What can possibly go wrong?

Lessons in building resilient systems

By Zuodong Xiang ("Shawn")

@ Conf42

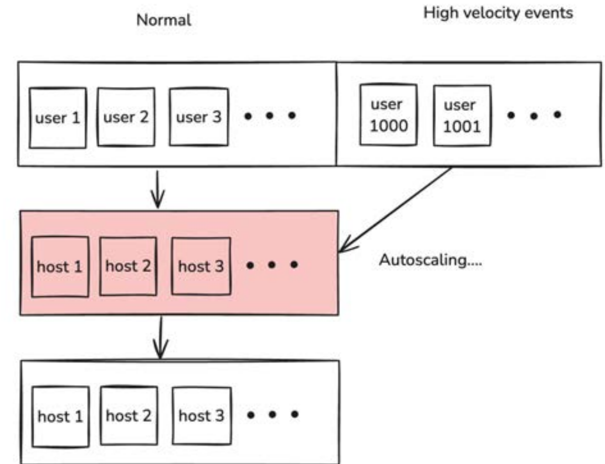
Motivation

- The cost of maintenance in software is huge
- Outage sucks for everyone, especially oncalls to wake up in the middle of the night solving Sevs
- How can prevent outage from happening on the first place?

Real sev happened and their lessons

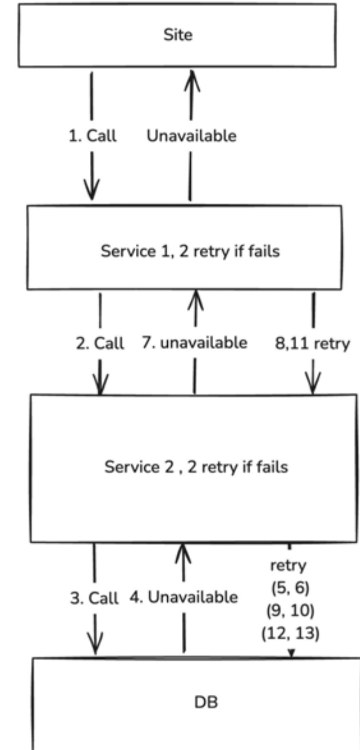
Flood of traffic

- 🚨 CPU near 100%, followed by availability drop, high client side error rate
- Unpredictable site-wide event with more traffic
- Autoscaling kicks in but host keep getting replaced due to failed health check
- All new hosts keep going down
- **Mitigation**: drop traffic by adjusting max connection to hosts
- **Lesson**: load test on max connection a host can handle



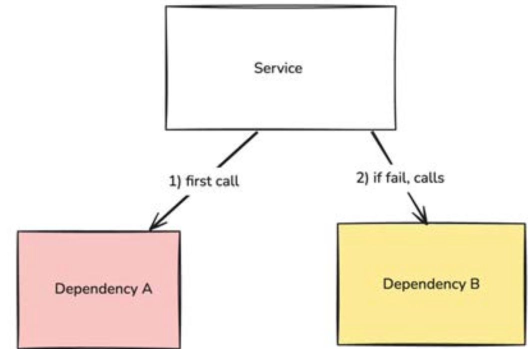
Retry storm

- 🚨Client availability below X%
- If I see an error, I retry
- Problem: If each microservice retries, number of calls to the database will multiply. 6 retries in 1 service request
- The bottom layer will easily get overwhelmed
- **Mitigation**: dropping traffic, no retry, and add more hosts
- **Lesson**:
  - Standardize retry logic. Exponential retry. Respect “too many request” http code
  - Circuit breaker on dependency: stop calling troubled dependency



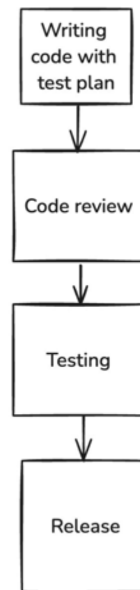
Plan B went poorly

- 🚨Dependency availability below X%
- Sometimes, there are more than 1 dependencies to satisfy what we need
- In the case of circuit breaker, if there is a troubled dependency, send traffic to dependency B
- Dependency B goes down too due to unable to handle such traffic and caused more problems to services using dependency B
- **Mitigation**: Focus on fixing dependency A and cut traffic from dependency B
- **Lesson**: do not introduce a back-up dependency unless acting as a lever for high velocity events

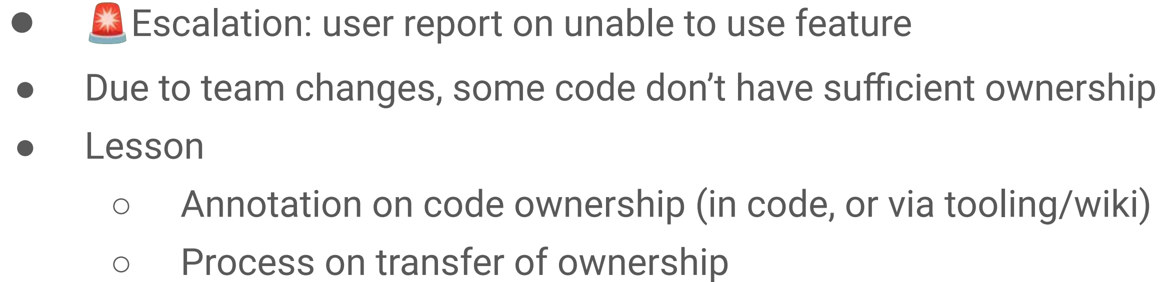


Bad commit

- 🚨100% server availability drop
- A bad commit that can go out to production is not the fault of the commit owner, but rather lack of proper process to prevent it from happening
- Lines of defense
  - Code review
  - Testing
    - Unit test
    - End to end test
    - Manual / QA test
  - Gating
  - Change management / risk assessment
- Lesson: No blame. Avoid making the same mistake. Focus on learning

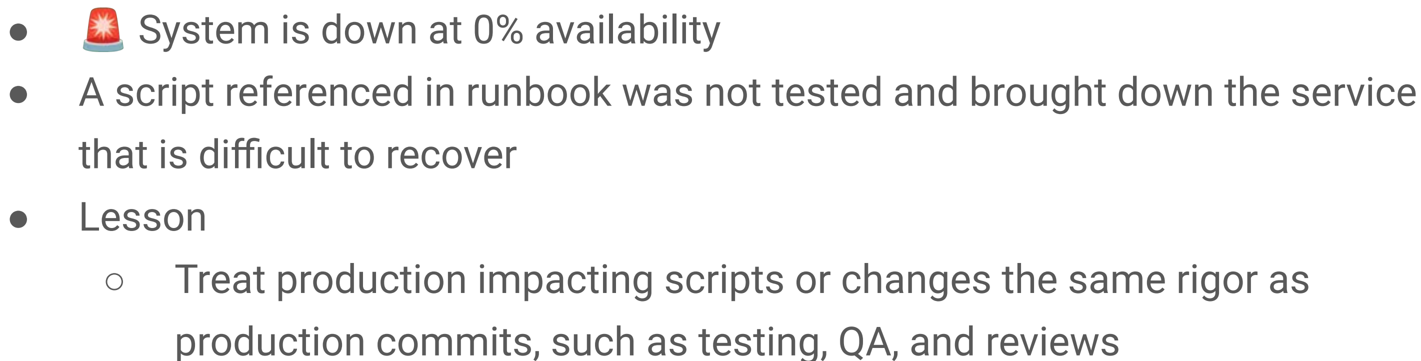


Lack sufficient ownership



- 🚨Escalation: user report on unable to use feature
- Due to team changes, some code don't have sufficient ownership
- Lesson
  - Annotation on code ownership (in code, or via tooling/wiki)
  - Process on transfer of ownership

Bad scripts



Prevention

Defensive coding practices

- Gate your feature
- Log any exceptions or corner cases and include critical information
- Do not overly rely on backup option. Fail fast
- Graceful error handling
- Ensuring null safe
- Set a timeout
- Retry

Detection

- Alert
  - Have an operational goal in mind when setting alerts
  - Set reasonable threshold based on operational goal
  - Find the balance to avoid noisy alert and missed alert
  - Use the historical data inform alert threshold
  - Alerts need to have a runbook to cover common steps on what to do and lessons from past scenarios
- Dashboard
  - Inform people on the general trend, such as seasonal patterns
  - Can help debug and find correlation
  - Have a clear name, time period, and useful references/marker (alert criteria, wow, etc)
- System limitation
  - Max you can support
  - Load testing
  - Chaos testing

Mitigation

- Mitigation is like saving life in the emergency room. Time is the essence
- Follow the runbook
- Root cause the issue following alerts and dashboards
  - Is it happening at a specific time?
  - Does it line up with a specific commit, app version, or gating changes?
  - Is there any exception logs or metric to pinpoint specific files
  - Are there related issues identified by other teams at the same time? (company-wide events)

Prepared for high velocity events

- High velocity events are dangerous time, such as an anticipated large increase of traffic, a product launch
- Have a risk mitigation plan based on factors, such as the traffic estimation
- Set up dashboards and alerts on critical metrics to monitor
- Have levers for the worst case scenarios
- Audits on your service, e.g. re-run load test, chaos test, etc

Sev review

- Sev review is intended to be a learning experience. Blameless
- Timeline of the events
- Root cause
  - 5 why
- Retrospective on how to improve
  - Detection time
  - Time to root cause and mitigate sev
- Action items to prevent it from happening in the future
- Hold accountable (timeline, tracked)
- Culture of continuous improvement

Thank you