rollback

* rollback-container → interrupts an active attack against a local Docker container.

* status → displays the status of any attacks.
  e.g. gremlin status
  gremlin status -u <GUID> → display the status of a specific attack

* version → state the installed version of Gremlin
  must be specified. e.g gremlin version

- C·E: Build, Fael, Learn, Repeat (BFLR)

- C·E —tool→ to better understand our systems.
  - ↓ should
    Prepare engineers for prodⁿ.

Critical role in C·E:
- Tech.
- People & Process

- C·E ⇒ uncover new infoⁿ
  - ↓
    Buy-in from Leadership

- Reliability → Technical Debt

- Path to Reliability

**Scan**
* assess what could go wrong
* evaluate risk

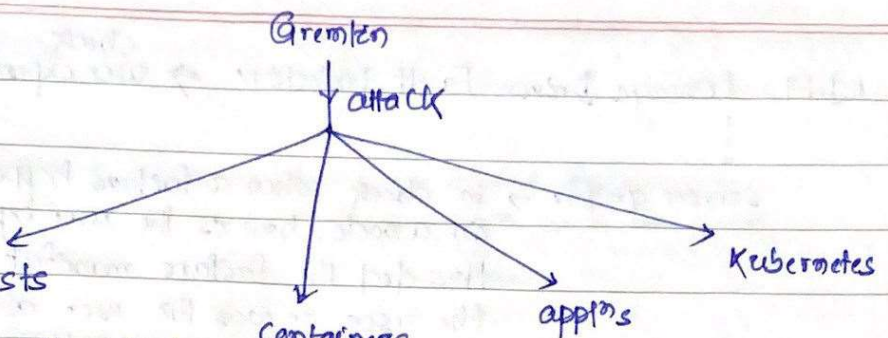
**Baseline**
* understand steady state
* check for observability gaps

**Analyze & Plan**
* baseline risk → checklist
* leverage industry patterns

**Harden**
* iterate to tune the system
* automate in CI/CD

**Report**
* surface risks
* show improvement from baseline

- Reliability — journey




(public cloud and/or data centre)

Attacks at infrastructure layer ⟹ infrastructure category : Resource – CPU, disk, IO, memory
state – process killer, ShutDown, time travel
network

↓

Black Hole
DNS
Latency
Packet loss

Attacks at appln layer ⟹ using Gremlin's alfi fabricates attacks can be configured webapp, API, CLI

- Call Leader: Person responsible for managing an incident.
  ↳ (judgement call)

- C.E → easy to do runtime validation.

- FIT (Failure Injection Testing) → tool developed by Netflix in 2014
  ↓
  allows an engg. to add a failure scenario to the request header of a class of reqs at the edge of our service.

- LDFI: Lineage Driven Fault Injection ⇒ check user experience

  ↓

  service graph → to check when a failure happens in a node, how is the user exp.? How did the failure manifest to the user or was the user able to continue doing what they wanted to do?

- Challenges in CE:

  - Apprehension (Taking risk)
  - Cultural

- C.E : History, Principles and Practice :

  Every brief outage can impact a company's bottom line, so the cost of downtime ⟷ is becoming a KPI for many engg. teams

  CEO of British Airways → tens of thousands of stranded passengers in May 2017

  ↓

  cost the company 80 million pounds

  C.E → preventive medicine

  ↓

  disciplined approach to identify failures before they become outages.

  C.E —literally→ breaking things on purpose

migration from phy. infra to cloud infra. (AWS) and the loss of an Amazon instance shouldn't affect the Netflix streaming experience.

2010 → Netflix Engg. Tools created Chaos Monkey.

2011 → Simian Army → added failure inj$^{n}$ modes on top of Chaos Monkey

2012 → Netflix shared the source code for Chaos Monkey on GitHub.

2014 → Netflix created new role called Chaos Engineer.

FIT ( new tool based on the concepts of Simian Army)

2016 : Gremlin

2020 : AWS adds CE to the reliability pillar of the AWS (Well-Architected Framework) – WAF.

AWS – announces Fault Inj$^{n}$ Simulator (FIS)

⇓

running C. Exp on AWS services

2021 : State of CE Report

* Recommended config^n for an initial Gremlin latency attack (100 ms 60 secs)

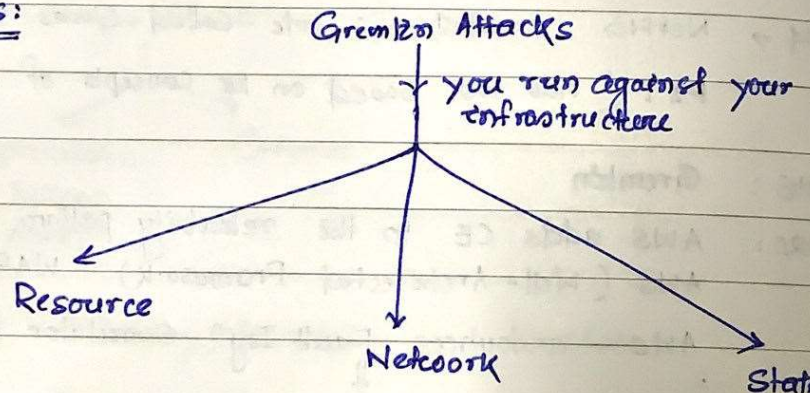
* Gremlin disk attack

* Game Day participants and roles

* What env.s do you conduct Chaos Engg.?

* CPU attack

Attacks:




Resource
- * CPU — generates high load for one or more CPU cores.
- * memory
- * IO
- * Disk

Network
- * Black Hole
- * Latency
- * Packet Loss
- * DNS

State
- * Shut Down
- * Time travel
- * Process Killer

Memory - allocates a specific amount of RAM. (malloc)

IO - generates large amounts of IO req.s to the sys. that are mounted on block devices.

Disk - fills up a target block device by writing to files at the dest^n (dest^n → --dir, no. of files written → --workers)

* Black Hole
→ drops all matching network traffic.

* Latency
→ injects latency into all matching egress network traffic

Shut down - performs a shut-down on the host OS to test how your sys. behaves when losing one or more cluster machines

Time travel - changes the host's sys. time which can be used to stimulate ^ DST adjustments

* Packet Loss
→ includes packet loss into all matching egress network traffic

* DNS: Blocks access to DNS servers
↳ Blackhole attack against port 53

Process Killer:
Kills the specified process which can be used to simulate appln or dependency crashes.
↳ doesn't work for PID
↓
shutdown attack

Game Days: * are like fire drills – an opportunity to practice a potentially dangerous scenario in a safer environment.
↓
capstone to measure reliability

* Running a GameDay tests our company from engagement to incident resolution. across team boundaries and job titles.

* verifies the sys. at scale, ultimately in Prod$^n$.

* Planning out the GameDay is a great opportunity to collaborate with other parts of the company, share context and learn about the sys. as a whole.

* Communicate → Command Center

* End Goal → to run in Prod$^n$

* coined by Jesse Robins (Amazon)