# Team Based Performance Engineering

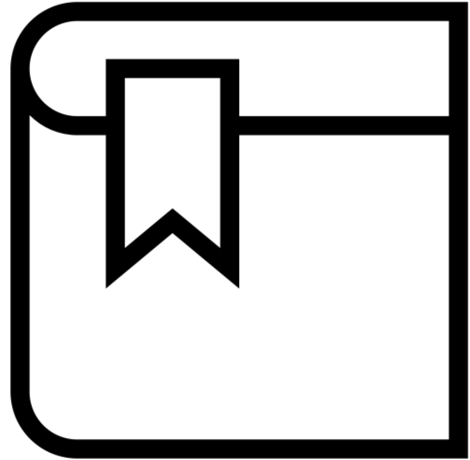
# What you'll learn

- **Designing** for performance
- **Adding** metrics and telemetry
- **Enabling** your performance engineering effort early
- **Making** your tests and environments stable and repeatable

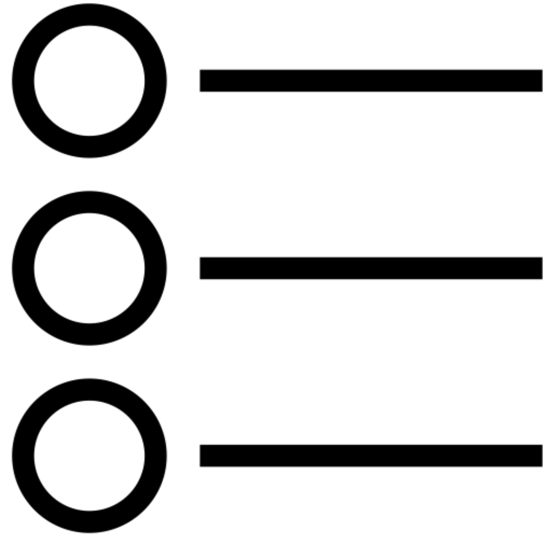
# More specifically

- **Examine** system designs and architecture for opportunities and bottlenecks
- **Use** tooling such as Artillery and Locust to start building early load tests
- **Add** Prometheus instrumentation to your application
- **Consider** how pipelines augment continuous performance testing

# But who are we?



# Team Test for Performance Engineering



# Answer these 10 questions

# Team Test for Performance Engineering

- Do you have a dedicated environment for performance testing?
- Does your code review checklist reference performance related concerns?
- Have you chosen a load testing tool?
- Is performance testing included in your definition of done?
- Are performance requirements (NFRs) for your system known and available to the team?

# Team Test for Performance Engineering

- Can you set test data to a known state in your test environments?
- Have you mocked your third party dependencies?
- Are performance improvements celebrated within your organisation?
- Does your continuous integration build take less than ten minutes?
- Can you check how your applications performance metrics are trending?

# What does it all mean?

# Fundamentals and Principles

# What is performance testing?

**Performance testing** is a technique that measures the **speed, response time, scalability, stability, and responsiveness** of software under **varying workloads**.

# What is performance tuning?

**Performance tuning** is remedial work **after** identifying performance issues. Detecting the **exact problems** that the system has, getting to their **root cause**, and fixing these **bottlenecks**.

# Problems

- Performance testing can take a lot of effort to enable
- Performance problems can be hard to diagnose and fix
- Performance tuning means more performance testing
- And repeat

# What is performance engineering?

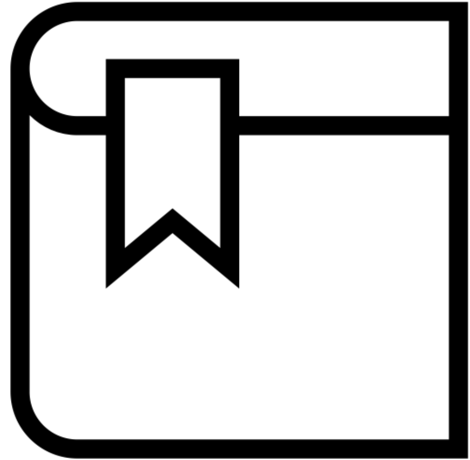
**Performance engineering** is a **cultural** shift towards integrating the **performance mindset** into **every facet** of your software development life cycle.

# This might include

- Designing architecture for performance
- Integrating performance into user stories
- Early performance testing
- Calculating future workloads
- Monitoring performance
- Chaos engineering
- Better UX and UI design
- Understanding your business better

# What about site reliability engineering?

Site reliability engineering (SRE) is the practice of **applying software engineering principles to operations** and infrastructure processes to **help organizations** create highly reliable and scalable software systems.



# A Career Retrospective

# Early days

- Performance testing was done by another company.
- It took a long time to setup and run.
- The testing tool and not the system generated the metrics.
- It was done by someone else, at the end of the project and nothing ever got fixed.

# Growing up fast

- Worked on a 're-platforming' project.
- The performance requirements were 'not worse than the current system' please.
- (As we couldn't tell clients what we were doing)
- The team owned the environment, except for the database.
- We built a performance environment first and ran regular tests

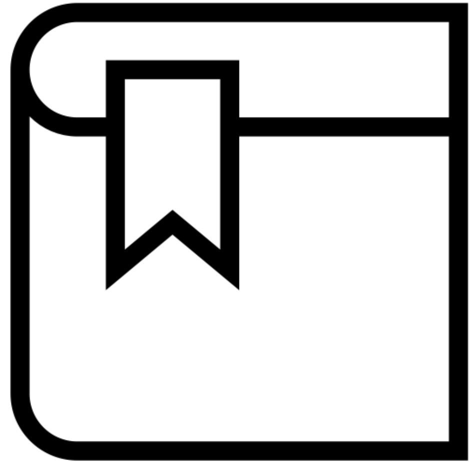
# Regression in testing

- Used production for performance testing.
- Created a Site Reliability function.
- Who ended up coming in early morning to run performance tests.
- Still not the remit of the team who was building the software.
- Performance problems seen as problems for technical people to solve.

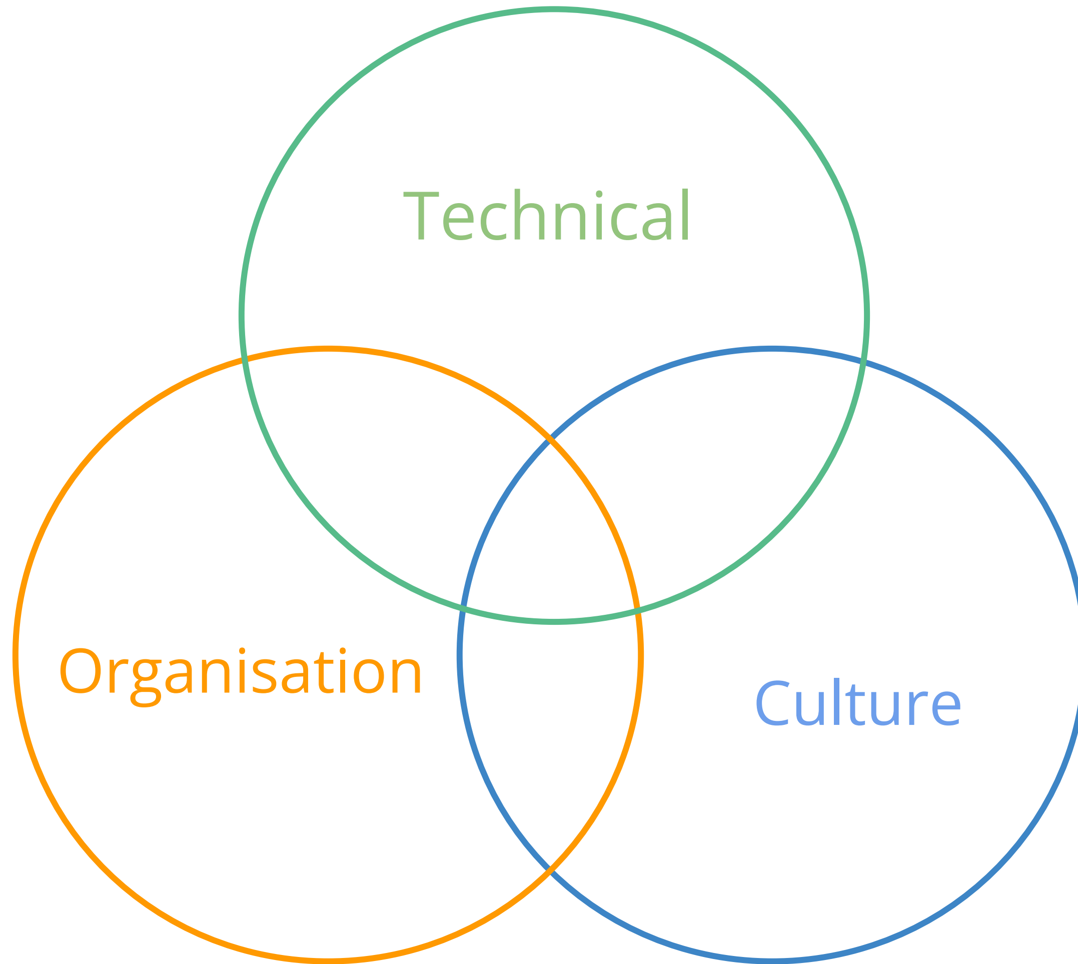
# My later years

- Big Brexit sized problem with unknown loads on a customs system.
- Had to gather metrics to understand the workload.
- As a team, taking ownership of performance engineering.
- Designing specifically for performance, not attempting to retrofit

# What was your performance engineering journey ?



# Three Foci of Performance Engineering


Technical
Organisation
Culture

# Technical

- Tools
- Instrumentation
- Build automation
- Performance in tests
- Information radiators

# Organisation

- Teams accountable for performance
- Performance in organisation design - enabling teams for performance
- Service Level Objectives - response time and availability

# Culture

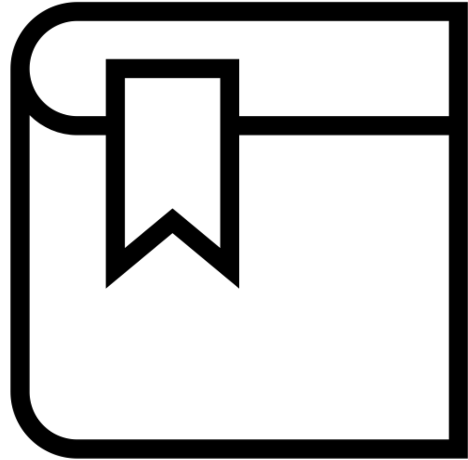
- Place value on performance
- Compensation and performance
- What happens when a performance incident occurs?

Let's look again at the Team Test results.

# Designing for Performance

# In this exercise

- We will focus on designing architecture for performance
- We will also cover:
  - Code design patterns for performance
  - Organisations design patterns for performance



# Architecture Patterns

# A Brexit Tale

- Volume was the great unknown, no one had to think about it for years either!
- Performance, capacity and scale were first class concerns
- As were reliability and time to recovery
- So we designed with these quality aspects in mind...

# What design patterns can we use for performance engineering?

# Patterns we applied to our design

- Eventual consistency
    - Rules of the system - first writer wins
    - Not all data sources are up to date
- Distributed systems
    - Splits responsibilities
    - Deployable as individual systems
- Caching
    - Consider data that is accessed often but changes less frequently

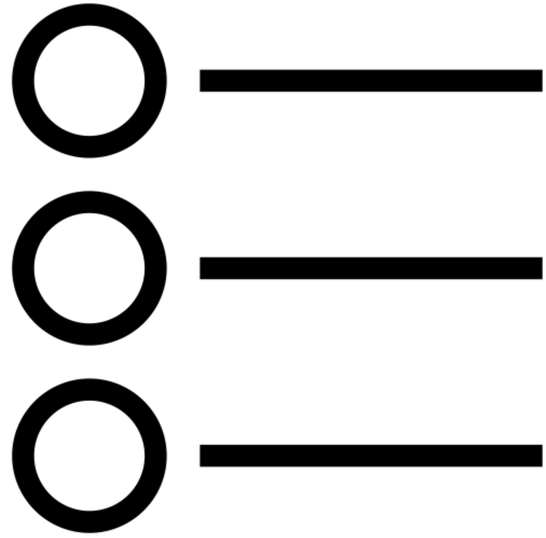
# Patterns we applied to our design

- Circuit Breakers
  - Based on load or errors, protects the service
  - Can have a detrimental effect on performance
- Load Balancing
  - Dealing with peaks of traffic and distributing load
  - Direct traffic away from servers being deployed to
- Command Query Responsibility Segregation
  - Two models for reading and writing data

# Patterns we applied to our design

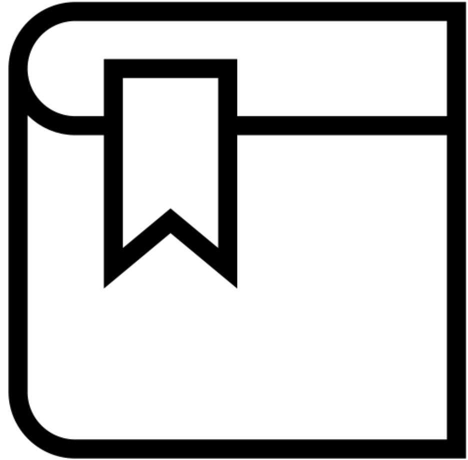
- Separation of concerns
  - One piece should deal with only one responsibility
  - By volume, action or by layer
- Configuration as code
  - Performance related settings with other config
  - Environment specific so you know the difference
- Infrastructure as code
  - Built from code in source control for repeatability
  - Deploy and auto scale on demand

# What did our architecture evolve into?



# Individual -Draw Your Architecture

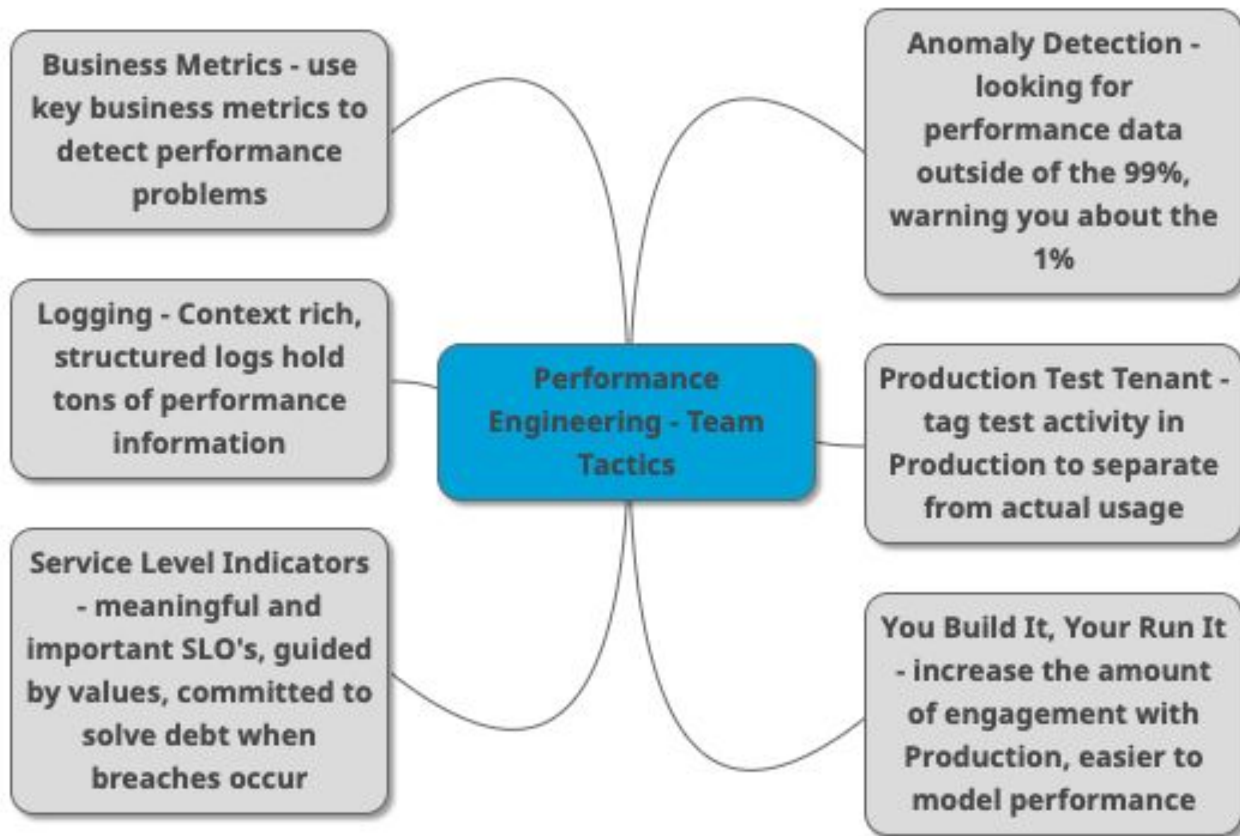
# Groups - Apply the patterns to your architecture

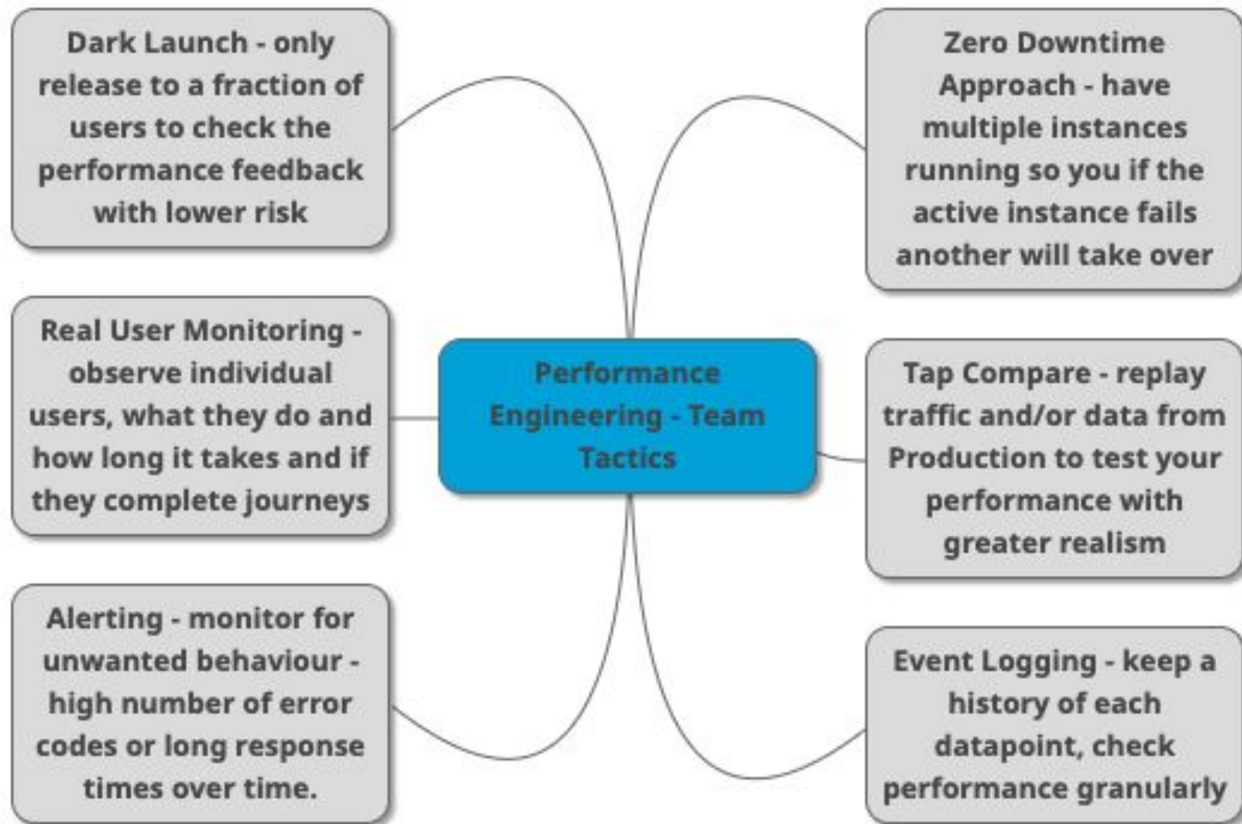


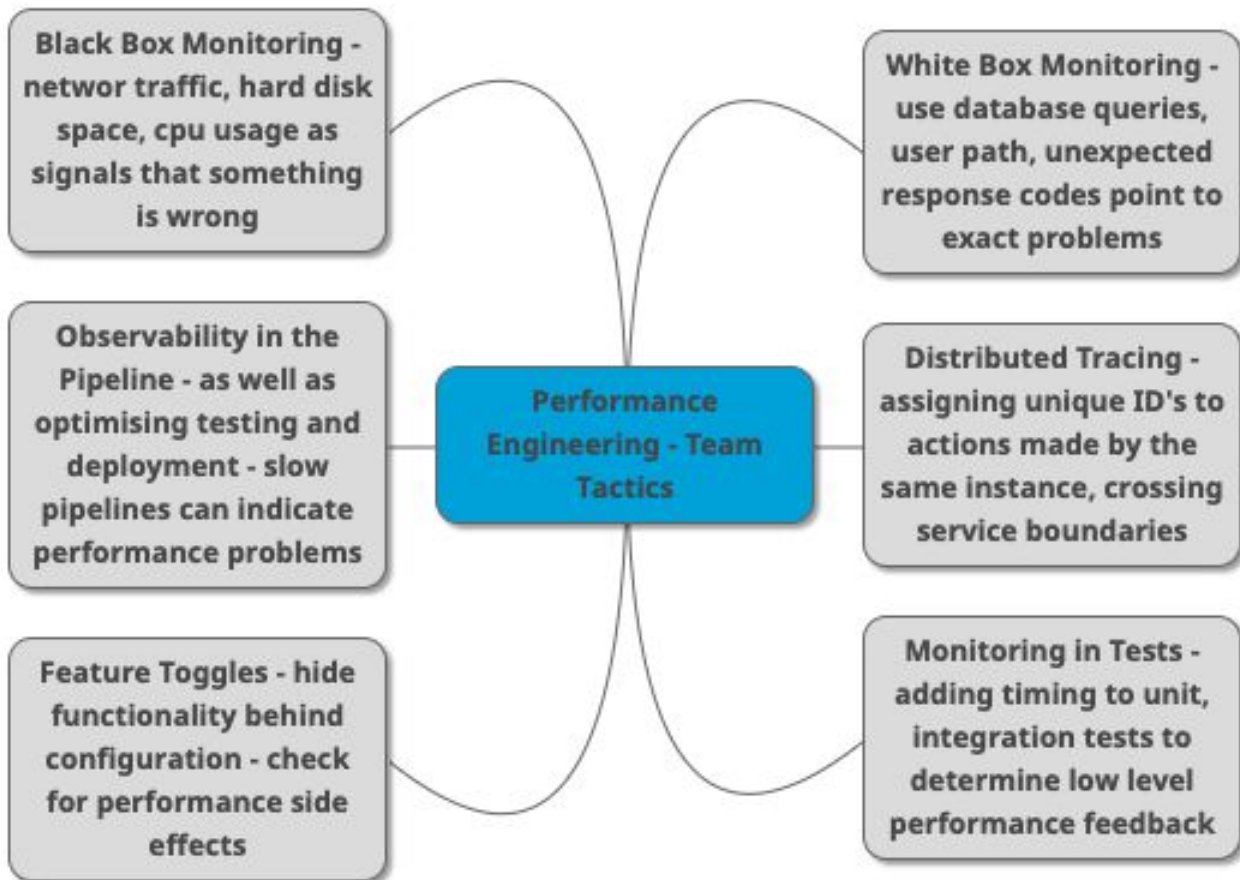
# Team Tactics

# Tactics and patterns

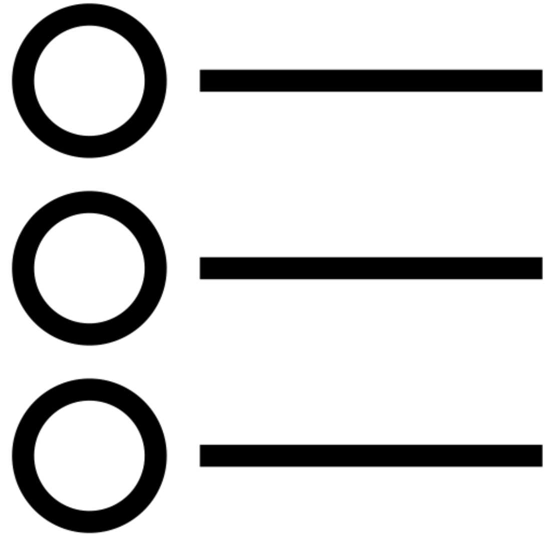
- **Tactics** influence the **patterns** you want to achieve in your design
- Have a **range** of tactics at your disposal
- Don’t be afraid to **change** the tactics used as you evolve
- Evolving towards a design is a **team** concern, find tactics that increase buy in


Business Metrics - use key business metrics to detect performance problems
Logging - Context rich, structured logs hold tons of performance information
Service Level Indicators - meaningful and important SLO's, guided by values, committed to solve debt when breaches occur
Performance Engineering - Team Tactics
Anomaly Detection - looking for performance data outside of the 99%, warning you about the 1%
Production Test Tenant - tag test activity in Production to separate from actual usage
You Build It, Your Run It - increase the amount of engagement with Production, easier to model performance

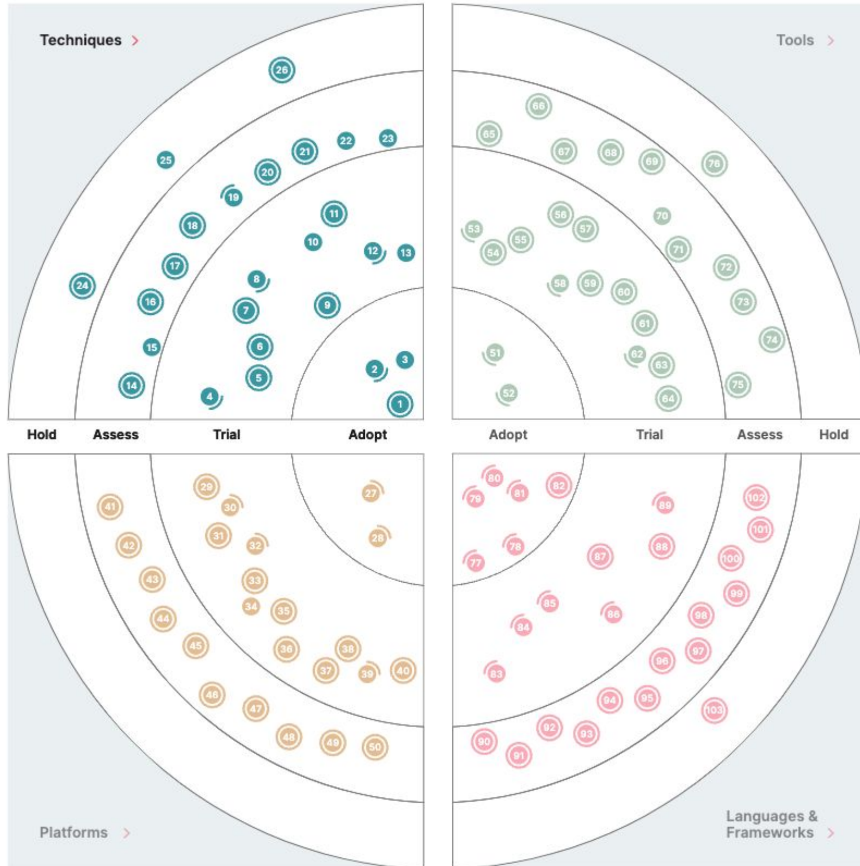

Performance Engineering - Team Tactics
Dark Launch - only release to a fraction of users to check the performance feedback with lower risk
Real User Monitoring - observe individual users, what they do and how long it takes and if they complete journeys
Alerting - monitor for unwanted behaviour - high number of error codes or long response times over time.
Zero Downtime Approach - have multiple instances running so you if the active instance fails another will take over
Tap Compare - replay traffic and/or data from Production to test your performance with greater realism
Event Logging - keep a history of each datapoint, check performance granularly


Black Box Monitoring - networ traffic, hard disk space, cpu usage as signals that something is wrong
White Box Monitoring - use database queries, user path, unexpected response codes point to exact problems
Observability in the Pipeline - as well as optimising testing and deployment - slow pipelines can indicate performance problems
Performance Engineering - Team Tactics
Distributed Tracing - assigning unique ID's to actions made by the same instance, crossing service boundaries
Feature Toggles - hide functionality behind configuration - check for performance side effects
Monitoring in Tests - adding timing to unit, integration tests to determine low level performance feedback

# What other tactics have you used?



# Choose 5 tactics for your team




# Follow the radar

# Early Performance Engineering

# Always too late...

- Testing a new websocket push messaging server
- Needed to work at scale for the next Football World Cup
- Quality was poor, so the last thing we did was performance testing
- No time to fix issues, business had to scale back their plans

# Is this a familiar story?

# What is a early performance engineering?

Any **activity** that can help to improve performance that can be done involving the **team**, close to when the **code** was written, within an **iteration** that has meaning for the team.

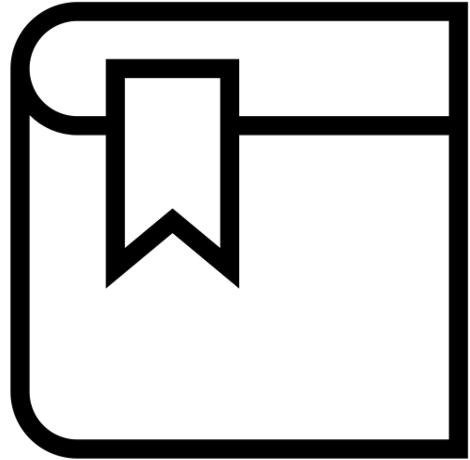
# Potential benefits

- Talk about expectations early
- Detect problems early
- Actual time to resolve performance problems
- Less team context switching
- Continuous focus on performance
- Operations like you more, a lot more
- Think about test data needs
- Experiment with configurations
- Build useful tests that can be scaled later

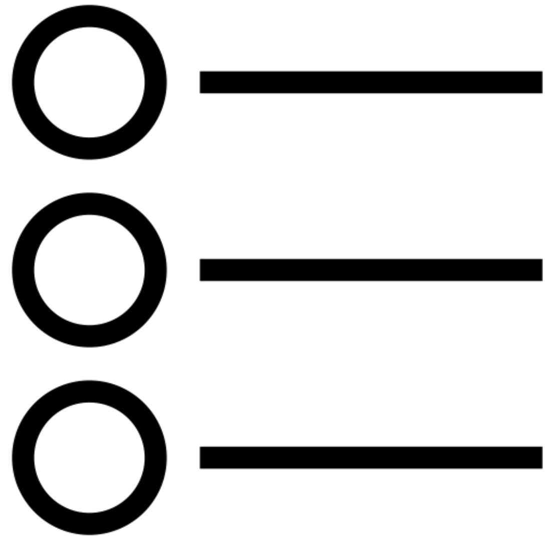
# Potential activities

- Focus on operational features
- Code reviews with performance focus
- Monitoring pipeline run times
- Adding timers into tests
- Working on technical debt
- Exploratory local performance testing!

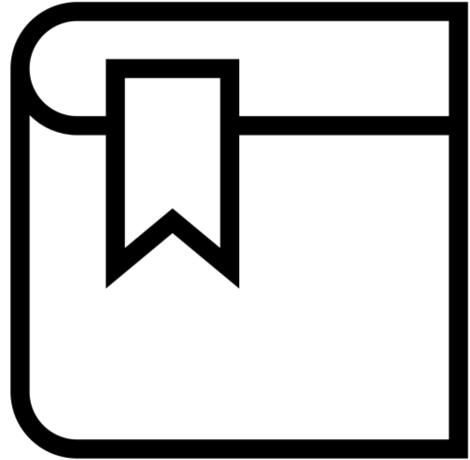
# Any challenges to this approach?



# Getting up and running



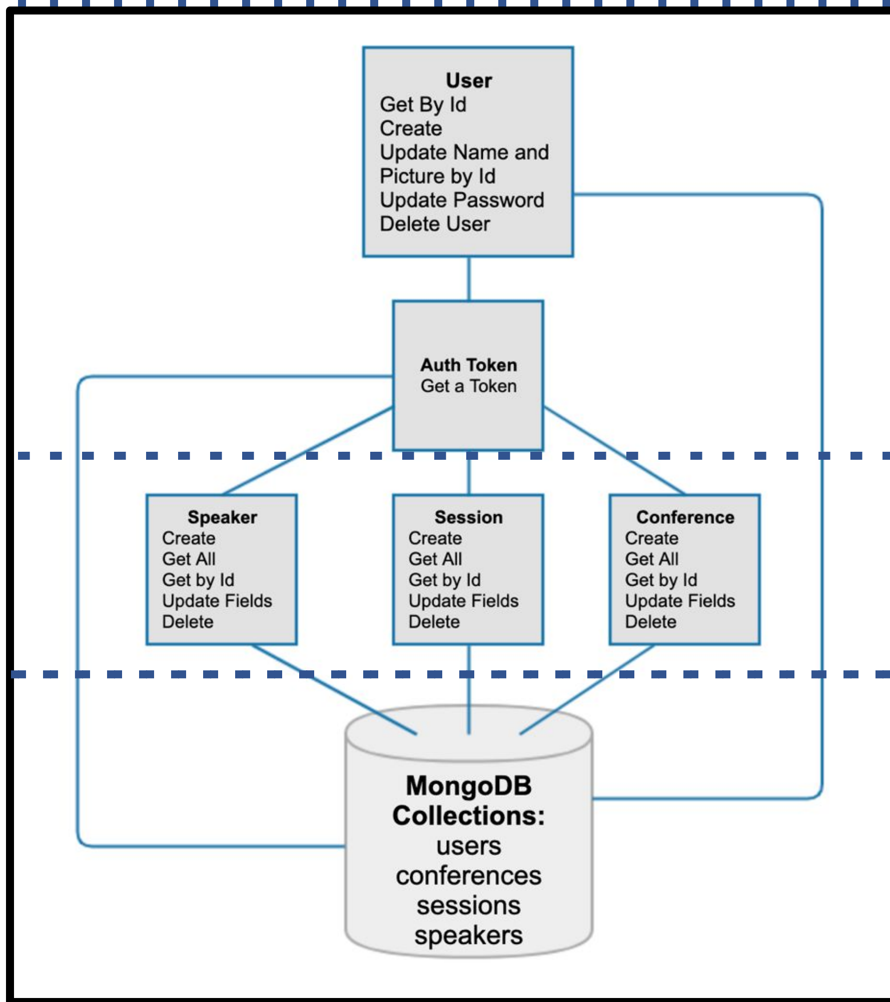
# Follow along: Install and run application from source control



# Understand the application

# Application purpose

- Allow organisers to showcase their events on a new site and app
- We are working on the back end servers and data storage
- First testing conferences, then the world!

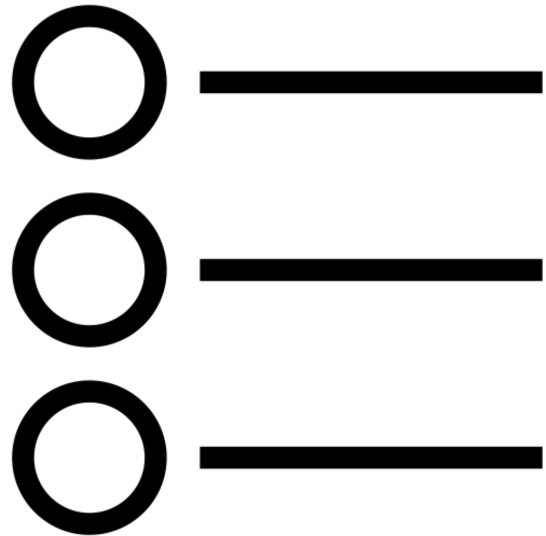



Create a user or admin with Master Token

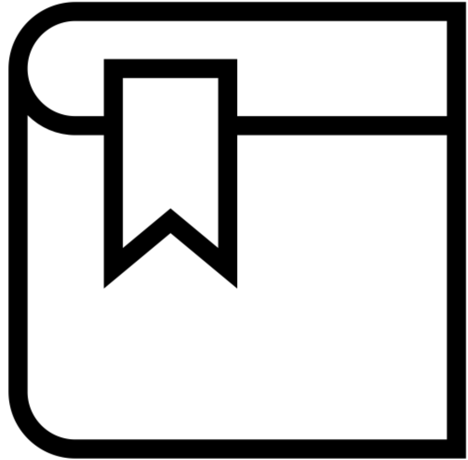
Username and password to get an auth token

Use the token to invoke endpoints

Use that auth token to invoke other endpoints, some are admin



# Follow along: Explore the app with Postman



# Artillery

# Running your first test

- Target
- Phases
- Variables
- Environment Variables
- Before
- Scenarios
- Capture
- Weight
- Expect

```
ashwinter@Morpheuss-MBP.lan /Users/ashwinter/code/conferencesApp/artillery [atd2022-app]
% artillery run --dotenv ../.env first_test_scenario.yaml
```

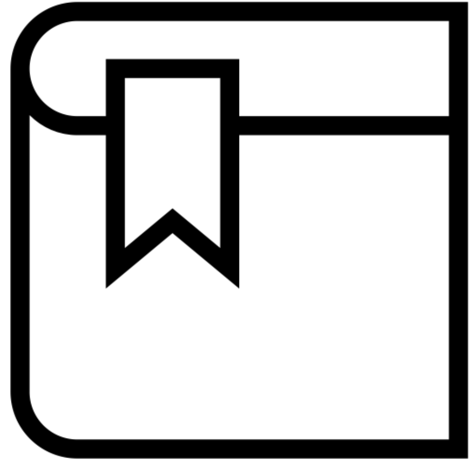
# How can we improve this test?

# Debugging in Artillery

```
ashwinter@Morpheuss-MBP.lan /Users/ashwinter/code/conferencesApp/artillery [atd2022-app]
% artillery run --dotenv ../.env challenge_user_by_token.yaml
```

```
ashwinter@Morpheuss-MBP.lan /Users/ashwinter/code/conferencesApp/artillery [atd2022-app]
% DEBUG=* artillery run --dotenv ../.env challenge_user_by_token.yaml
```

- Pick tools with debug powers
- Commit often, get review
- All tools have limitations



# Artillery Challenge

# Artillery challenge

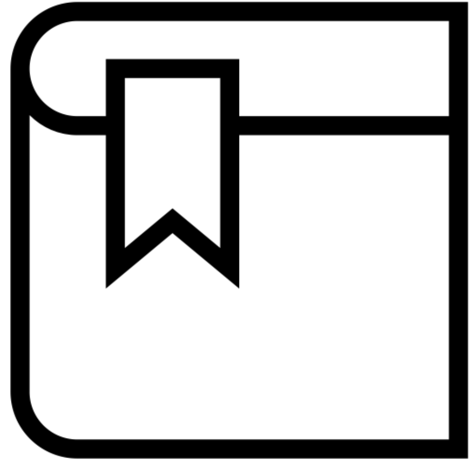
- Create a new test
- Create and authenticate a new user
- Add a new conference
- Capture the id
- Get the conference by ID
- (Advanced) Create a data generator - hint payload or processor

Choosing and using
performance tooling

# Choices, choices

- Testing an application used for credit referencing.
- API requests sent by the clients needed to persist an audit and a billing record.
- The billing record was not being persisted.
- SQL Exception in the logs but not visible in the application.

# What factors influence tool choice?



# Choosing performance tooling

# 10 point plan

- Can the developers get involved?
- Library or tool or both?
- Integrates with pipeline?
- Load injector compatibility?
- Maintenance and update plan?

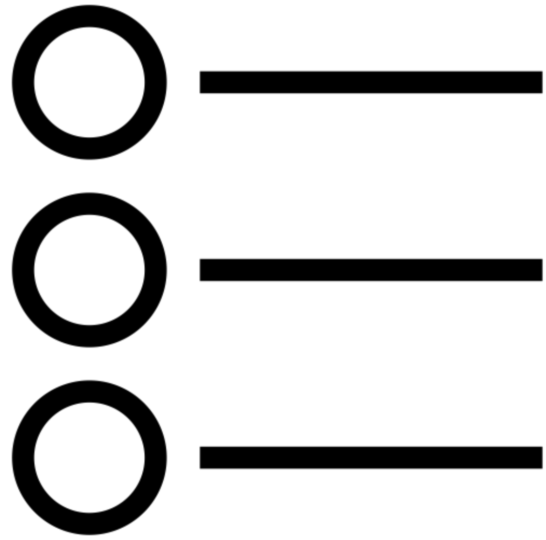
# 10 point plan

- Test data capabilities?
- Integrates with metrics tooling?
- Operations sponsored?
- Cloud compatibility?
- Early testing compatibility?

# Anyone stuck with an pre-existing (and/or paid) tool?

# Open source vs Proprietary

- I am extremely biased to open source, I should be more open minded!
- Proprietary tools can be useful but...
  - Can be too general to cater for larger market.
  - They can shape you, rather than you shaping them.



# Follow along: Working with Locust

# Running our first test

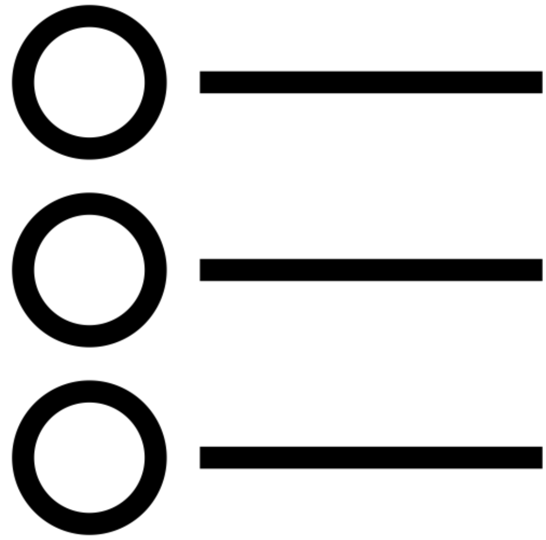
- Running pip install (use recognised strictures for ease of installation)
- Create a user
- Test data generator
- Simple task
- Catch the response and assert
- Run and use the Locust UI

```
ashwinter@Morpheuss-MBP.lan /Users/ashwinter/code/locustExample [main]
% locust -f locustfiles/conferencecreateuser.py -u 10 -r 1 -t 10s --host http://localhost:9000
```

# What better patterns could we implement?

# Better patterns

- Add test start and stop events
- Add on_start and on_stop for setup and teardown for tasks
- Better logging
- Making it fail with better information



# Follow along: Implement the patterns

# Locust challenge

- Create a new test
- Create and authenticate a new user
- Add a new speaker
- Capture the id
- Get the speaker by ID
- (Advanced) Delete the speaker - tip you will need an admin user

# Advanced tips

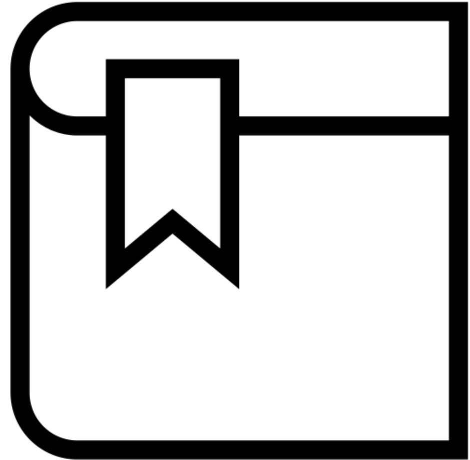
- Separate data generation from tests
- Same for loading specific data sets
- Be able to set your application to a known state
- Prioritise ability to run continuously over complex tests

# Instrumenting Our App

# A story about not collecting performance metrics

- Big replatform from one set of technologies to another.
- Performance testing was pushed to the end every time.
- The new tech and system had not been performant all along.
- Collecting metrics early on and continuously would have really helped.

# Earliest point in development for performance metrics?



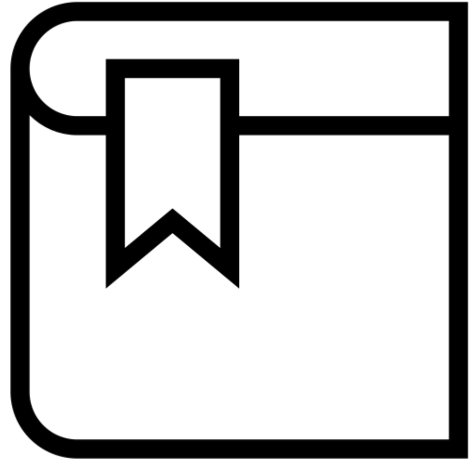
# Benefits of instrumentation

# What is instrumentation of code with metrics?

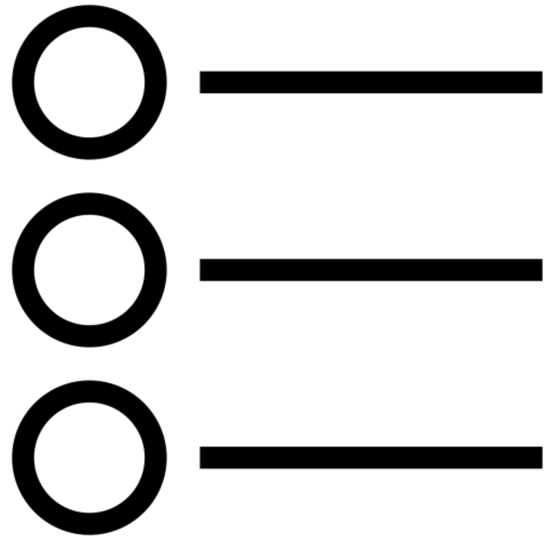
- Adding code to your application to gather information.
- Response time and request and response sizes are key examples.
- Can be scary. Often there is resistance to “polluting” code with operability concerns.

# Benefits to testing and operability

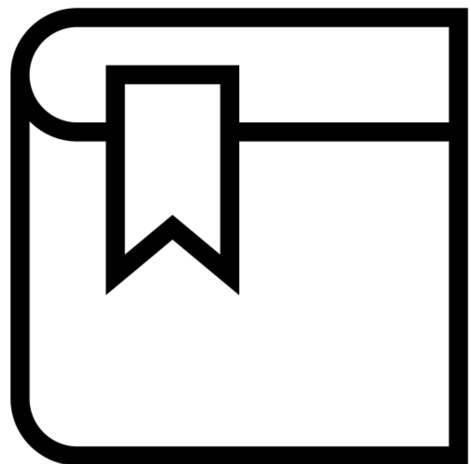
- Gather information while you test.
- Broaden your test approach early.
- Validate your test approach and focus over time.
- Information useful and common in both development and operations.



# Adding an instrumentation library



# Follow along: Prometheus api metrics library

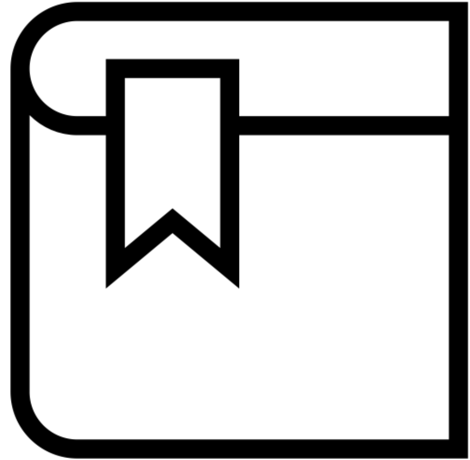


# Prometheus metrics

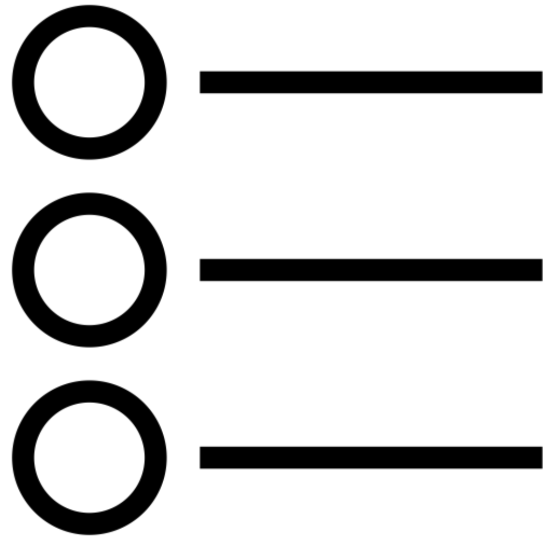
# Metrics available in Prometheus

| Metric Type | Description | Testing Significance |
| --- | --- | --- |
| Counter | Cumulative count, like total http requests to an endpoint | Focus on areas of the application with highest traffic. |
| Gauge | Value that can go up or down, like CPU | See how diagnostic information changes as you test. |
| Histogram | Samples response duration into buckets | Spotting outliers in a measurement during testing |
| Summary | Similar to histogram but provides sum and count | Rolling averages to check for deviations |

# How might you use these metrics as you test?



# Exposing configured metrics



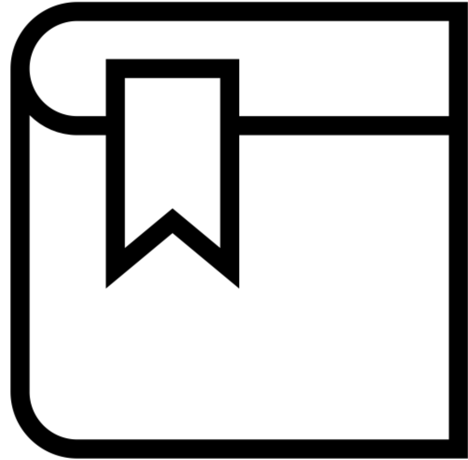
# Follow along: /metrics endpoint

# Querying Instrumented Data

# A story about lots of data

- Our applications generate a lot of data, pinpointing what matters is important.
- Post instrumentation our code for testing, we need a means of querying that data.
- Access to a New Relic instance monitoring both client and server side metrics.
- Turning that access to data into insight was much harder, new skills are required.

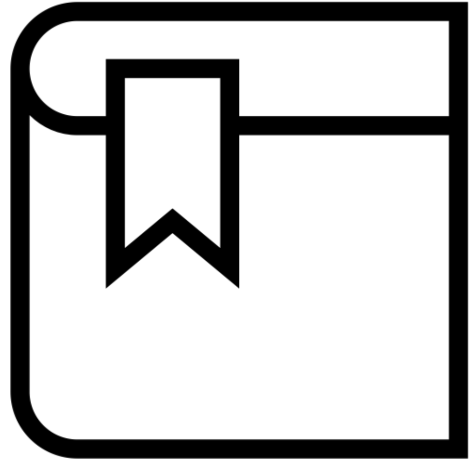
# What access do you have to data from logging and monitoring ?



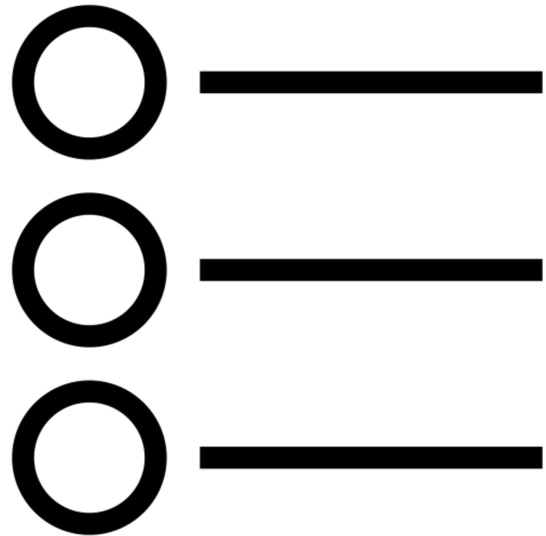
# Time Series Databases

# What is a time series database?

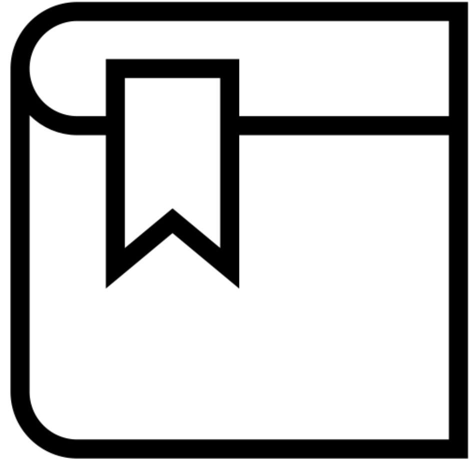
- New data is almost always a new entry
- Typically stored in time order
- Indexed by time, rather than an identifier
- Sampled and aggregated for long term analysis



# Configuring Prometheus



# Follow along: Listening to our application



# Prometheus Query Language

# Using labels to filter metrics

- These are like clauses in your SQL queries
- Key value pairs within the Prometheus time series database

```
_api_request_duration_seconds_count{status="500",job="demo"}
```

# Using operators to combine metrics

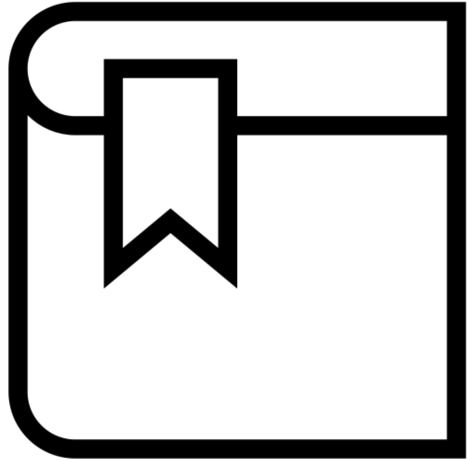
- Arithmetic
- Logic
- Comparison
- Aggregation

```
histogram_quantile(0.9, rate(demo_api_request_duration_seconds_bucket{job="demo"}[5m])) > 0.05
  and
rate(demo_api_request_duration_seconds_count{job="demo"}[5m]) > 1
```

# Using time offsets to target metrics

- Time - in the last 5 minutes
- Offset - relative to 1 hour ago

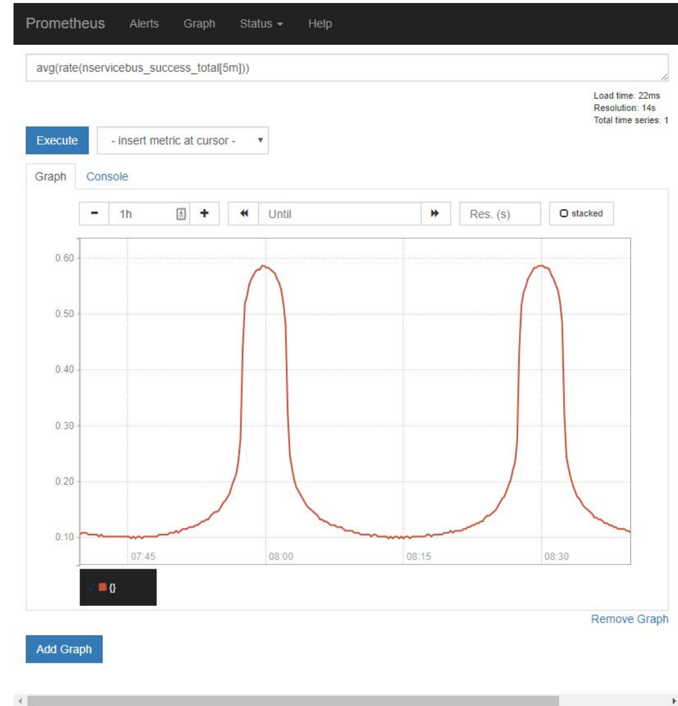
```
(api_http_requests_total{status=500}[5m] offset 1h)
```



# Using PromQL

# The power of queries in Prometheus

- Data analysis
- Visualisation
- Basis for alerts
- Modelling for performance



# Follow along: Performing queries on our metrics

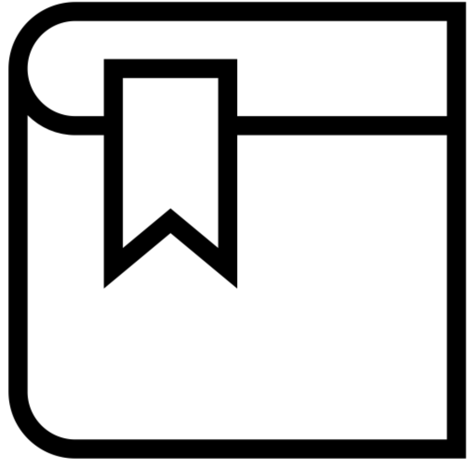
What metrics of your application will change most over time?

# Alerting

# Slower and slower…

- Test environments got slower and slower over time.
- Eventually ran out of memory and collapsed.
- Batch jobs were continuously failing and retrying.
- No alerts in the test environments…

# Ever seen a problem coming?



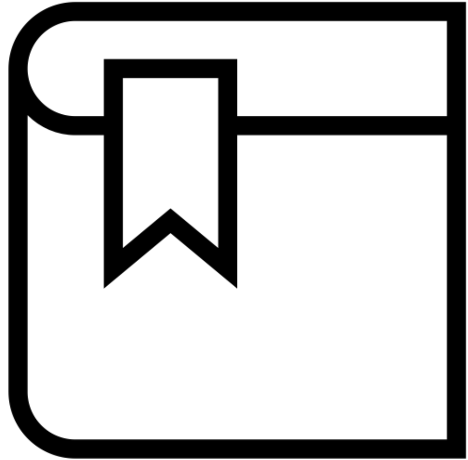
# Effective Alerts

# Characteristics of effective alerts

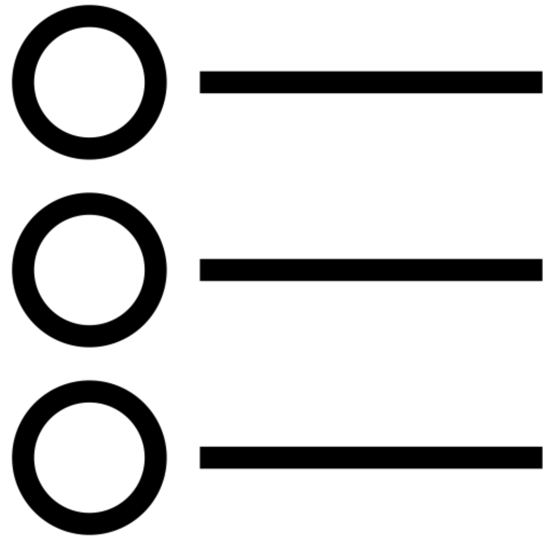
- Quality over quantity
- Actionable within your own application
- For dependencies, either actionable or informational
- Provide context as part of the bigger picture

# Questions for testers to ask

- How do we know our application is ready?
- How will we test our alerting?
- Are we open for business?
- How are we doing against normal?

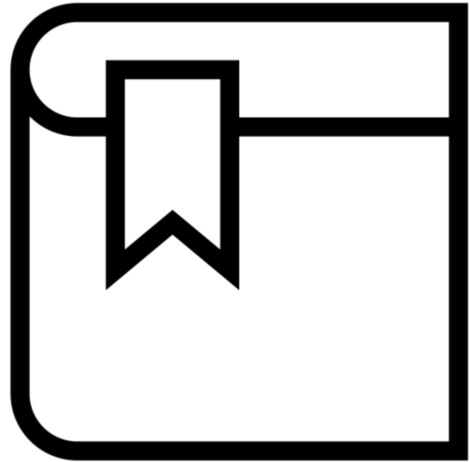


# Simple Alerts



# Follow along: Build a Simple Alert

# Designing tests and alerts



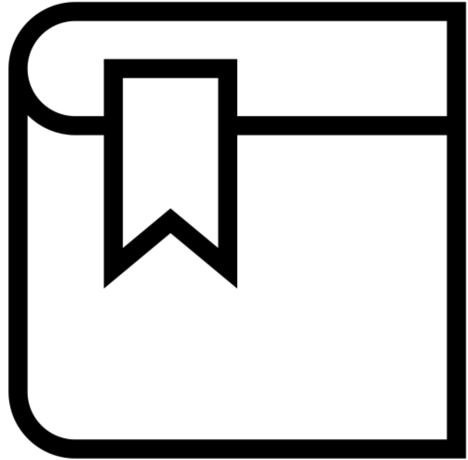
# Symptom and Cause Based Alerts

# Symptom Based Alerts

- Users having problems
- Correctness
- Latency over time
- Freshness - time since last use
- Expected features in use

# Cause Based Alerts

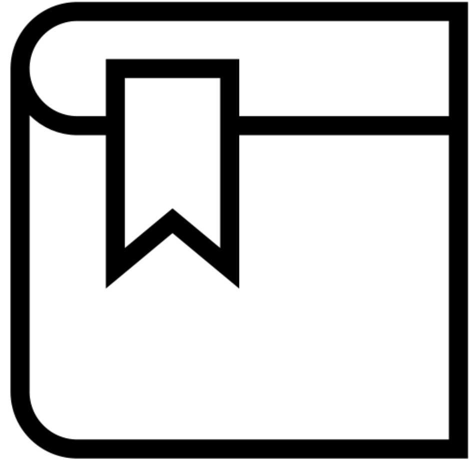
- System diagnostic alerts
- Deployments or infrastructure changes
- Almosts - 90% disk space.
- Focus on catching the symptom first before the collapse



# Complex Alerts

# Follow along: Build Symptom and Cause Alerts

# Performance engineering in the pipeline



To be meaningful, performance engineering must be continuous

# When is important...

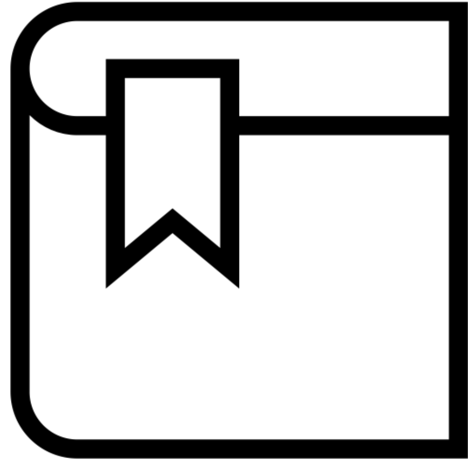
- Early - get your performance engineering information from CI and early environments
- Just in time - whenever an artifact is being pushed to environments, test then.
- Chronological job - run your performance tests on a periodic basis and gather trends over time.

# How and who is important...

- Performance tooling that can be run via the command line
- Code that has the same care as production code
- Teams can take ownership of the tests that run

# Don’t for what, how and who...

- Performance tooling that can be run via the command line
- Code that has the same care as production code
- Team can take ownership of the tests that run



# Back to our Brexit example...

As always, start with why. :)

# Thanks for your time