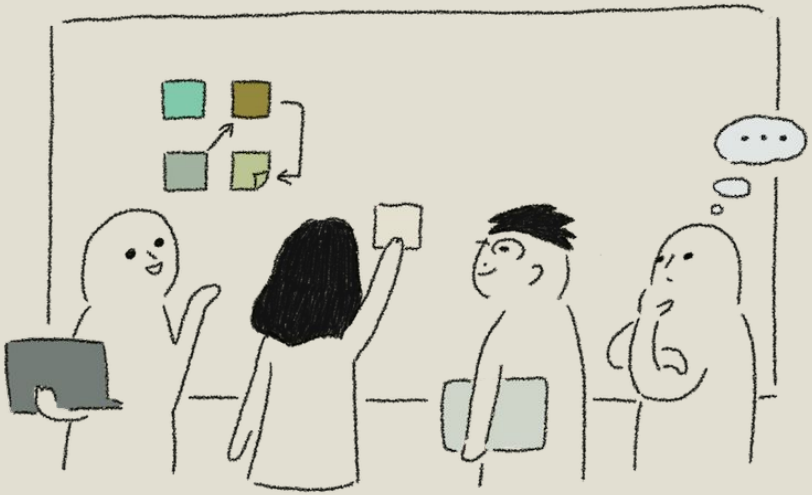
DATE 20250407

Session 6



backend

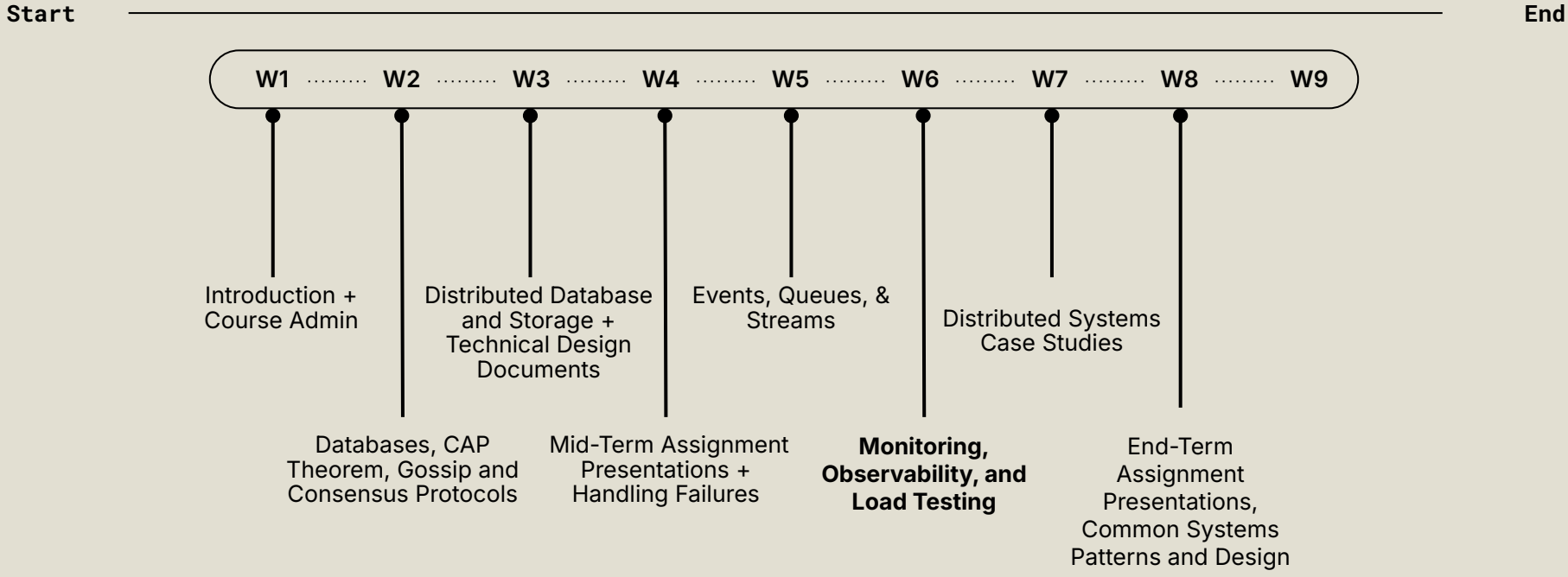
cohort #0 by open camp

#6's agenda

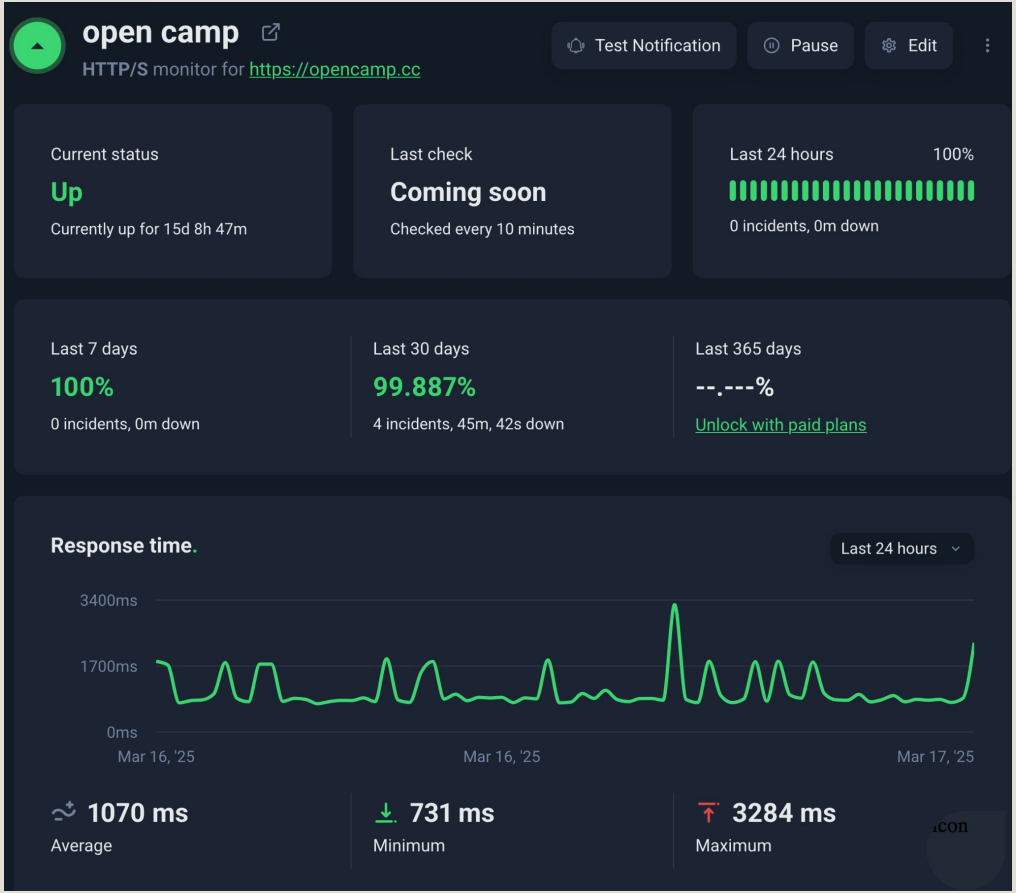
1. admin matters (if any)
2. monitoring, observability
3. load testing
4. w6 assignment

admin matters

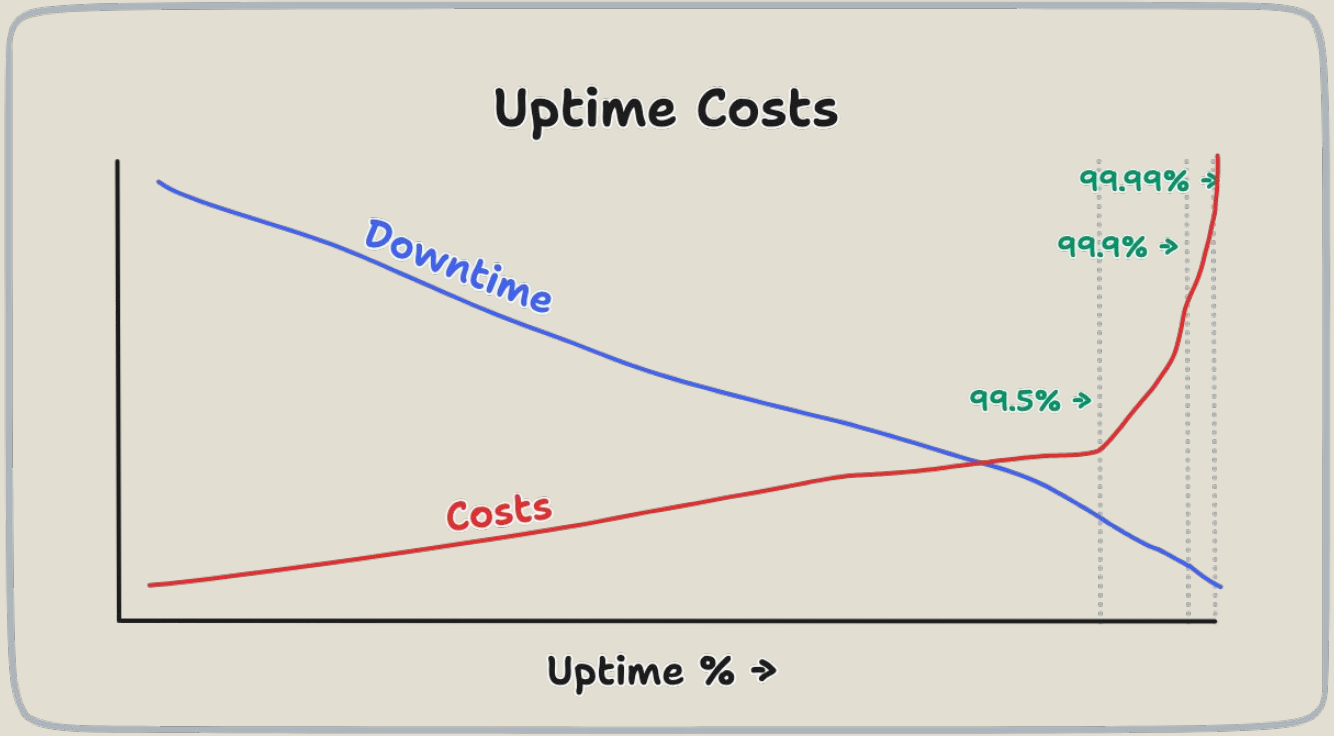
Curriculum: https://opencamp-cc.github.io/backend-curriculum/



6.1 monitoring, observability



Uptime Monitoring



Uptime Guarantees – A Pragmatic Perspective

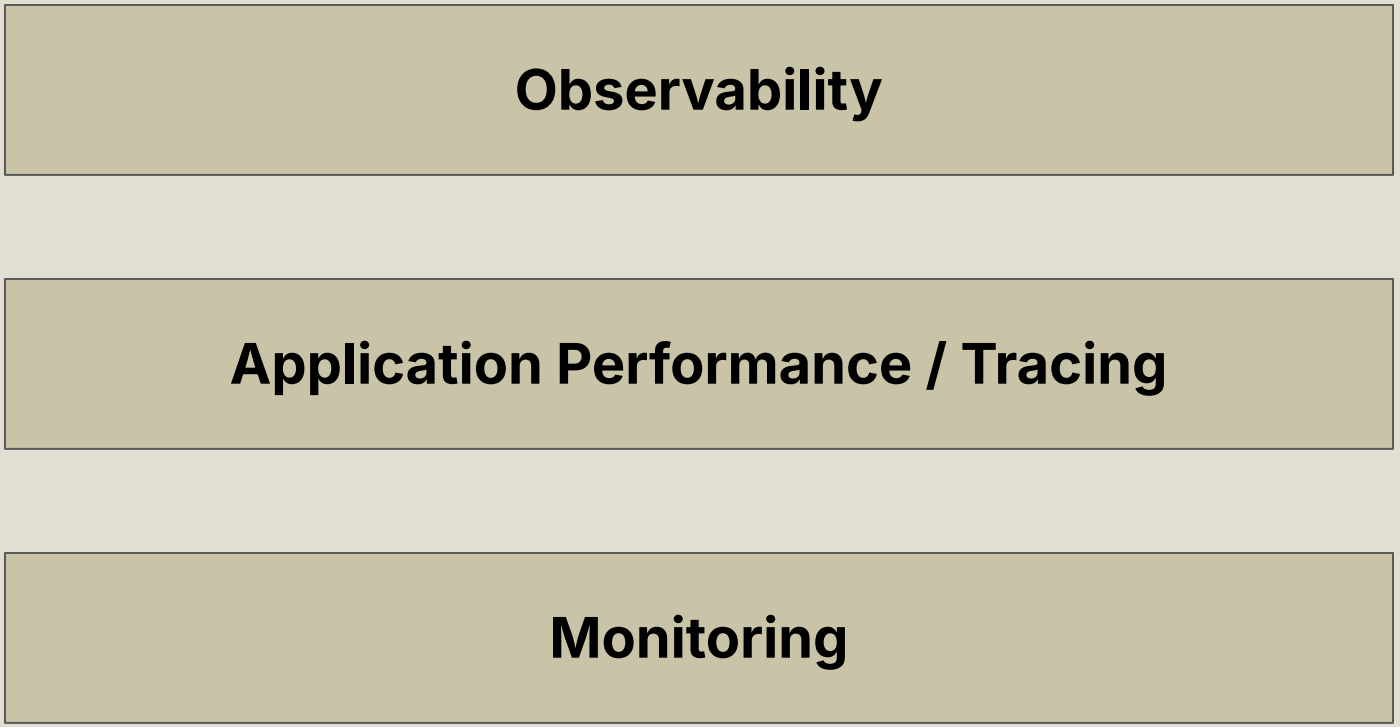
https://world.hey.com/itzy/uptime-guarantees-a-pragmatic-perspective-736d7ea4

Uptime Target: 99.99%, etc.

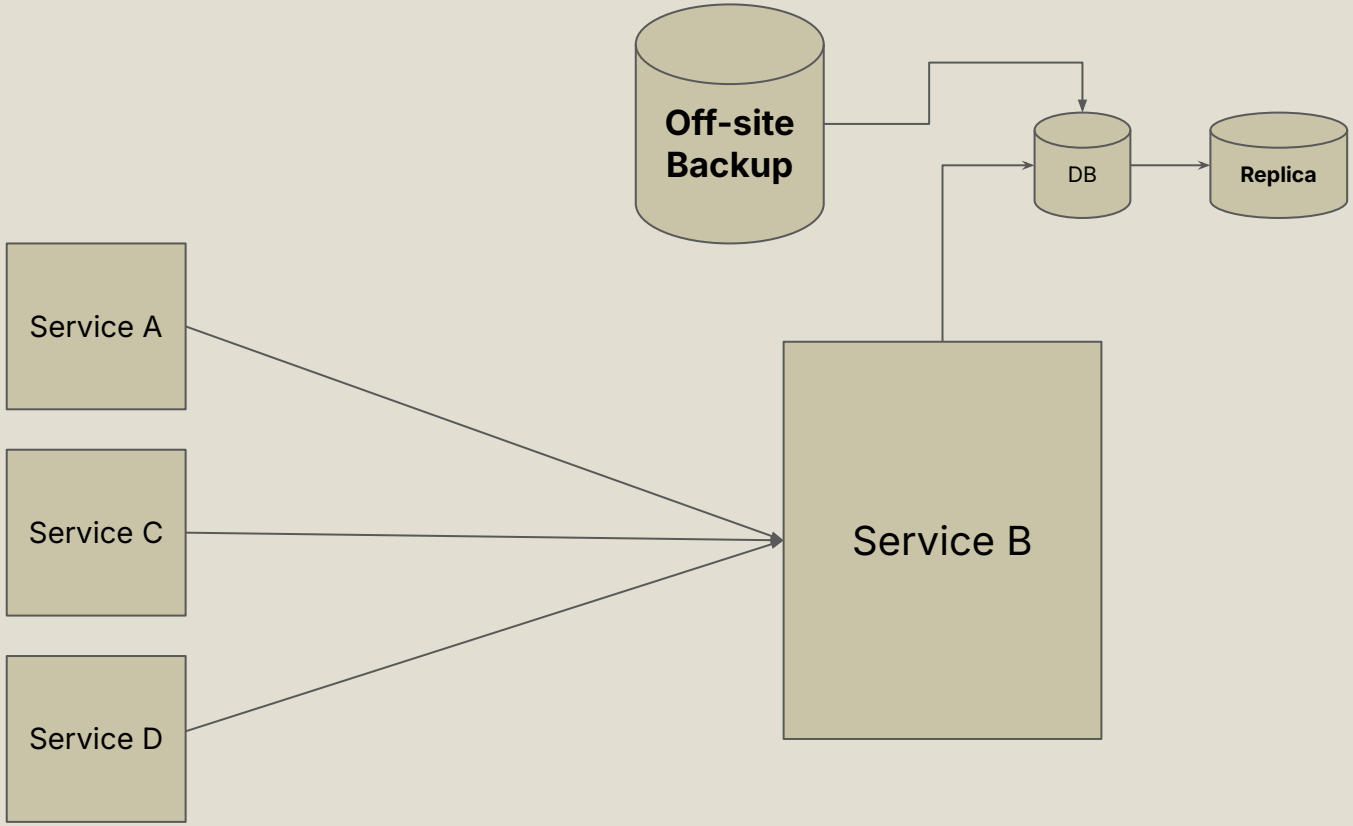
Recovery Point Objective (RPO): The maximum amount of data loss (measured by time) that an organization can tolerate.

Recovery Time Objective (RTO): The maximum length of time it should take to restore normal operations following an outage.

Three Key Metrics



Three “Tiers” of Observability



Example System: RTO and RPO for Service B

Recovery Point Objective (RPO): The maximum amount of data loss (measured by time) that an organization can tolerate.

Solution	Target RPO
	24 hrs
	3~5 minutes (or less)
	5 seconds (or less)

Recovery Point Objective (RPO)

Recovery Point Objective (RPO): The maximum amount of data loss (measured by time) that an organization can tolerate.

Solution	**Target RPO**
Daily Offsite Backup	24 hrs
Warm standby (Log Shipping)	3~5 minutes (or less)
Hot standby (Streaming Replication)	5 seconds (or less)

Recovery Point Objective (RPO)

Recovery Time Objective (RTO): The maximum length of time it should take to restore normal operations following an outage.

Solution	**Target RTO**
Daily Offsite Backup	
Warm standby (Log Shipping)	
Hot standby (Streaming Replication)	

Recovery Time Objective (RTO)

Recovery Time Objective (RTO): The maximum length of time it should take to restore normal operations following an outage.

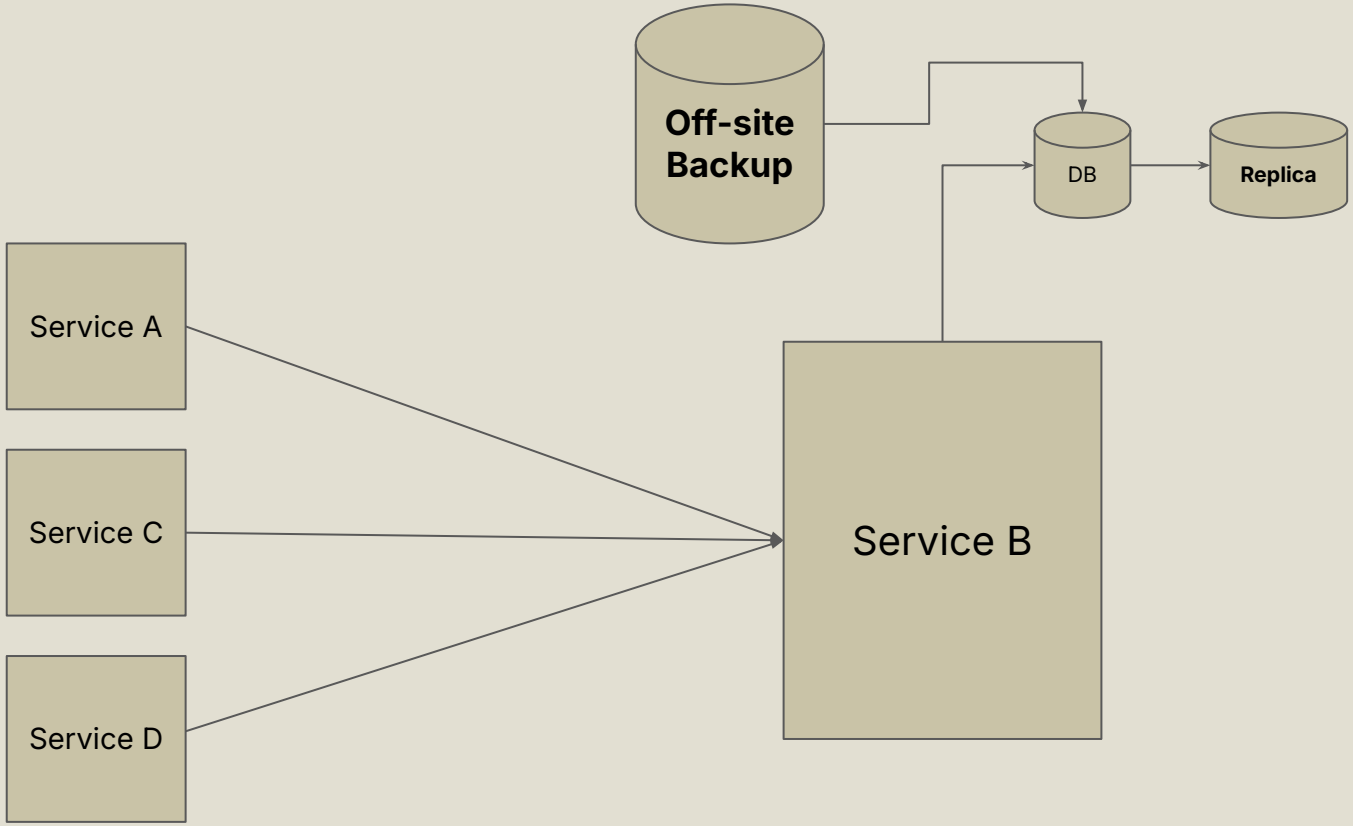
Solution	**Target RTO**
Daily Offsite Backup	30 mins ~ 1 hr
Warm standby (Log Shipping)	5~10 minutes (or less)
Hot standby (Streaming Replication)	5~10 minutes (or less)

Recovery Time Objective (RTO)

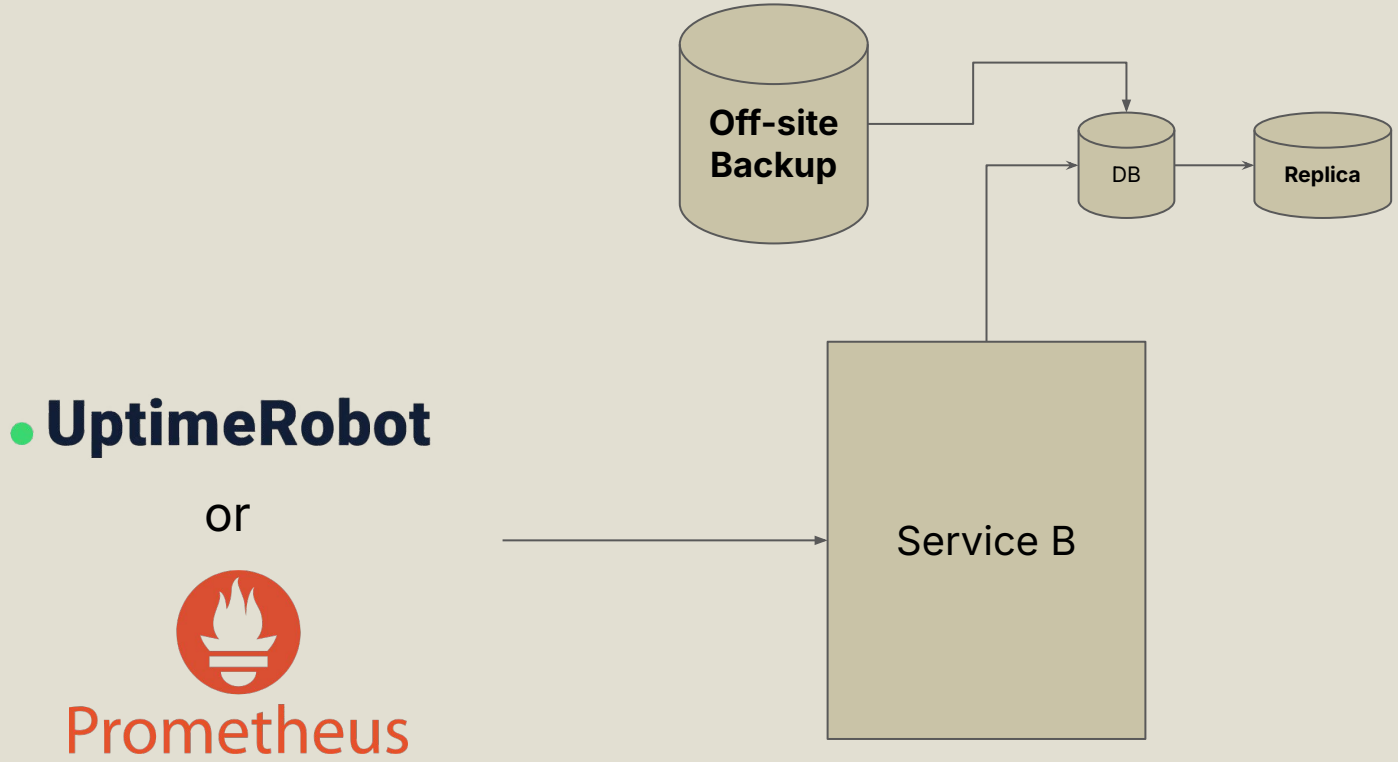
Both **RPO** and **RTO** needs to defined with the business to determine the right solution and investments.

Solution	**RPO**	**RTO**
Daily Offsite Backup	24 hrs	30 mins ~ 1 hr
Warm standby (Log Shipping)	3~5 minutes (or less)	5~10 minutes (or less)
Hot standby (Streaming Replication)	5 seconds (or less)	5~10 minutes (or less)

RTO + RPO



RTO: 30mins ~ 1 hr, RPO: 24 hours

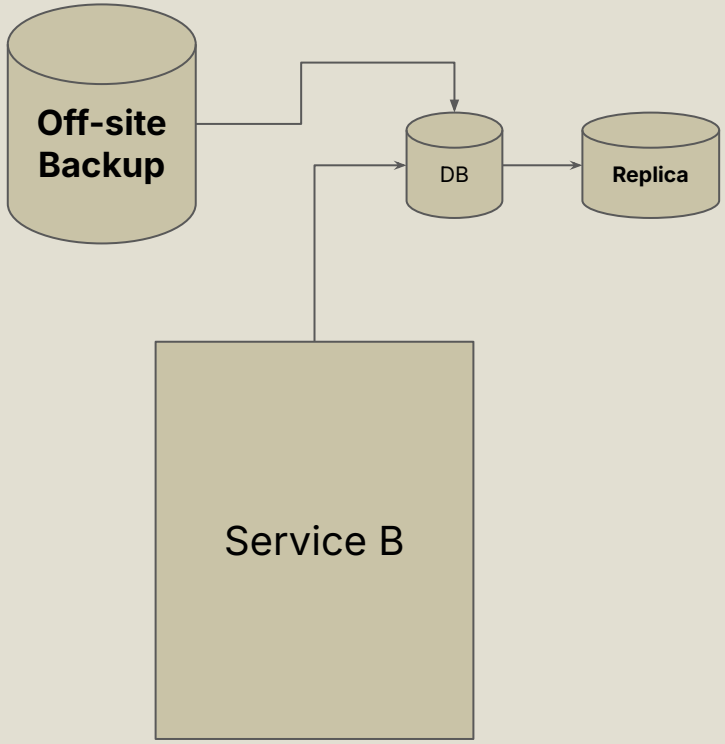


UptimeRobot

or

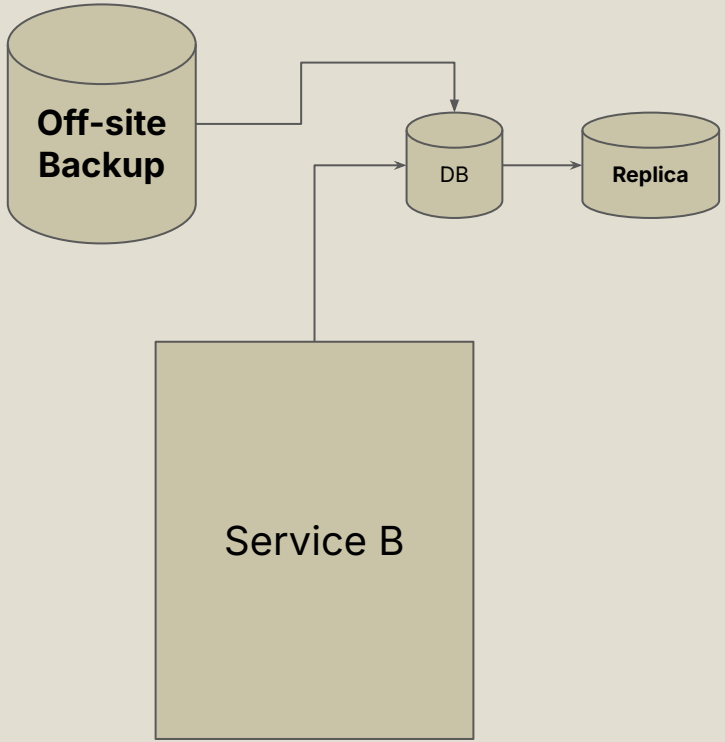
Prometheus

RTO: 1 hour, monitoring can be less “strict”



What does this mean for the human side of things?

- On-Call Rotation: At least 1 Engineer on standby
- Runbooks rehearsed every X months
- Off-site backup tested manually every X months

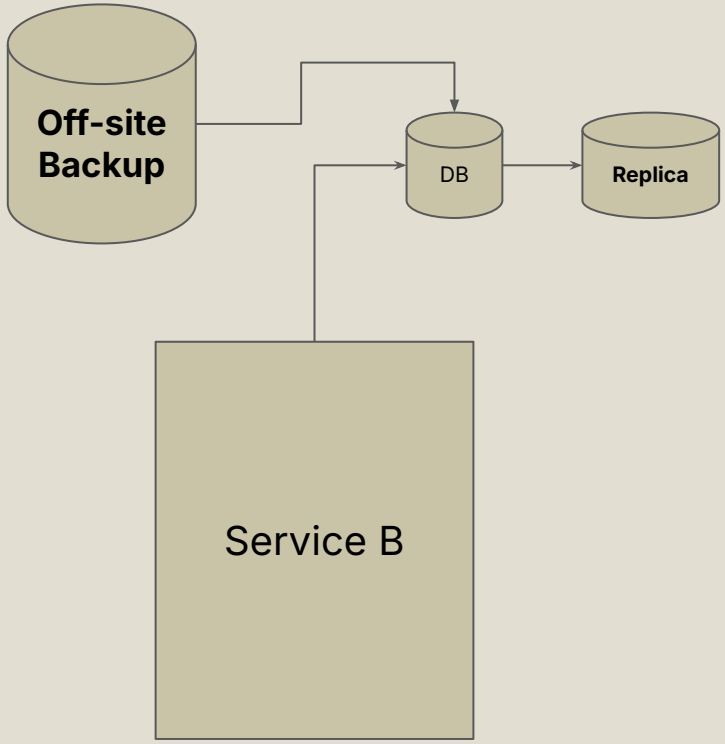


On-call, runbooks, testing backups: Time Investments

24 hours RPO is way too long!

Most companies cannot accept that we lose 24 hours of data.

Eg. 24 hours of sales data for a e-commerce platform might cause huge reputation and business loss



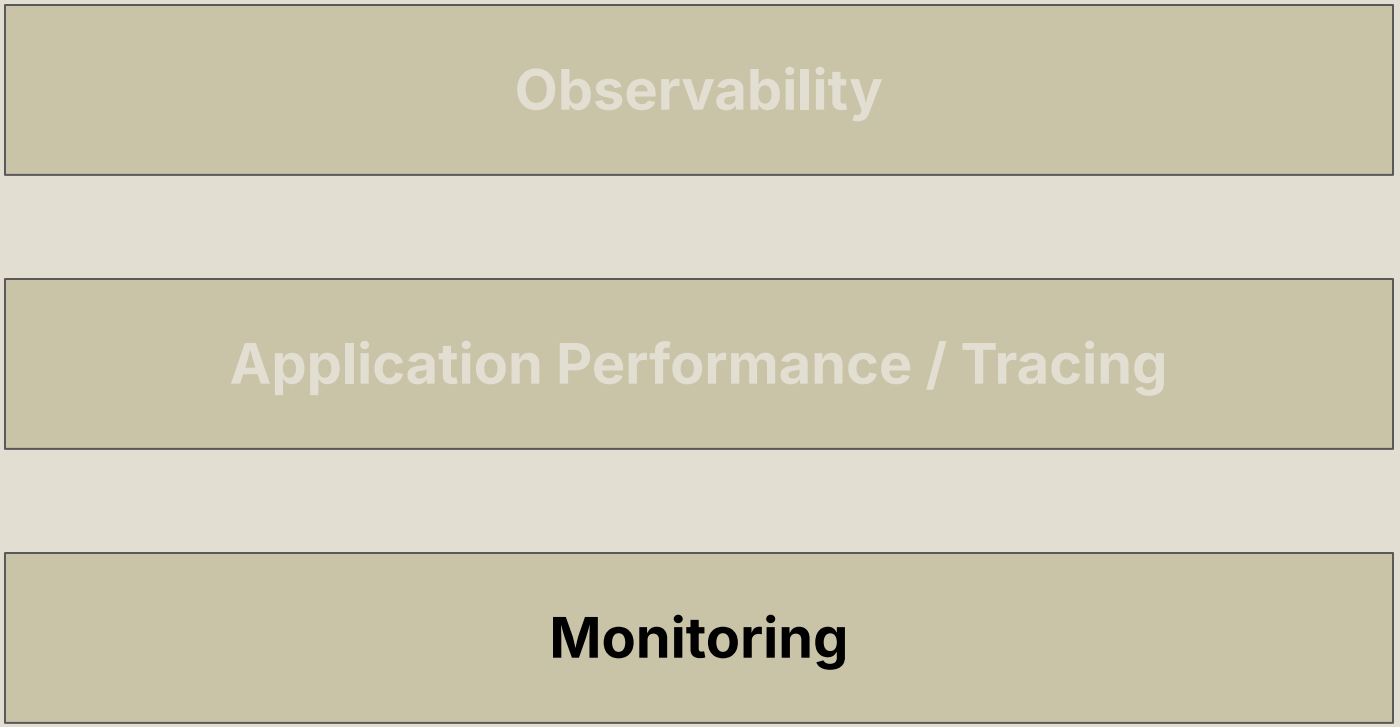
But in reality…

Offsite Backup: Disaster Recovery (Worst Case)

Hot standby: Able to quickly replace a dead primary DB

Solution	**RPO**	**RTO**
Daily Offsite Backup	**24 hrs**	**30 mins ~ 1 hr**
Warm standby (Log Shipping)	3~5 minutes (or less)	5~10 minutes (or less)
Hot standby (Streaming Replication)	**5 seconds (or less)**	**5~10 minutes (or less)**

Most Common Combination



Three “Tiers” of Observability

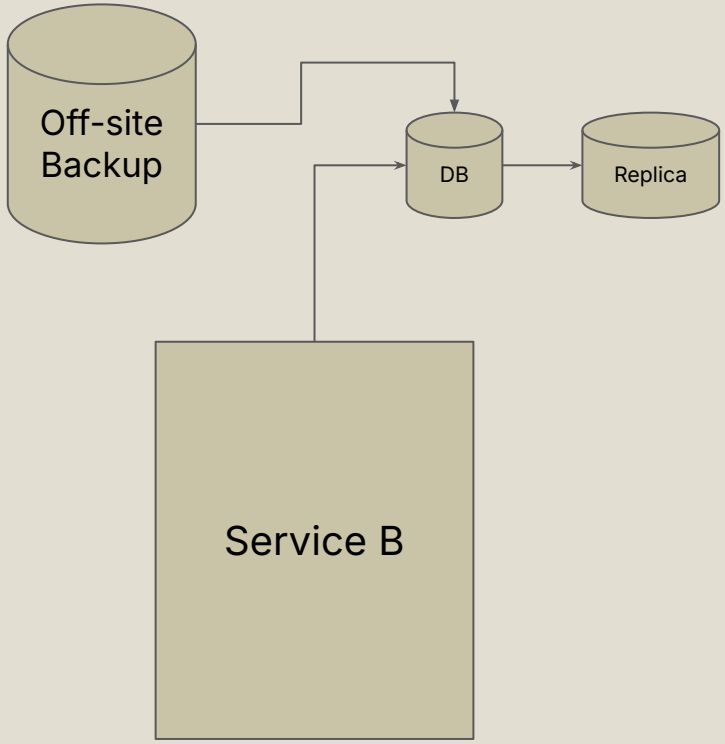
6.1.1 prometheus

“Prometheus collects and stores its metrics as time series data, i.e. metrics information is stored with the timestamp at which it was recorded, alongside optional key-value pairs called labels.”

Time Series DB

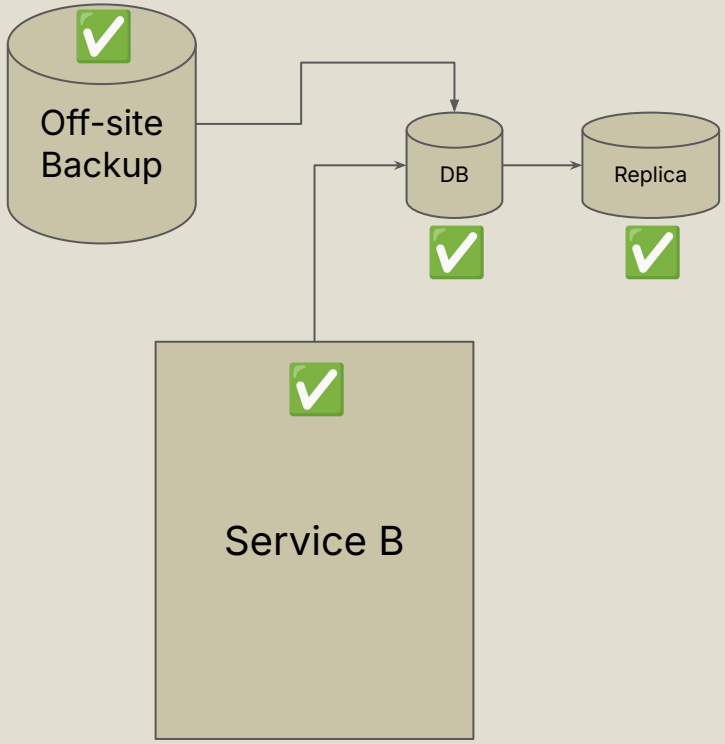
+

Visualization Webapp



Prometheus

What kind of metrics?



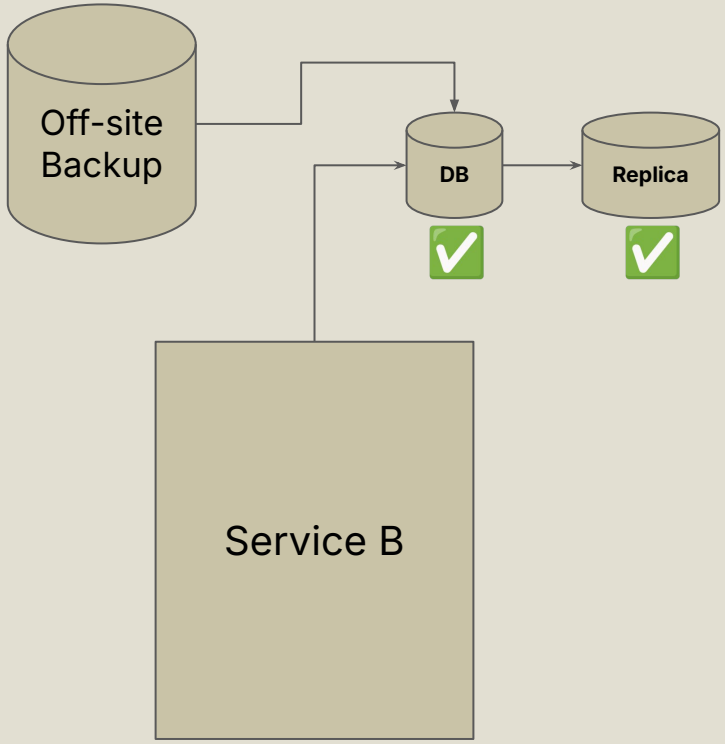
Prometheus

CPU Utilization, Memory Usage, Disk, etc.

PostgreSQL Server Exporter

Prometheus exporter for PostgreSQL server metrics.

CI Tested PostgreSQL versions: `11`, `12`, `13`, `14`, `15`, `16`, `17`.



PostgreSQL Server Exporter -> Prometheus

[no-]collector.replication Enable the replication collector (default: enabled).

[no-]collector.replication_slot Enable the replication_slot collector (default: enabled).

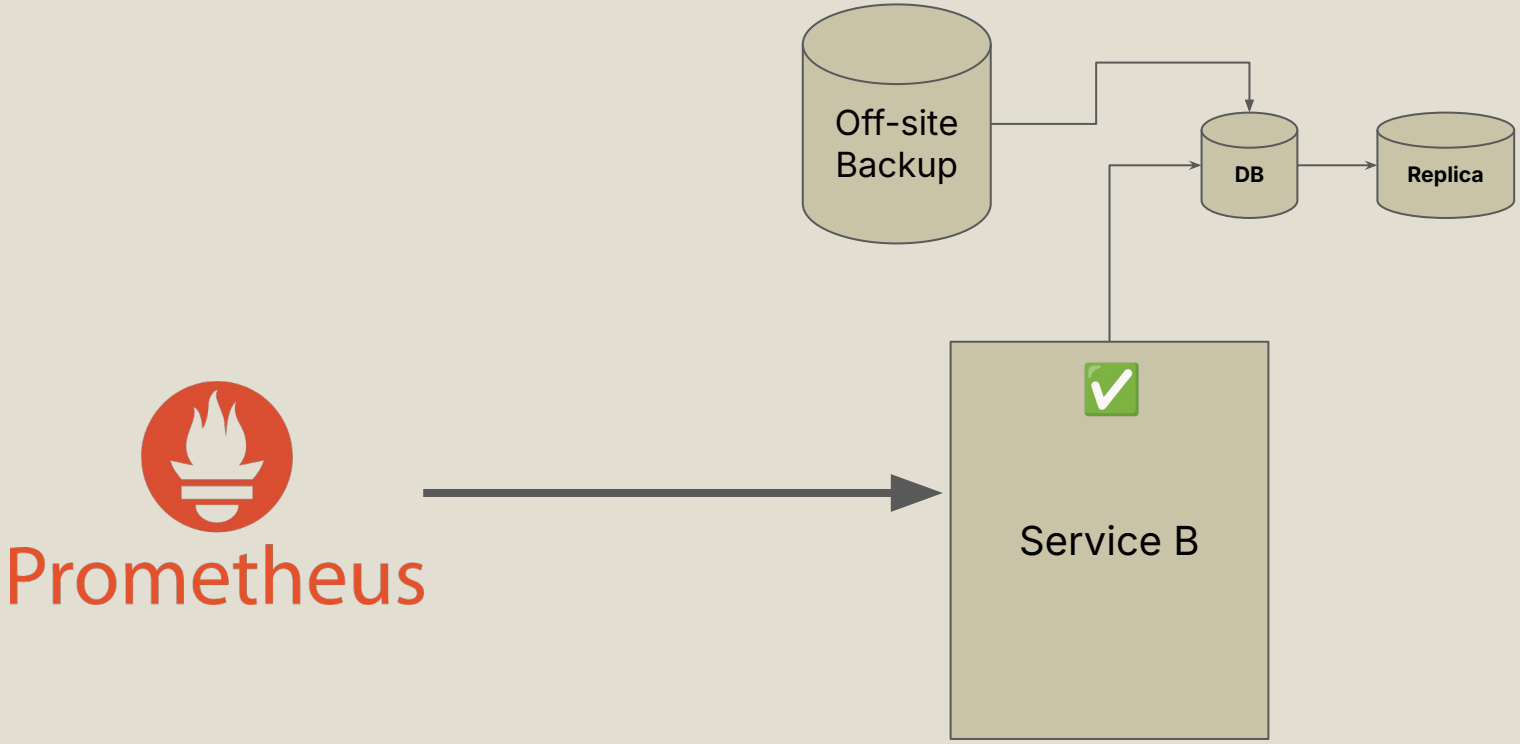
[no-]collector.stat_activity_autovacuum Enable the stat_activity_autovacuum collector (default: disabled).

[no-]collector.stat_database Enable the stat_database collector (default: enabled).

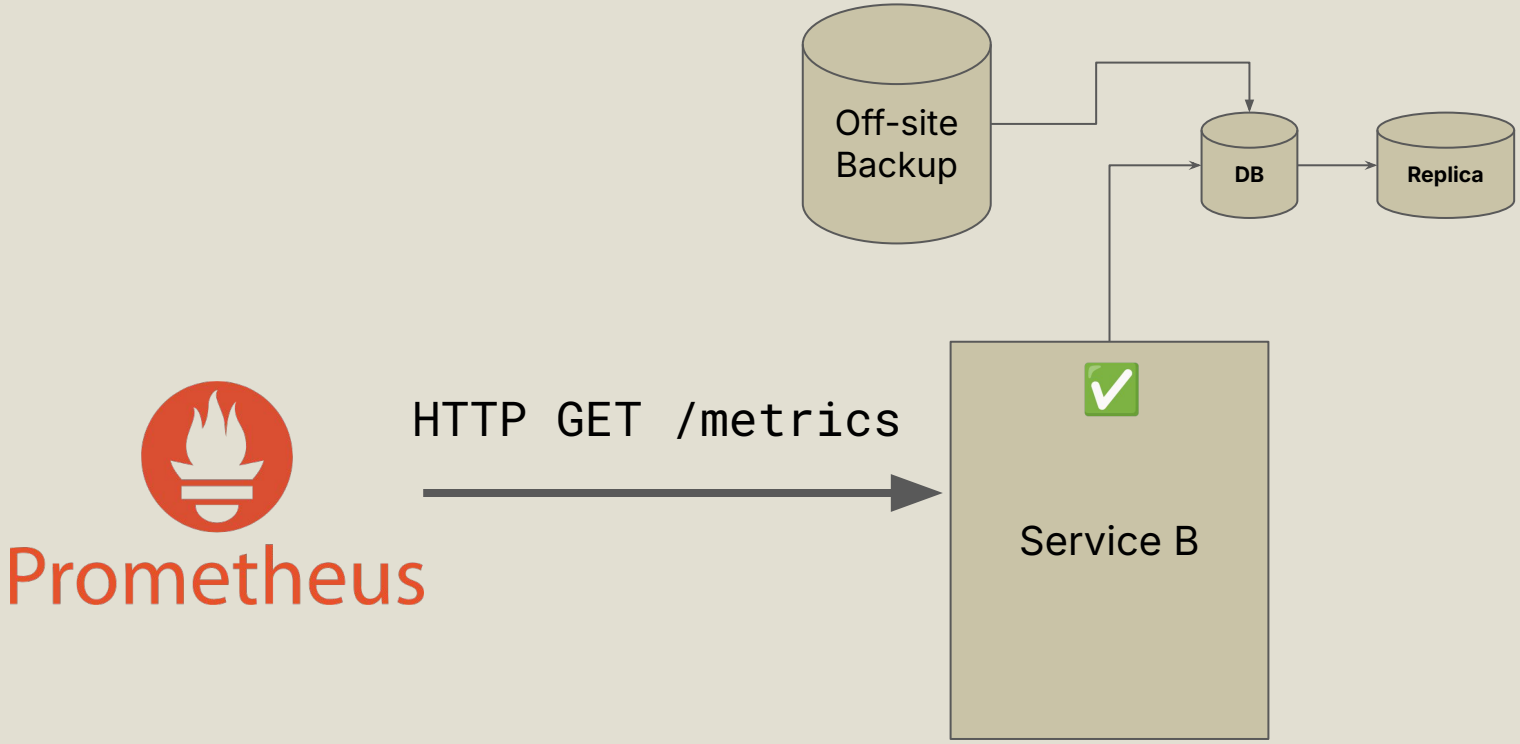
[no-]collector.stat_statements Enable the stat_statements collector (default: disabled).

… and more

PostgreSQL Server Exporter Metrics



Service Level Metrics?



Expose a metrics endpoint in the service

Metric	Description	Example Use Cases
Counter	A counter is a cumulative metric that represents a single monotonically increasing counter whose value can only increase or be reset to zero on restart.	Total visits to a website, total requests served
Gauge	A *gauge* is a metric that represents a single numerical value that can arbitrarily go up and down.	CPU utilization, Memory utilization

Prometheus Metric Types (i)

https://prometheus.io/docs/concepts/metric_types/

Metric	Description	Example Use Cases
Histogram	A histogram samples observations (usually things like request durations or response sizes) and counts them in configurable buckets. It also provides a sum of all observed values.	Target Request Time, eg. 95% of requests must be served within 300ms.
Summary	A summary samples observations (usually things like request durations and response sizes). While it also provides a total count of observations and a sum of all observed values, it calculates configurable quantiles over a sliding time window.	Response size

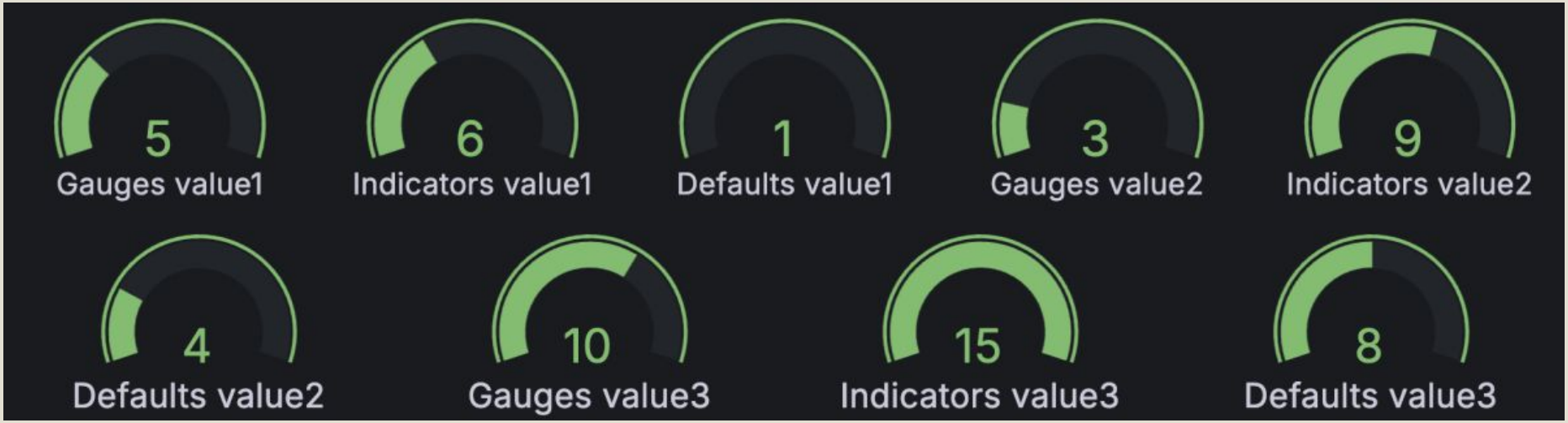
Prometheus Metric Types (ii)

https://prometheus.io/docs/concepts/metric_types/

Feature	Histogram	Summary
Percentile Calculation	Calculated server-side using PromQL (histogram_quantile)	Precomputed client-side during metric collection **You cannot recompute again when building dashboards!**
Aggregation Across Instances	Supported	Not supported
Use Case	Distribution tracking (e.g., latency)	Fixed percentiles (e.g., response size)

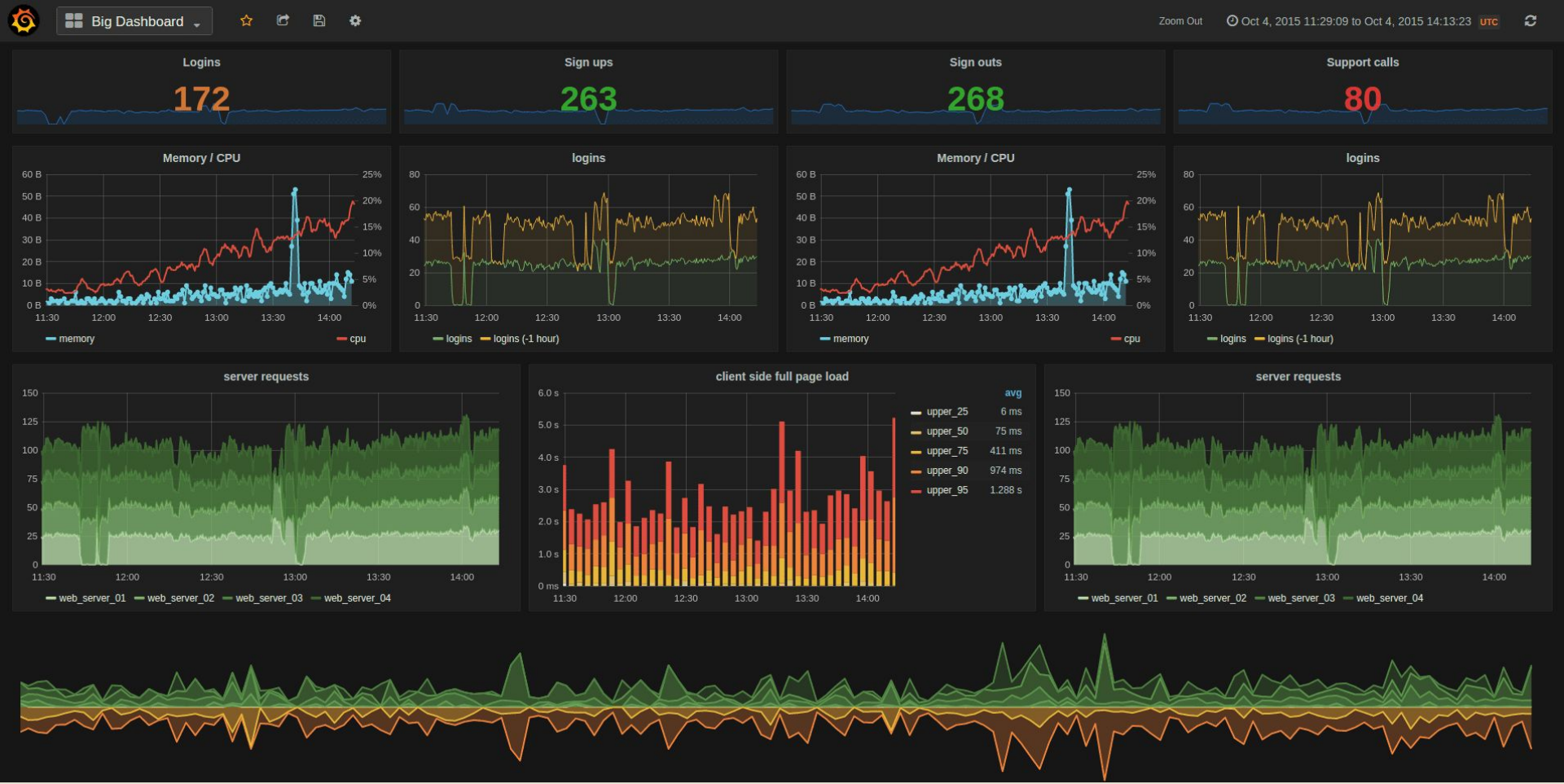
Prometheus: Histogram and Summary

https://prometheus.io/docs/practices/histograms/



Grafana: Gauges

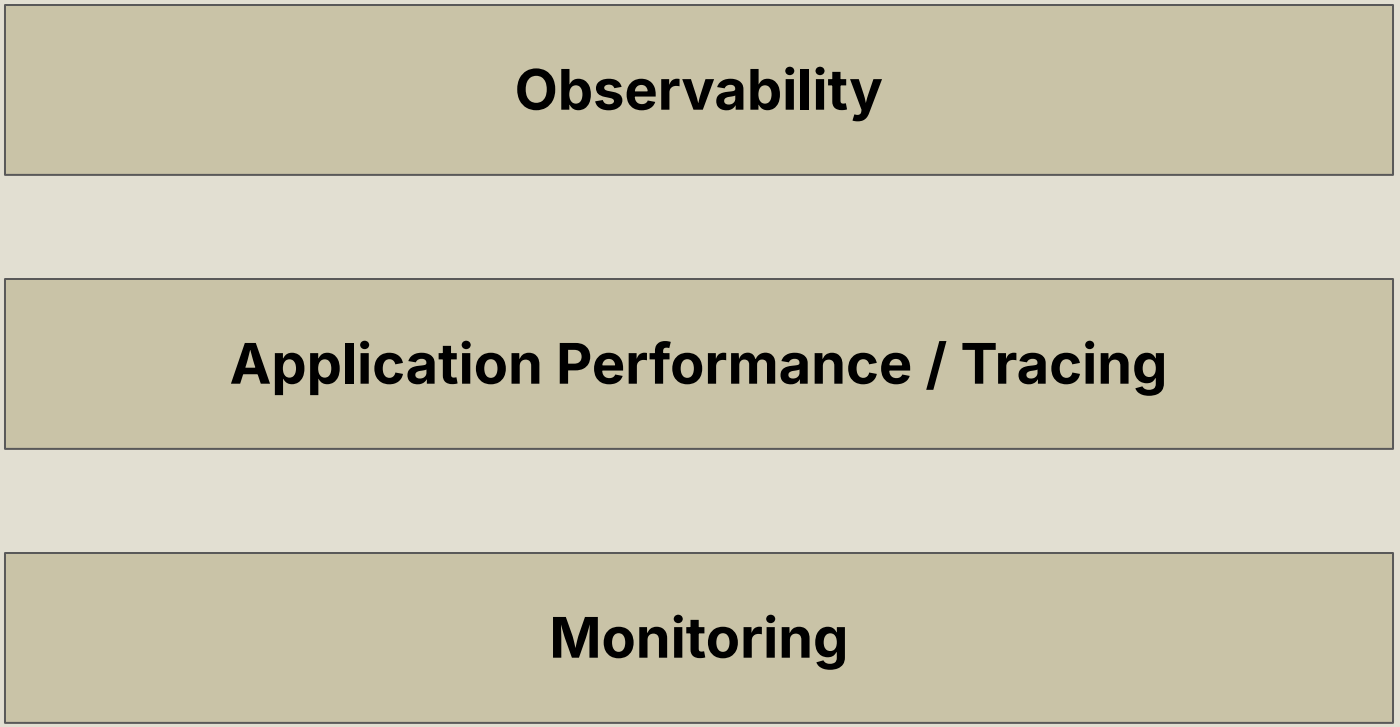
https://grafana.com/docs/grafana/latest/panels-visualizations/visualizations/gauge/



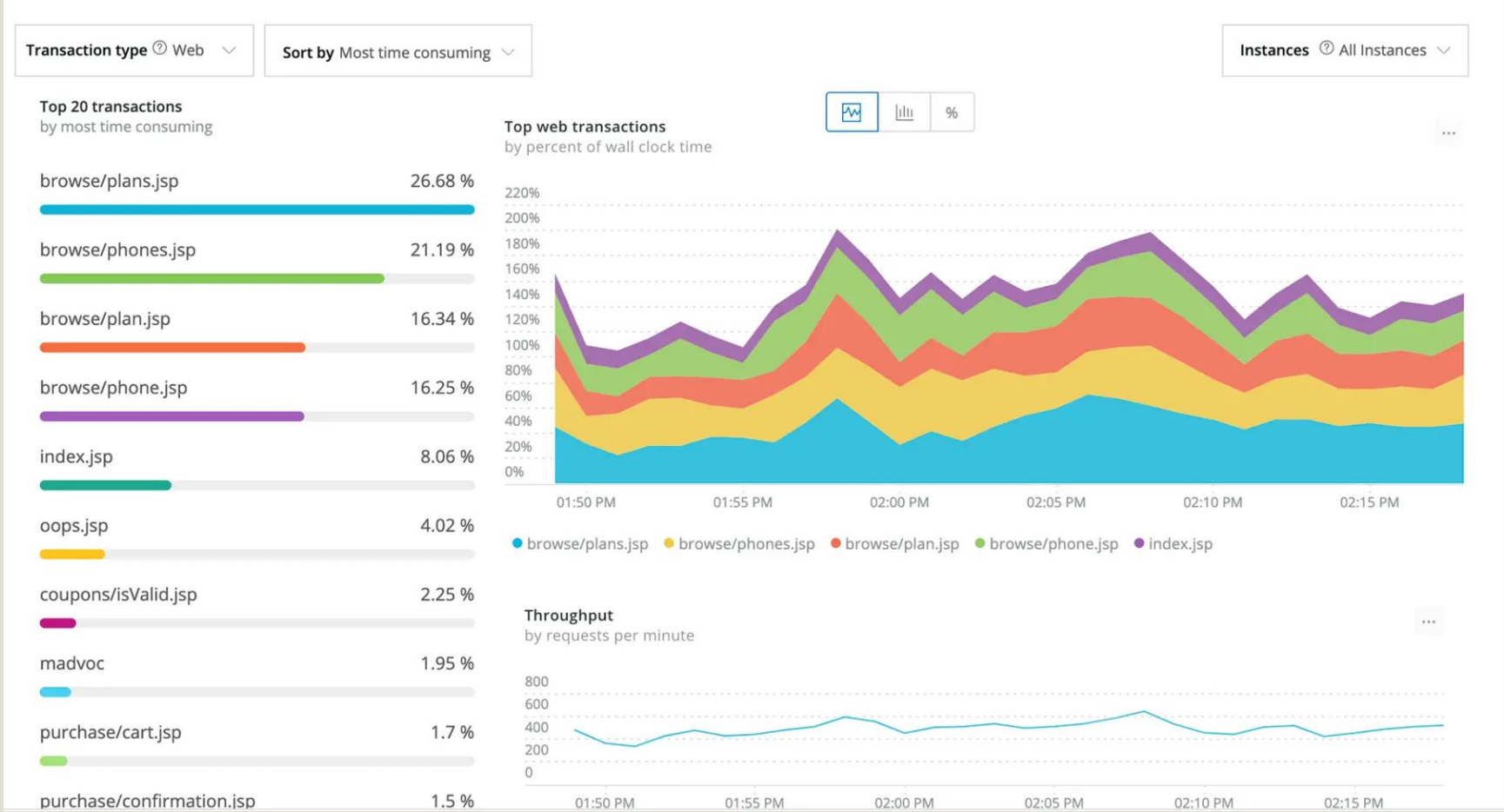
Grafana: Example

https://scaleyourapp.com/what-is-grafana-why-use-it-everything-you-should-know-about-it/

6.1.2 apm and tracing



Three “Tiers” of Observability

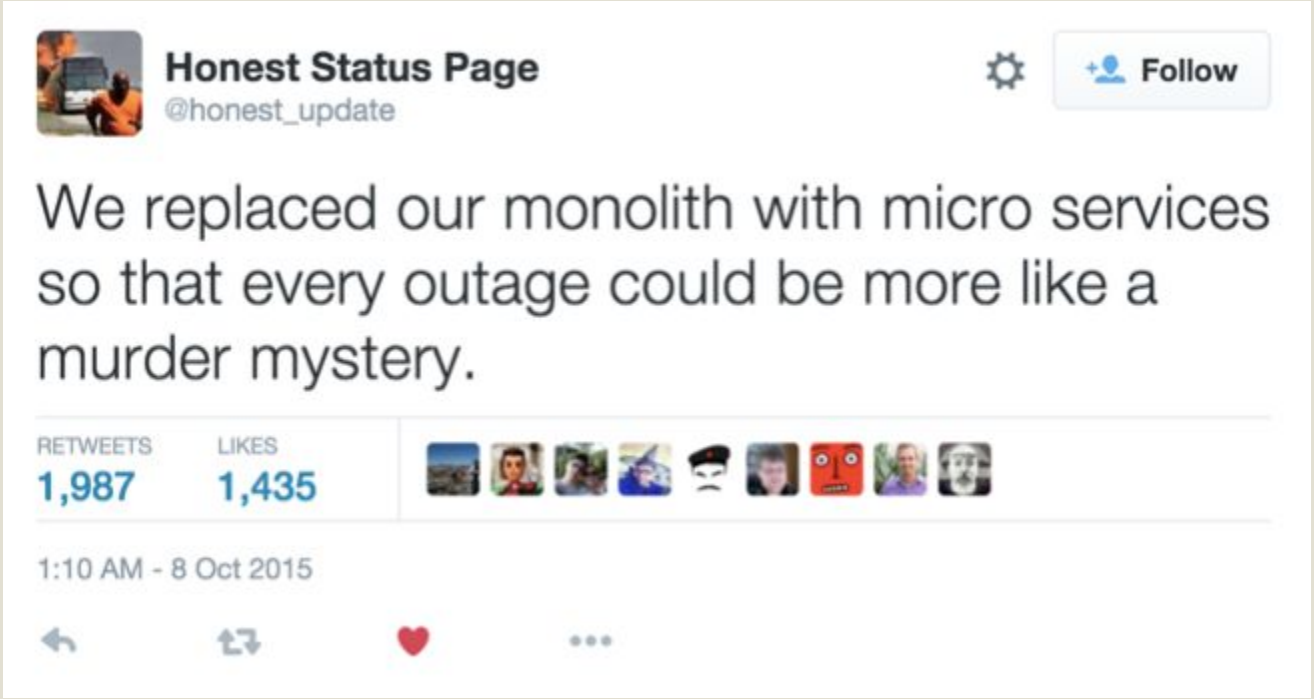


New Relic: Application Performance Monitoring (APM)

https://newrelic.com/blog/best-practices/how-to-monitor-with-apm

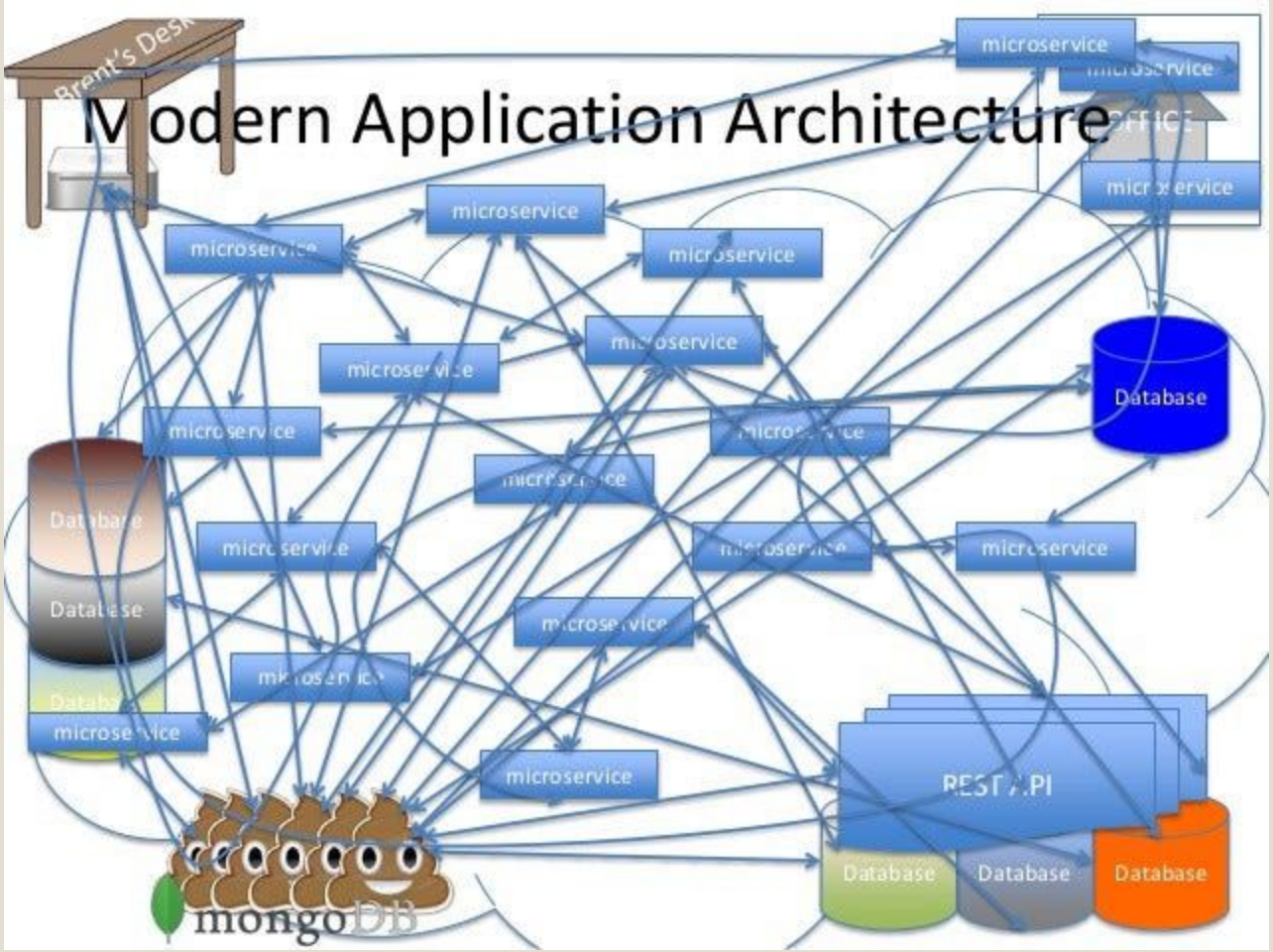
	Monitoring	**APM**
Intent	Monitor system and application metrics	Application-focused metrics broken down with traces
Types of Metrics	Anything*	Application metrics only, eg. request time, function call time
Instrumentation Method	Manual or via tools	Usually automatically captured once you install the SDK

Monitoring vs APM



A Case for Microservices

https://peter.bourgon.org/a-case-for-microservices/



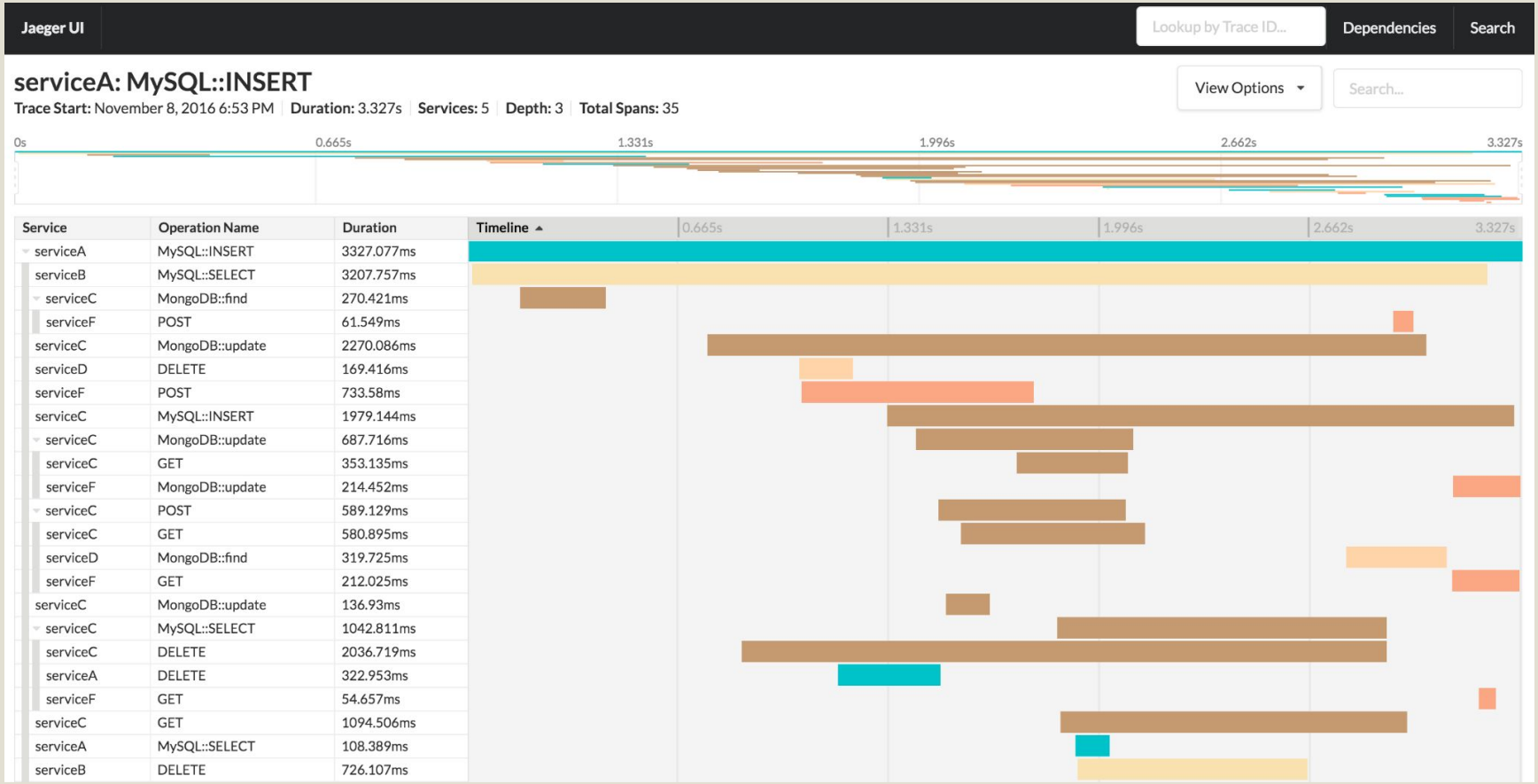
"Modern Application Architecture"

https://medium.com/@damonallison/evolving-microservices-with-event-sourcing-7396e015cfd2

JAEGER

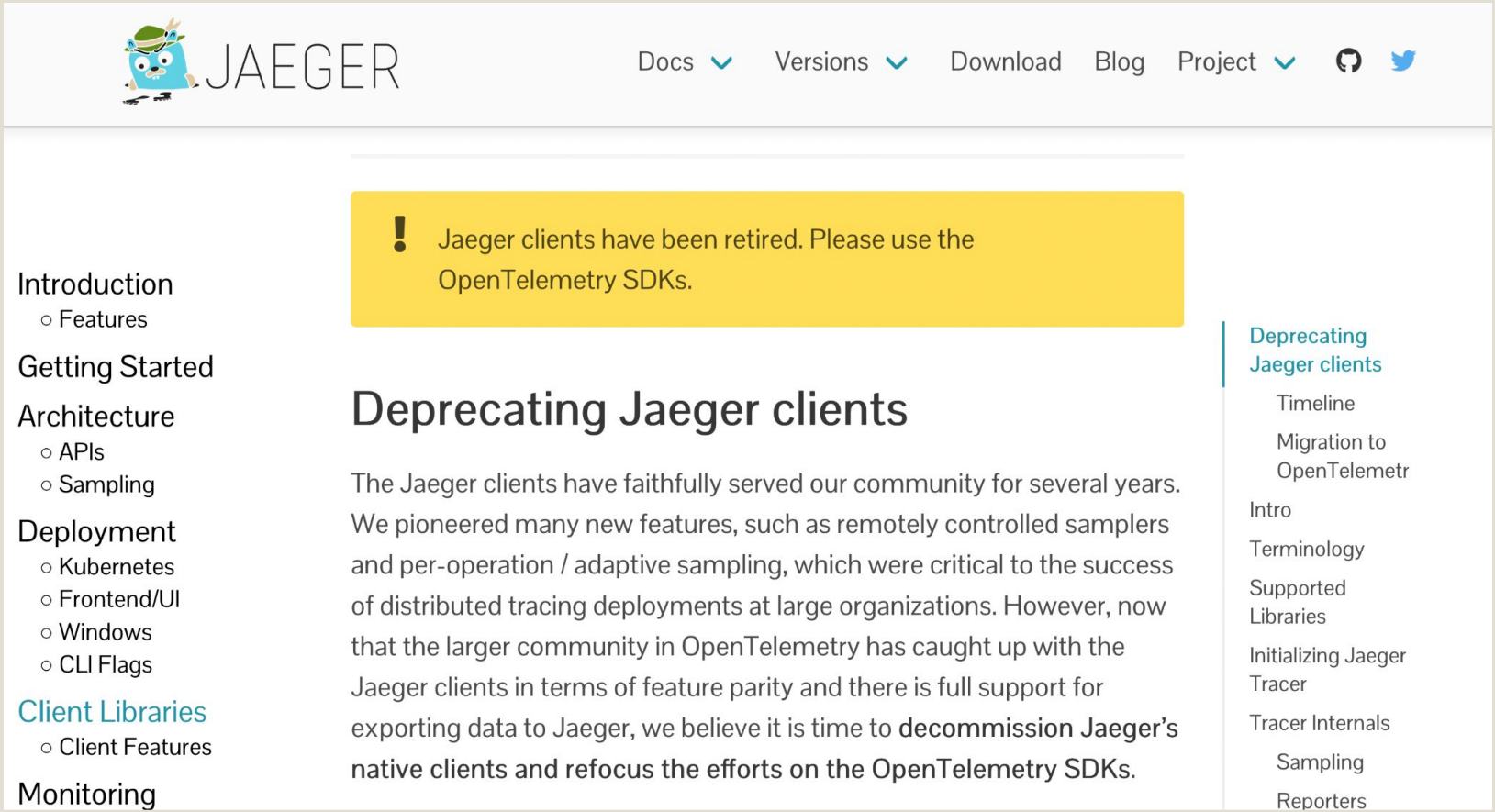
Jaeger: Distributed Tracing

https://www.jaegertracing.io/



Evolving Distributed Tracing at Uber Engineering

https://www.uber.com/en-AU/blog/distributed-tracing/



Deprecating Jaeger clients

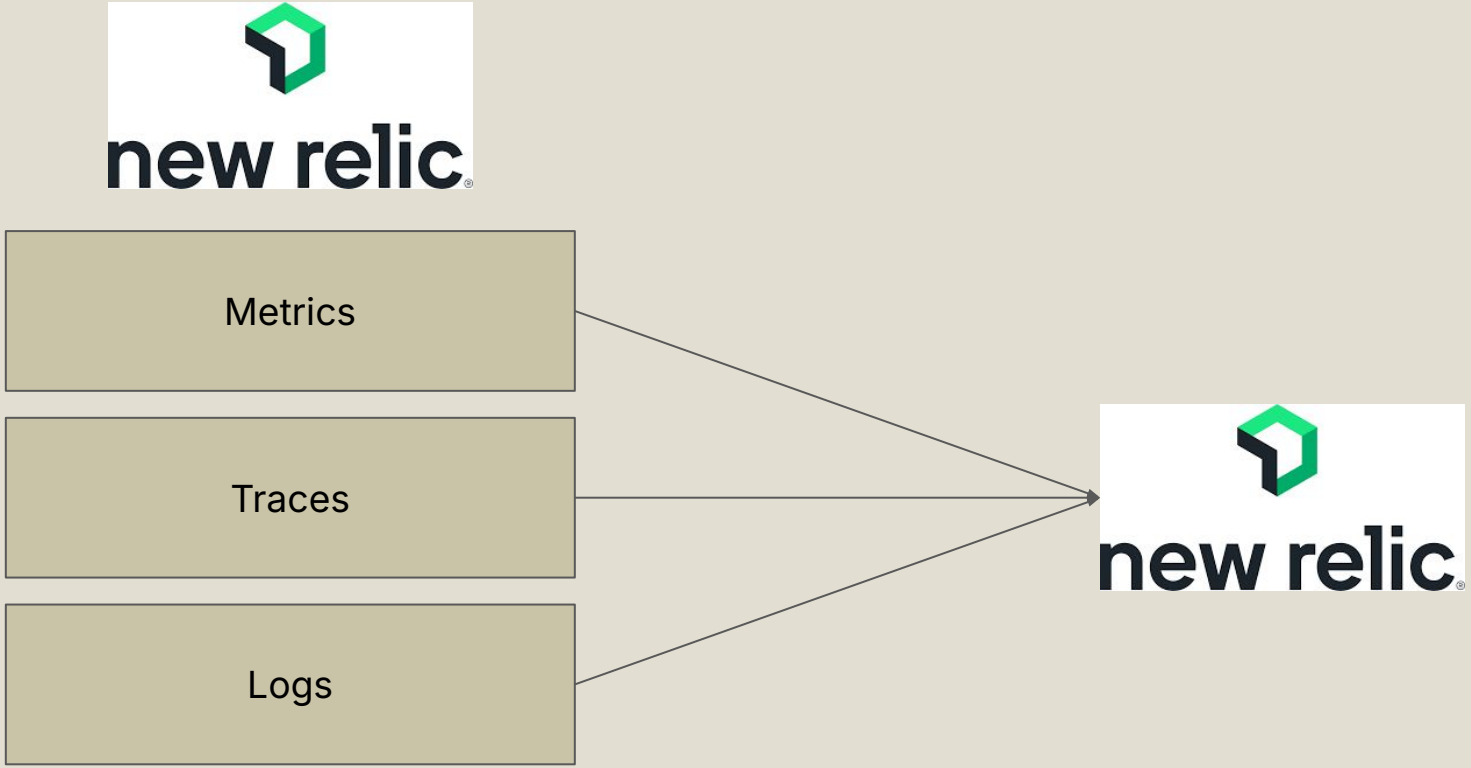
https://www.jaegertracing.io/docs/1.17/client-libraries/

OpenTelemetry is:

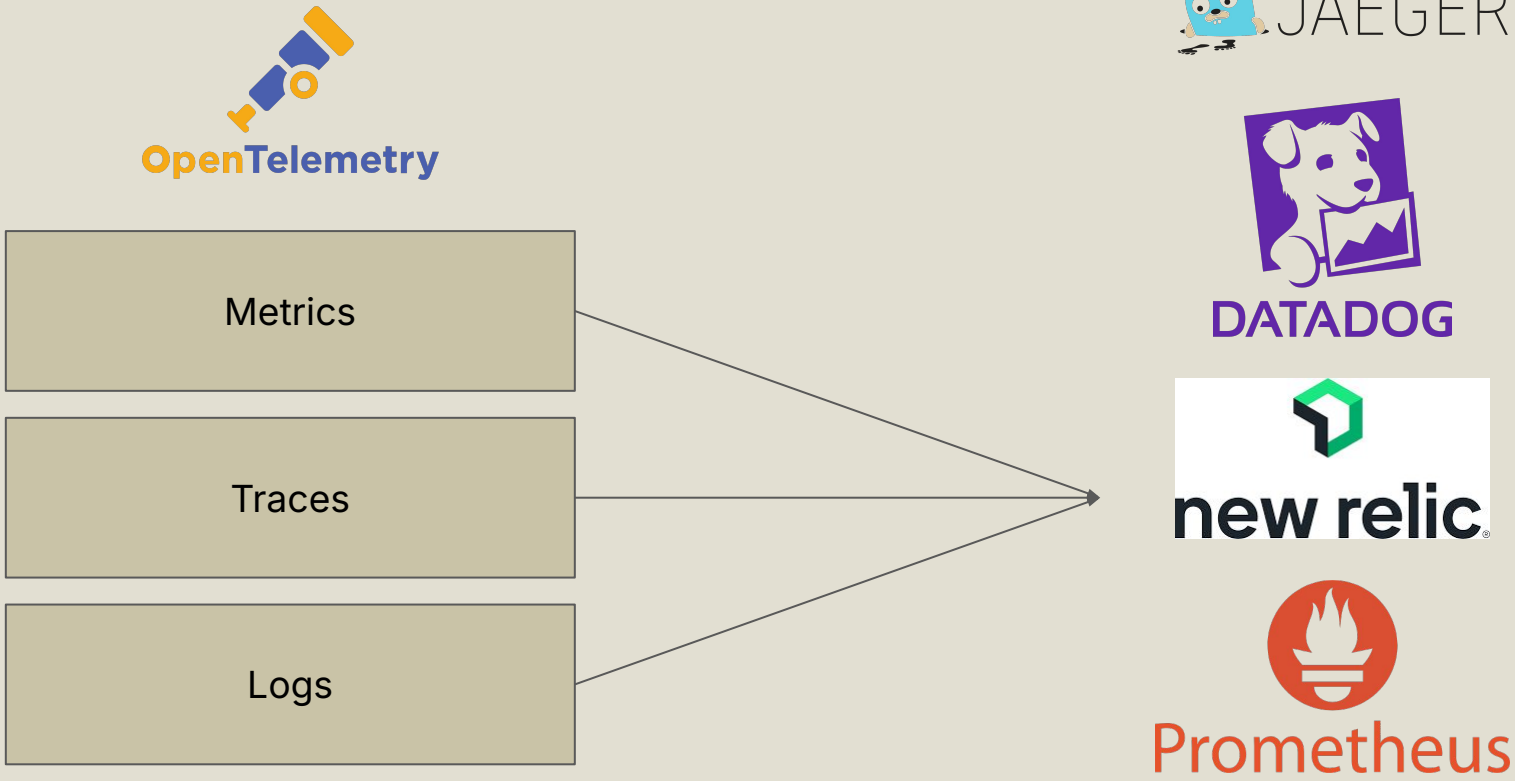
- An **observability framework and toolkit** designed to facilitate the
  - Generation
  - Export
  - Collection
- of telemetry data such as traces, metrics, and logs.
- **Open source**, as well as **vendor- and tool-agnostic**, meaning that it can be used with a broad variety of observability backends, including open source tools like Jaeger and Prometheus, as well as commercial offerings.
- **OpenTelemetry is not an observability backend itself.**

What is OpenTelemetry?

https://opentelemetry.io/docs/what-is-opentelemetry/



Past: SDKs for Each Vendor or Platform

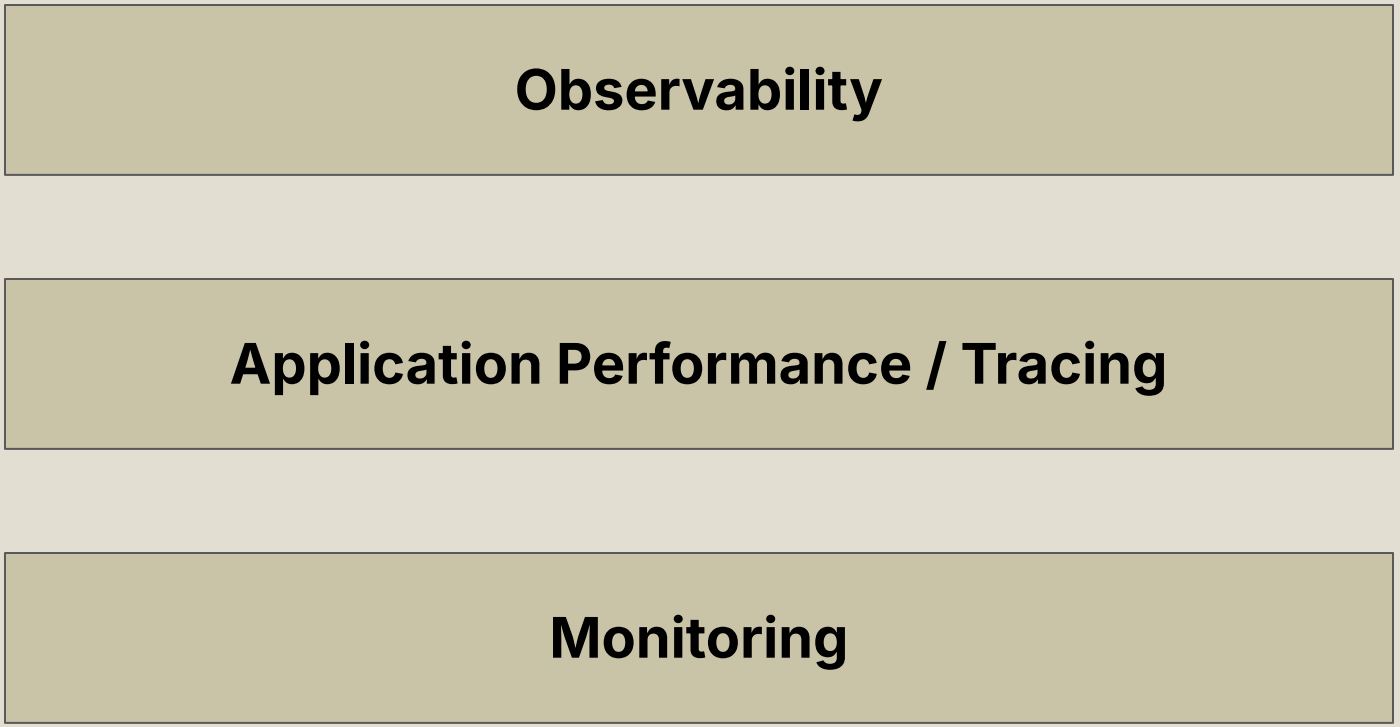


Now: OTel SDKs for All Vendors / Platform

Language	Traces	Metrics	Logs
C++	Stable	Stable	Stable
C#/.NET	Stable	Stable	Stable
Erlang/Elixir	Stable	Development	Development
Go	Stable	Stable	Beta
Java	Stable	Stable	Stable
JavaScript	Stable	Stable	Development
PHP	Stable	Stable	Stable
Python	Stable	Stable	Development
Ruby	Stable	Development	Development
Rust	Beta	Beta	Beta
Swift	Stable	Development	Development

OTel SDK Availability

https://opentelemetry.io/docs/languages/



Three “Tiers” of Observability

Uptime Target: 99.99%, etc.

Recovery Point Objective (RPO): The maximum amount of data loss (measured by time) that an organization can tolerate.

Recovery Time Objective (RTO): The maximum length of time it should take to restore normal operations following an outage.

Three Key Metrics

6.1.3 case studies

Monitoring

Monitoring focuses on tracking and alerting on known, predefined metrics and states.

It answers the question: "Is the system working as expected?"

Observability

Observability provides insights into unknown or unexpected behaviors by collecting rich, contextual data.

It answers the question: "Why is the system not working as expected?"

Monitoring: A monitoring tool flags that disk usage on a server has reached 95%, triggering an alert.

Scenario 1: Disk Usage Alert

Monitoring: A monitoring tool flags that disk usage on a server has reached 95%, triggering an alert.

Observability: Observability tools analyze logs and traces to determine why disk usage spiked.

For instance, they reveal that a backup script ran unexpectedly due to a misconfigured cron job.

Scenario 1: Disk Usage Alert

Monitoring: Monitoring detects increased response times in a web application after a code deployment.

Scenario 2: Slow API Response

Monitoring: Monitoring detects increased response times in a web application after a code deployment.

Observability: Observability traces the issue across microservices and identifies that the new code deployed 20 minutes ago introduced inefficient database queries, causing bottlenecks.

Scenario 2: Slow API Response

Scenario

Your company operates a global e-commerce platform deployed across three geographic regions (US, Europe, Asia) with multiple availability zones in each region. The architecture uses a combination of:

- IP address-based (proxy for location) load balancing
- Microservices using HTTP for inter-service communication
- Auto-scaling CockroachDB clusters
- Multiple load balancing algorithms (round-robin, least connections, weighted response time) for database connections

The system is experiencing intermittent performance issues, with some users reporting slow response times and occasional errors during peak traffic periods.

Case Study: Multi-Region Deployment with Dynamic Traffic

Monitoring: What's Visible

- Regional load balancer health checks (**pass**)
- Request count per region based on IP distribution (**regular**)
- Error rates per microservice (**increasing**)
- Average response times (**increasing**)
- CockroachDB node CPU/memory utilization (**very high**)
- Connection counts to database clusters (**normal**)
- HTTP 5xx errors spiking periodically

Case Study: Multi-Region Deployment with Dynamic Traffic

Observations and Action

1. Alert triggers show increased latency for database operations
2. Dashboards indicate all CockroachDB nodes are up and passing health checks
3. Metrics show IP-based routing is distributing traffic to appropriate regions
4. Scale up CockroachDB clusters based on connection count rules

Action: Add more database capacity to all regions

Case Study: Multi-Region Deployment with Dynamic Traffic

Result

Despite adding more capacity, the problems persist. The monitoring data shows symptoms but doesn't reveal why certain requests are slow while others work fine.

The round-robin algorithm appears to be working correctly based on connection distribution metrics

Case Study: Multi-Region Deployment with Dynamic Traffic

Observability: What's Visible

- End-to-end request traces across all microservices and database calls
- High-cardinality (highly unique) data including user IP blocks, query patterns, and request payloads
- Per-request timing breakdowns with detailed SQL query information
- Load balancing decision logs with IP-based routing explanations
- Database query execution plans with timing information
- Custom attributes for business transaction types
- Detailed HTTP call patterns between microservices

Case Study: Multi-Region Deployment with Dynamic Traffic

Investigation Process

1. Analyze trace data for slow requests, revealing:
    - Users from specific IP ranges (East Coast US corporate networks) are experiencing slowdowns
    - These requests involve complex joins across multiple tables
    - **Queries consistently hit specific CockroachDB nodes due to data locality**

2. Deep-dive into database telemetry shows:
    - **Round-robin load balancing is sending queries to nodes that don't have local data**
    - **Range splits are occurring during peak traffic periods**
    - **Some queries are causing excessive network traffic between regions**

3. Examine HTTP inter-service communication patterns:
    - Certain microservices make excessive sequential database calls
    - No circuit breakers are implemented for inter-service communication
    - Thundering herd occurs when database latency increases

Case Study: Multi-Region Deployment with Dynamic Traffic

Root Cause Identification

The combination of observability data reveals multiple contributing factors:

- IP-based routing is correctly sending users to their regional endpoints, but database queries don't respect the same locality
- **Round-robin load balancing for database connections doesn't account for CockroachDB's data distribution**
- **Complex queries from specific user segments are causing range scans across multiple regions**
- Microservices are making chatty (frequent) HTTP calls instead of batching operations

Case Study: Multi-Region Deployment with Dynamic Traffic

Possible Solutions

1. Implement database request routing aware of CockroachDB range locality (Follow-the-Workload Topology)
2. **Switch from round-robin to weighted response time for database connection load balancing**
3. Optimize schema design to improve data locality for common query patterns
4. **Implement circuit breakers and backoff policies for inter-service HTTP calls**
5. **Batch database operations in high-volume microservices**

Case Study: Multi-Region Deployment with Dynamic Traffic

Monitoring

- Showed that something was wrong (high latency, errors)
- Indicated which database operations were affected
- Provided metrics on system health and resource usage
- Led to generic scaling solutions that didn't solve the root problem

Observability

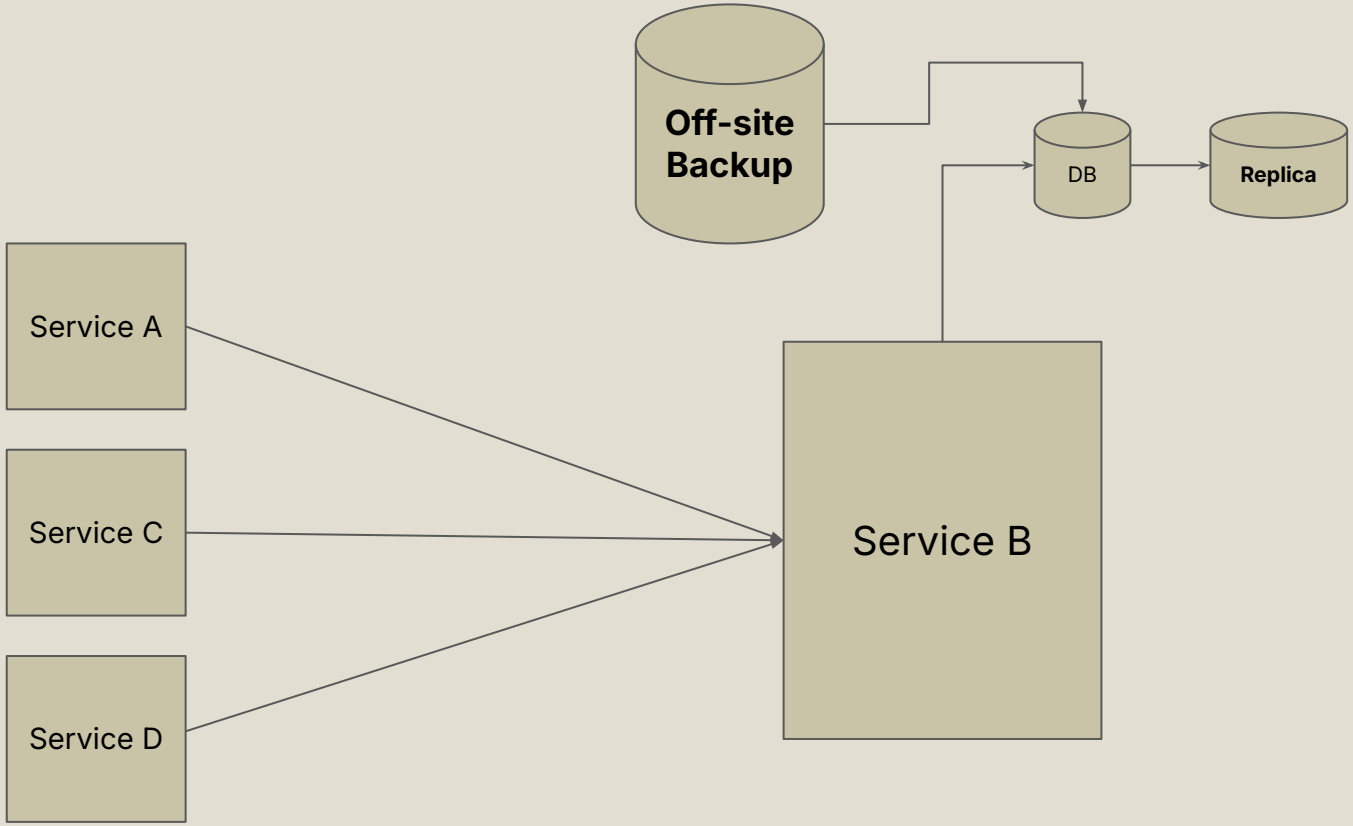
- Identified the interaction between IP-based routing and CockroachDB data locality
- Exposed the mismatch between load balancing strategy and distributed database architecture
- Showed how retry policies contributed to cascading failures
- Enabled targeted optimizations addressing the complex interplay between components

Monitoring & Observability: Summary

1. Monitoring focuses on tracking and alerting on known, predefined metrics and states.
2. Observability provides insights into unknown or unexpected behaviors by collecting rich, contextual data.
3. Prometheus is an OpenTelemetry compatible time-series database for storing metrics, and Jaeger can be used for distributed tracing between services
4. Distributed systems need observability, otherwise you will be dealing with murder mysteries every time there is an incident
5. Invest accordingly to meet your RTO and RPO objectives

short break

6.2 load testing

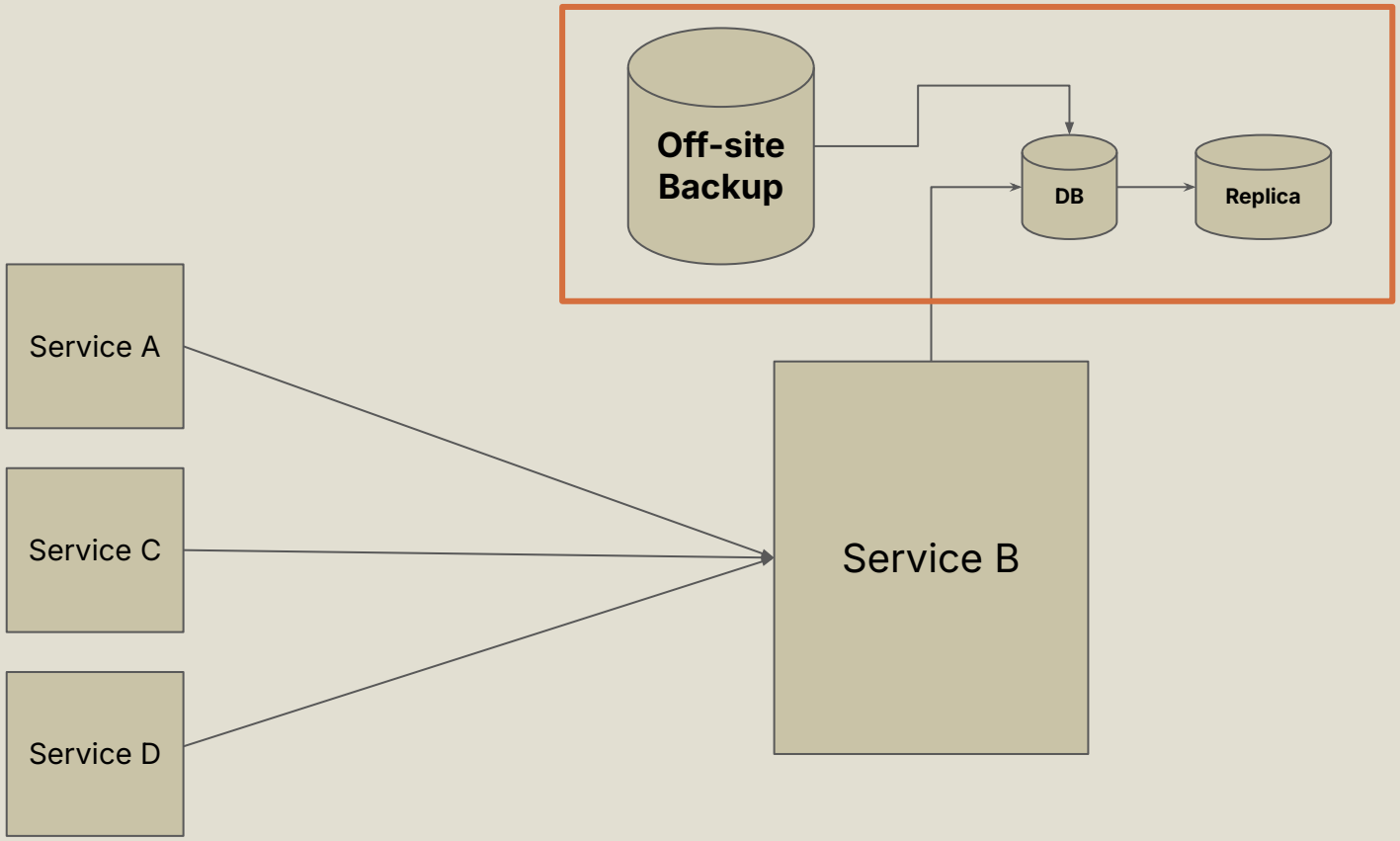


Load Testing: What to test?

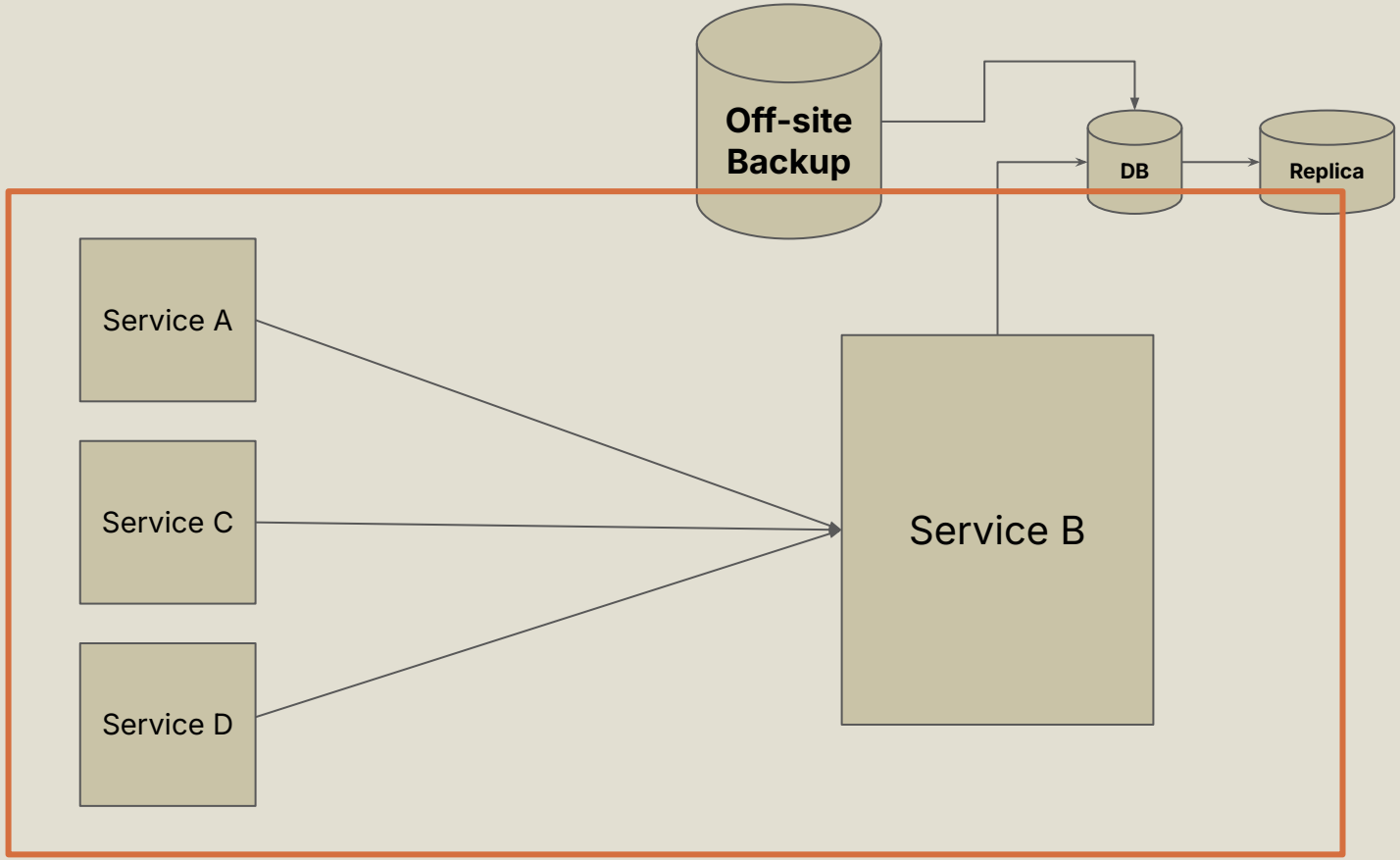
Peer Discussion

Discuss about what to test for when designing a load testing plan for distributed systems in different industries, eg. e-commerce, ride sharing, etc.

~15 mins, pick 1 person to share after discussing.



Load Testing: Database Load and Traffic



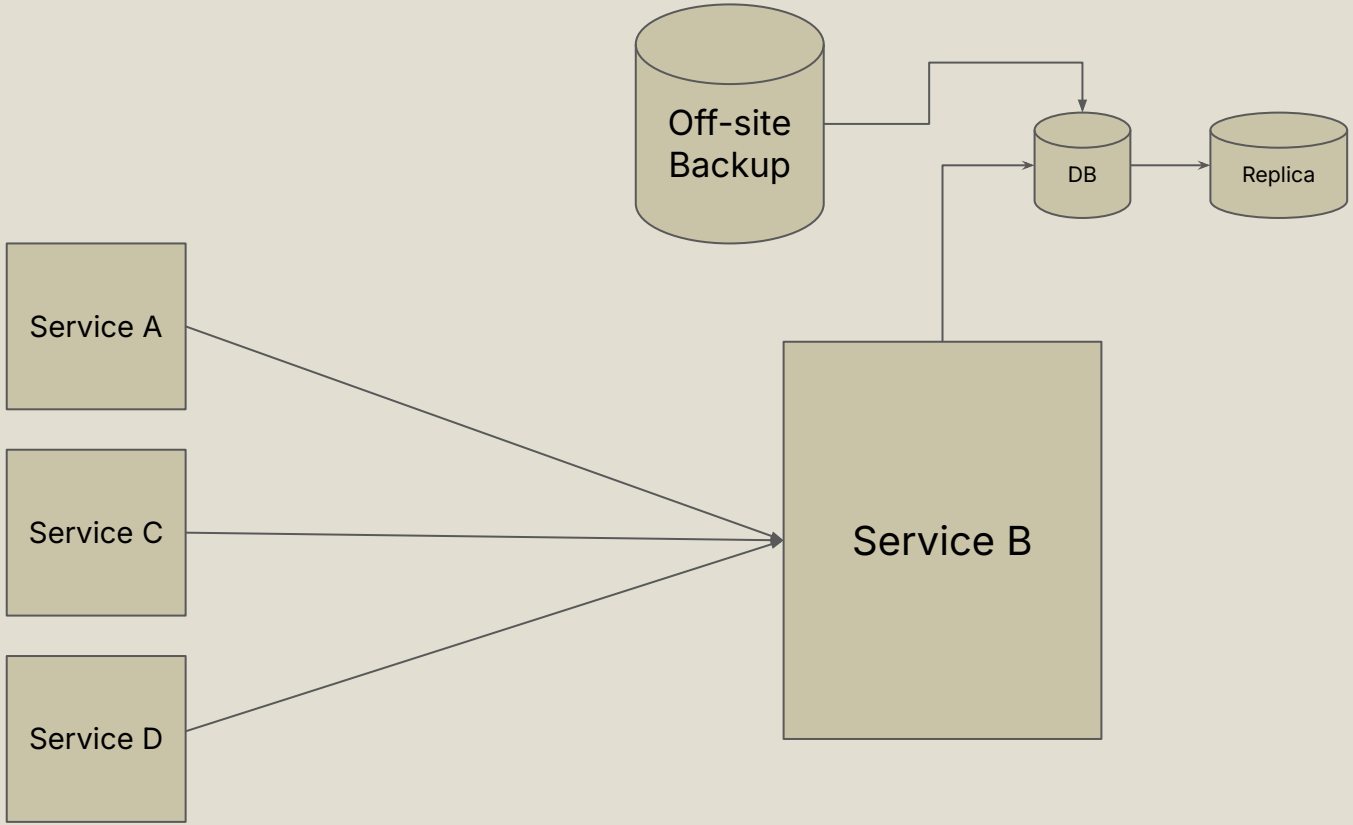
Load Testing: Traffic, Retries, Error Handling etc.

Uptime Target: 99.99%, etc.
95th Percentile Response Time Target: 750ms

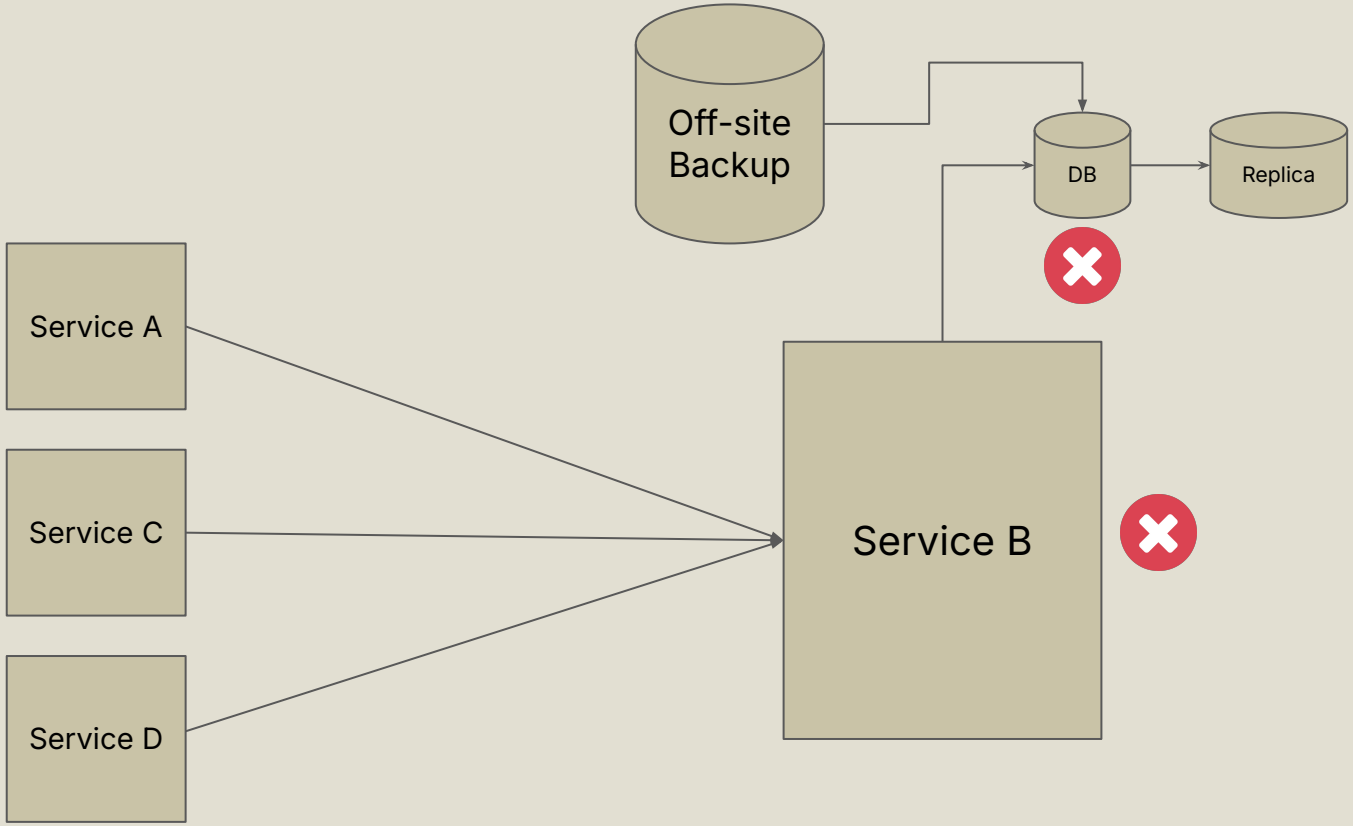
We can achieve both targets if our traffic remains below X requests per second. We can meet our response time target unless we face a spike in traffic that is Y times our normal traffic load.

Finding X and Y is one of the objectives of load testing.

Objective 1: Finding the Limit



Objective 2: Verify Correct Behaviour

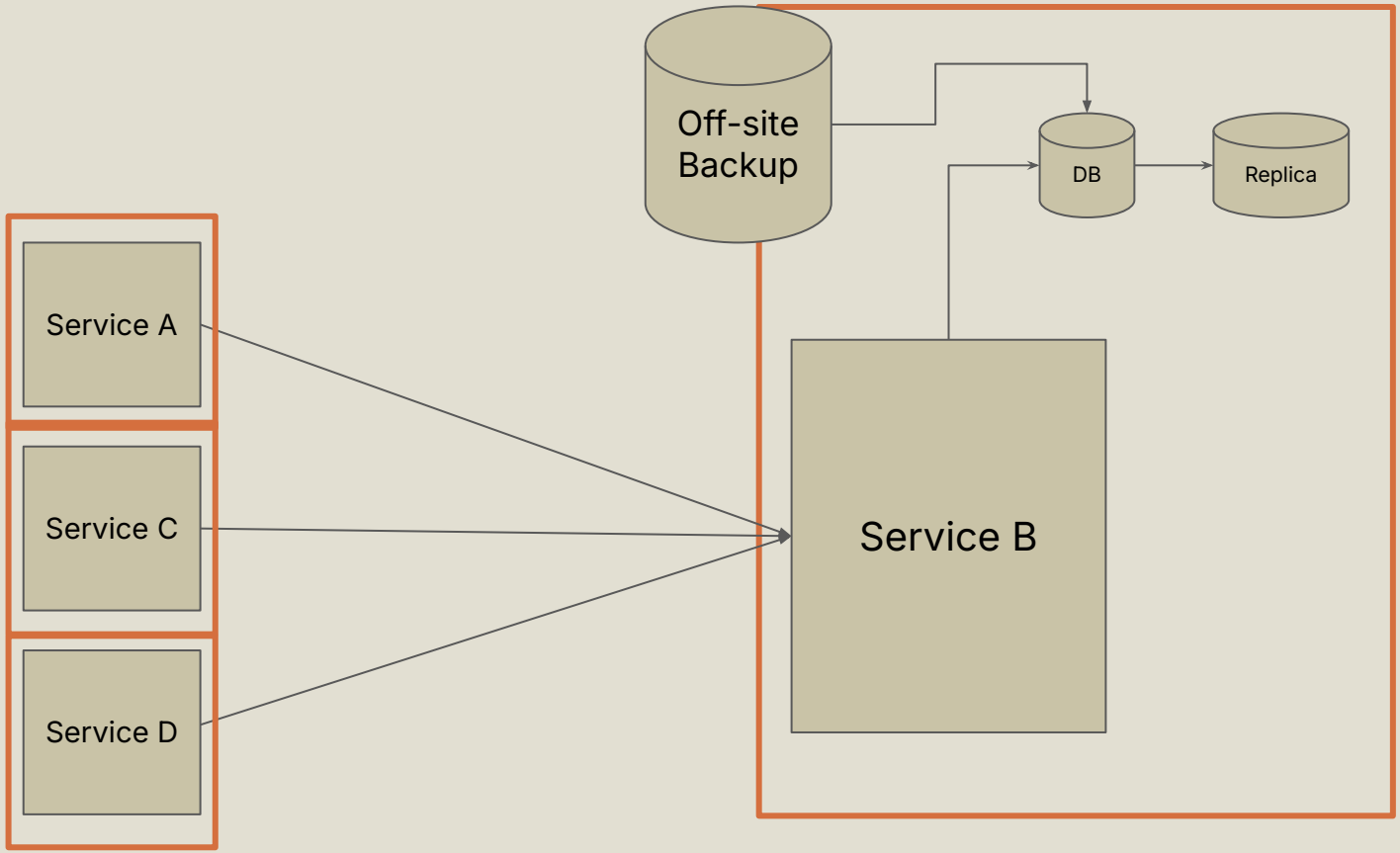


Objective 3: Verify Fallback Behaviour

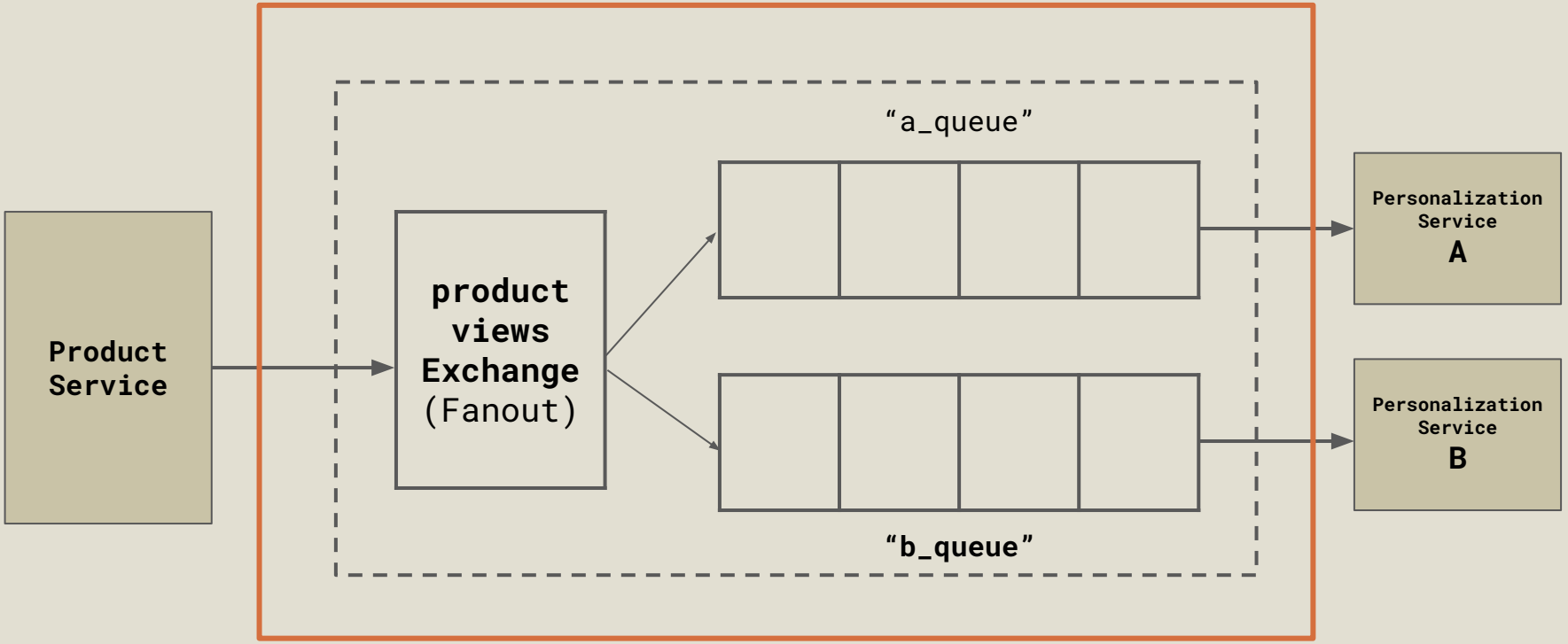
Business Objectives?

https://www.metricfire.com/blog/grafana-vs-powerbi-using-grafana-for-business-metrics/

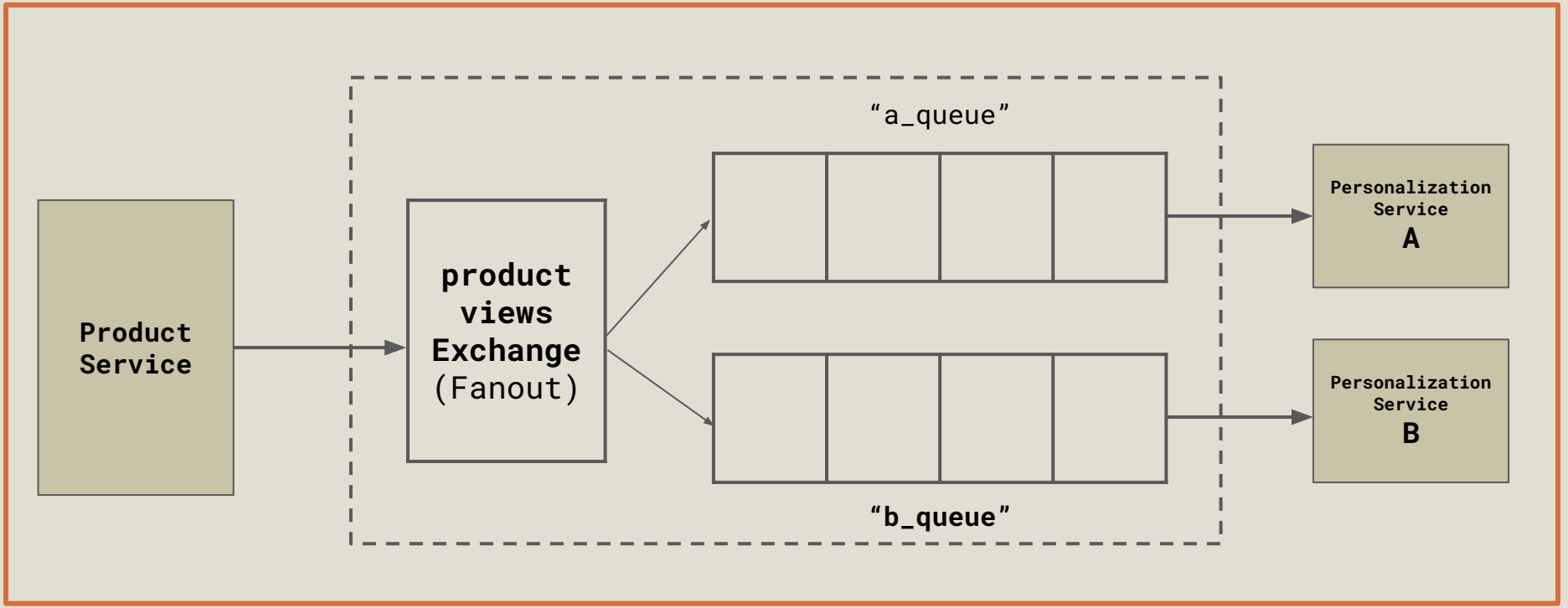
6.2.1 testing patterns



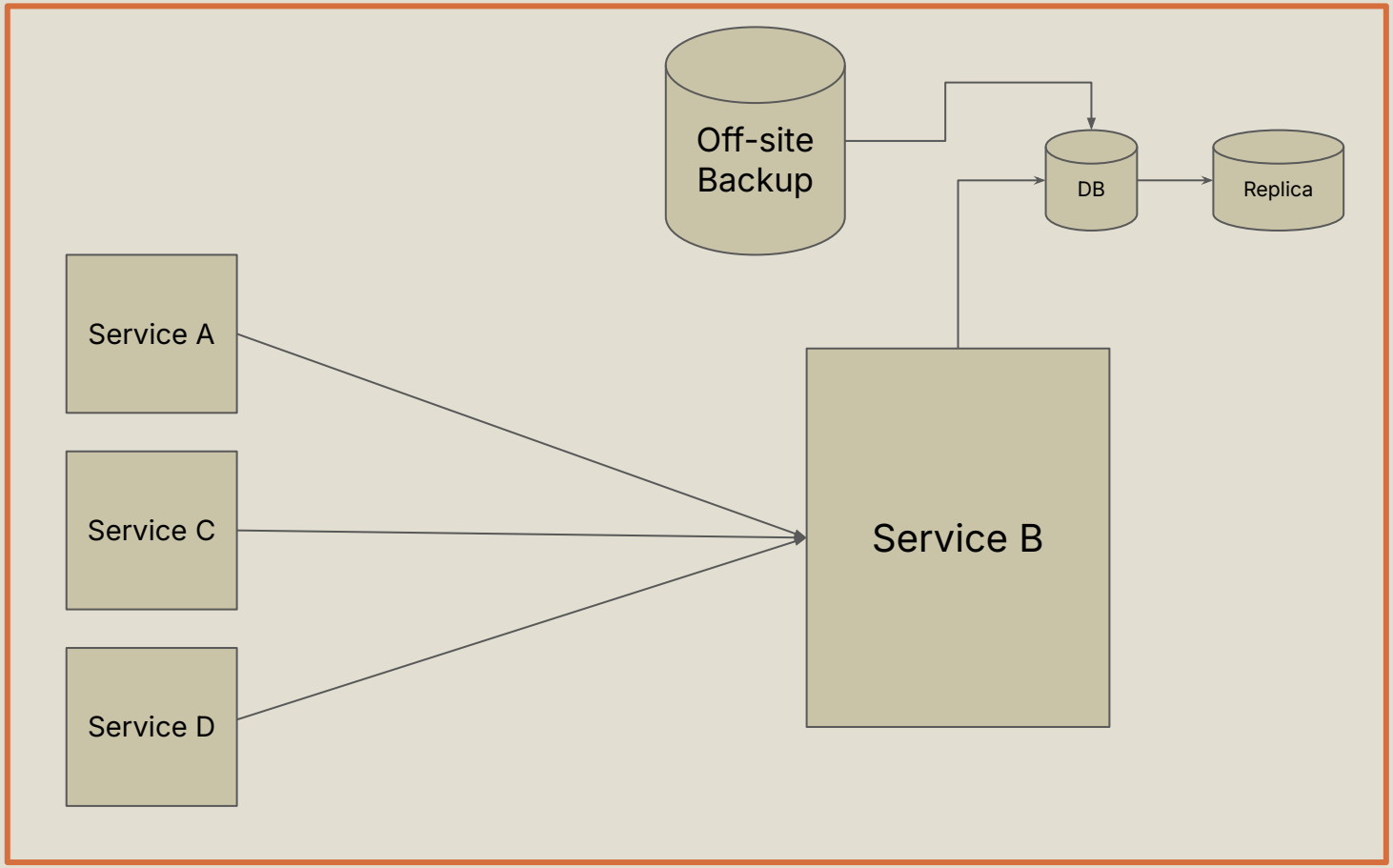
Isolated Service Testing



Isolated Tool Testing



Integrated Tool Testing



Full Service Testing

Method	Description	Examples
Constant Load	Test the system with a constant amount of traffic.	X rps to measure average, median, and 95th percentile response times
Step Load	Increase the traffic to a system by X% at fixed periods of time	Start with X rps, and go to 1.25x, 1.5x, 2x, and so on.
Spike Testing	Drastically increase the amount of traffic	Start with X rps, than go to 3 times X or even more
Soak Testing	Test the system for an extended duration	Run tests for 6 hours or more, and check for memory leaks, disk usage, etc.

Load Testing Methods

6.2.2 testing tools

+

Load Testing + Observability

Code-Based Load Testing


```
import http from 'k6/http';
import { sleep } from 'k6';

export const options = {
  iterations: 10,
};

export default function () {
  // Make a GET request to the target URL
  http.get('https://quickpizza.grafana.com');

  // Sleep for 1 second to simulate real-world usage
  sleep(1);
}
```

k6: Reliability Testing Tool in JS/TS

https://grafana.com/docs/k6/latest/get-started/running-k6/

k6-tests — -bash — 142×33

```
[k6-tests: k6 run script.js                                                    ]

          /\      |‾‾| /‾‾/   /‾/
     /\  /  \     |  |/  /   / /
    /  \/    \    |     (   / ‾‾\
   /          \   |  |\  \ |  (‾)  |
  / __________ \  |__| \__\ \_____/ .io

  execution: local
     output: -
     script: script.js

    duration: 1m0s, iterations: -
         vus: 100,  max: 100

    done [==========================================================] 1m0s / 1m0s

    data_received..............: 148 MB 2.5 MB/s
    data_sent..................: 1.0 MB 17 kB/s
    http_req_blocked...........: avg=1.92ms   min=1µs      med=5µs      max=288.73ms p(90)=11µs     p(95)=17µs
    http_req_connecting........: avg=1.01ms   min=0s       med=0s       max=166.44ms p(90)=0s       p(95)=0s
    http_req_duration..........: avg=143.14ms min=112.87ms med=136.03ms max=1.18s    p(90)=164.2ms  p(95)=177.75ms
    http_req_receiving.........: avg=5.53ms   min=49µs     med=2.11ms   max=1.01s    p(90)=9.25ms   p(95)=11.8ms
    http_req_sending...........: avg=30.01µs  min=7µs      med=24µs     max=1.89ms   p(90)=48µs     p(95)=63µs
    http_req_tls_handshaking...: avg=0s       min=0s       med=0s       max=0s       p(90)=0s       p(95)=0s
    http_req_waiting...........: avg=137.57ms min=111.44ms med=132.59ms max=589.4ms  p(90)=159.95ms p(95)=169.41ms
    http_reqs..................: 13491  224.848869/s
    iteration_duration.........: avg=445.48ms min=413.05ms med=436.36ms max=1.48s    p(90)=464.94ms p(95)=479.66ms
    iterations.................: 13410  223.498876/s
    vus........................: 100    min=100 max=100
    vus_max....................: 100    min=100 max=100

k6-tests:
```

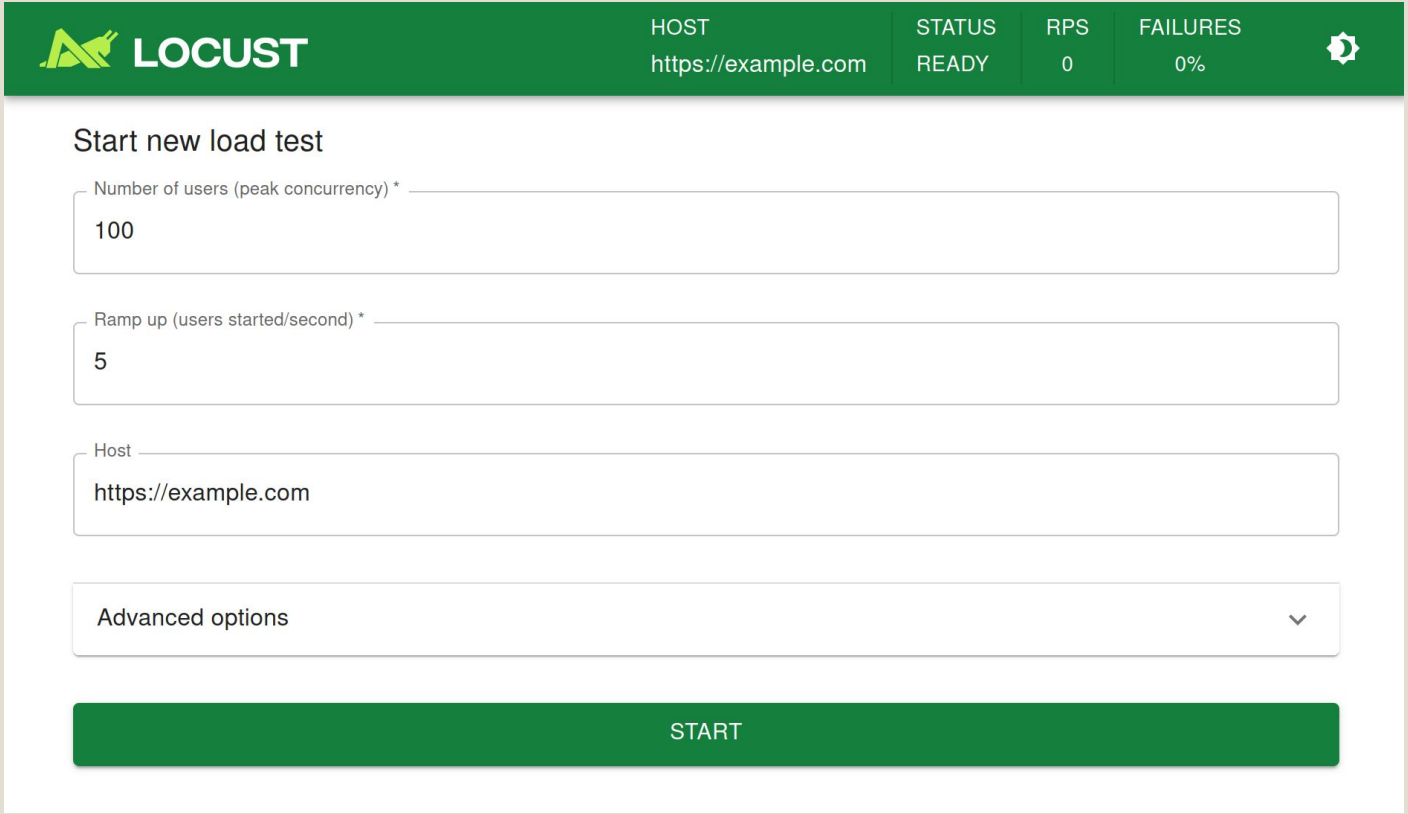
k6: Result Output or Export

```
from locust import HttpUser, task

class HelloWorldUser(HttpUser):
    @task
    def hello_world(self):
        self.client.get("/hello")
        self.client.get("/world")
```

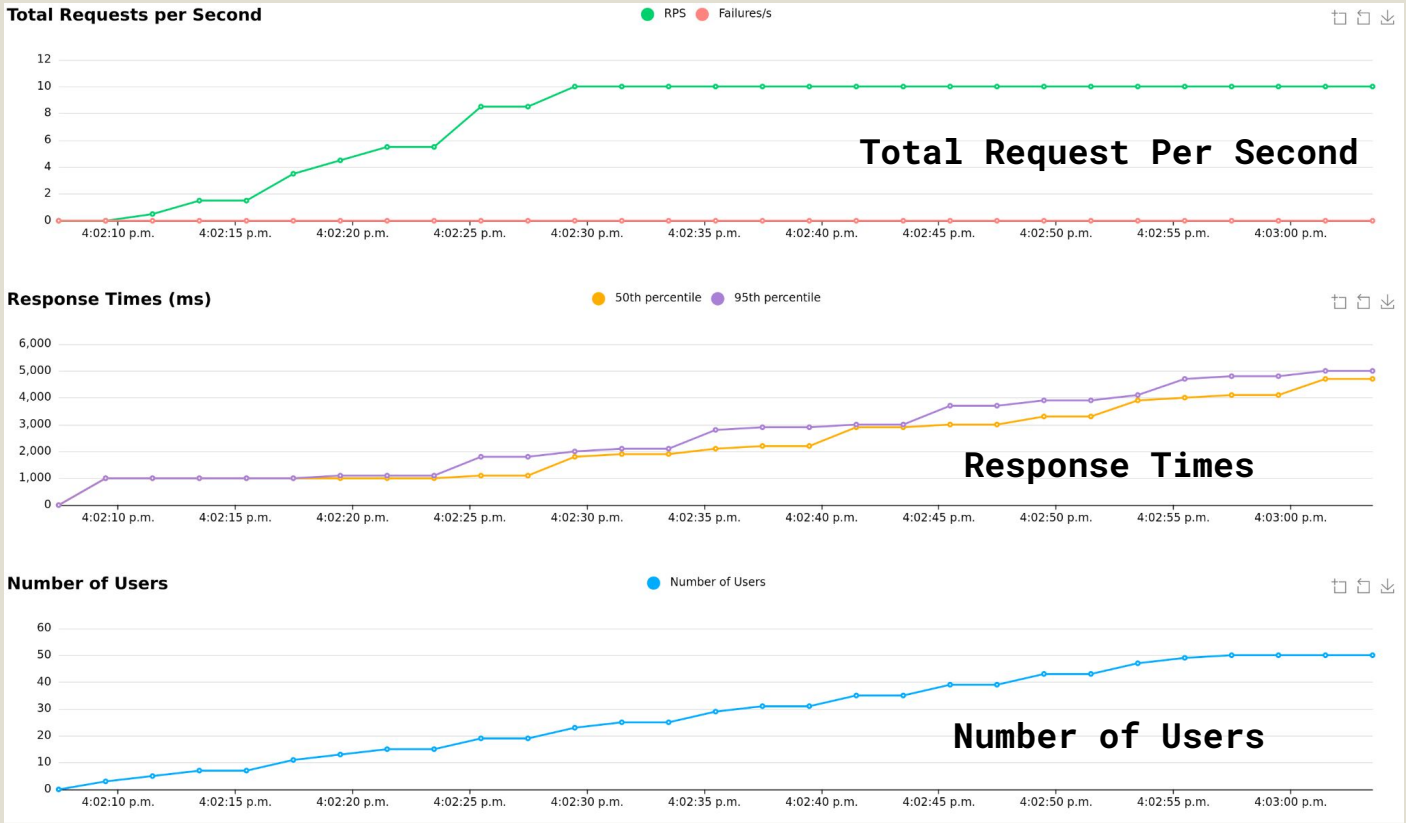
Locust: Performance Load Testing in Python

https://docs.locust.io/en/stable/quickstart.html



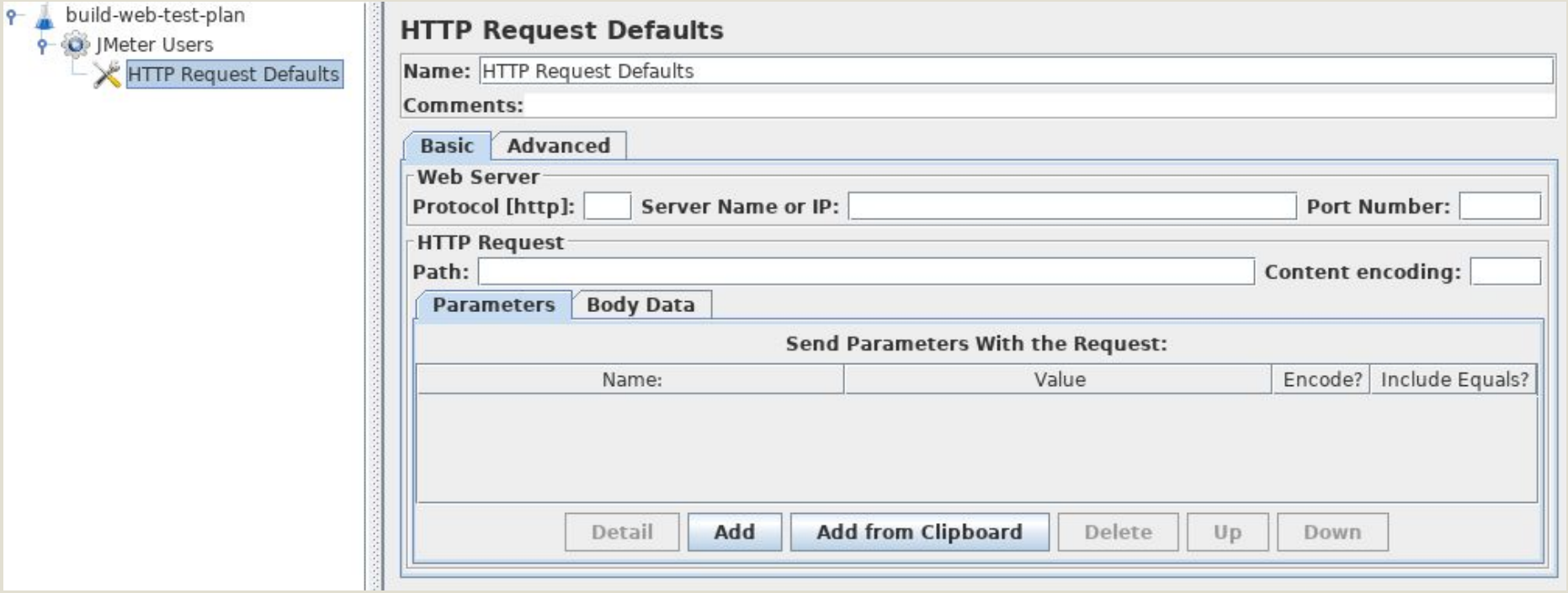
Locust: Performance Load Testing in Python

https://docs.locust.io/en/stable/quickstart.html

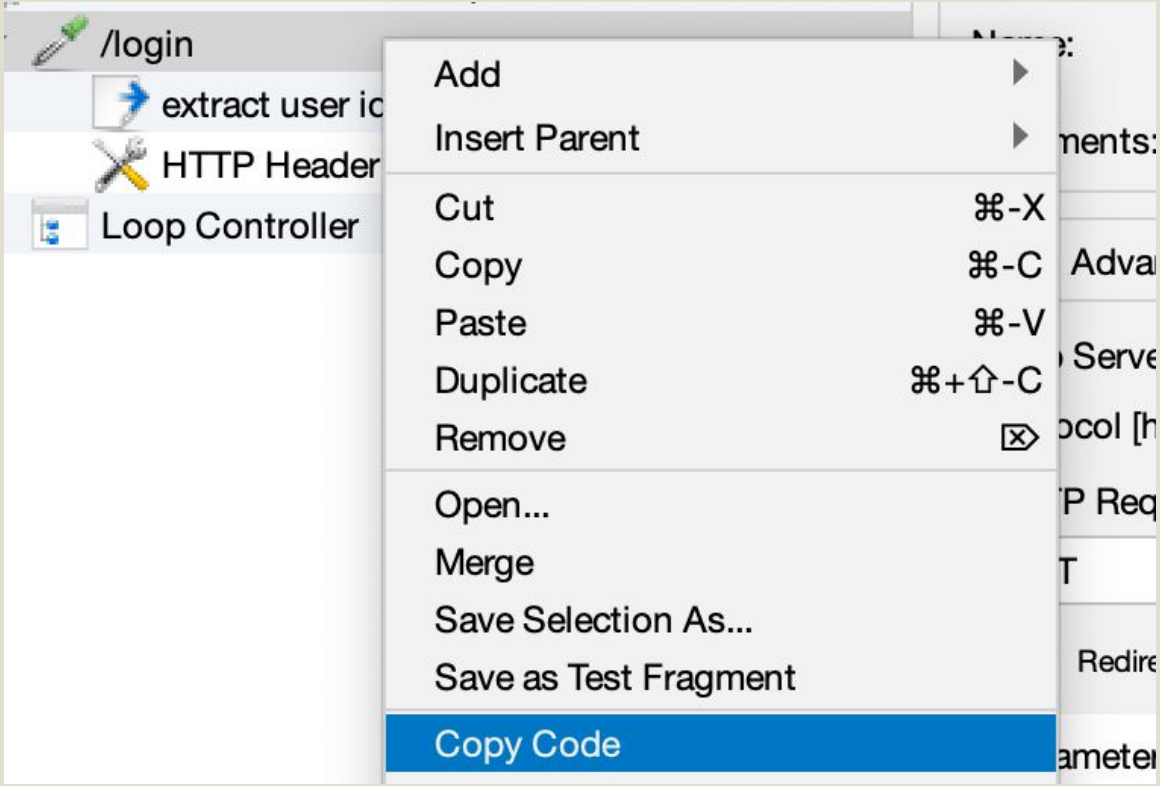


Locust: Performance Load Testing in Python

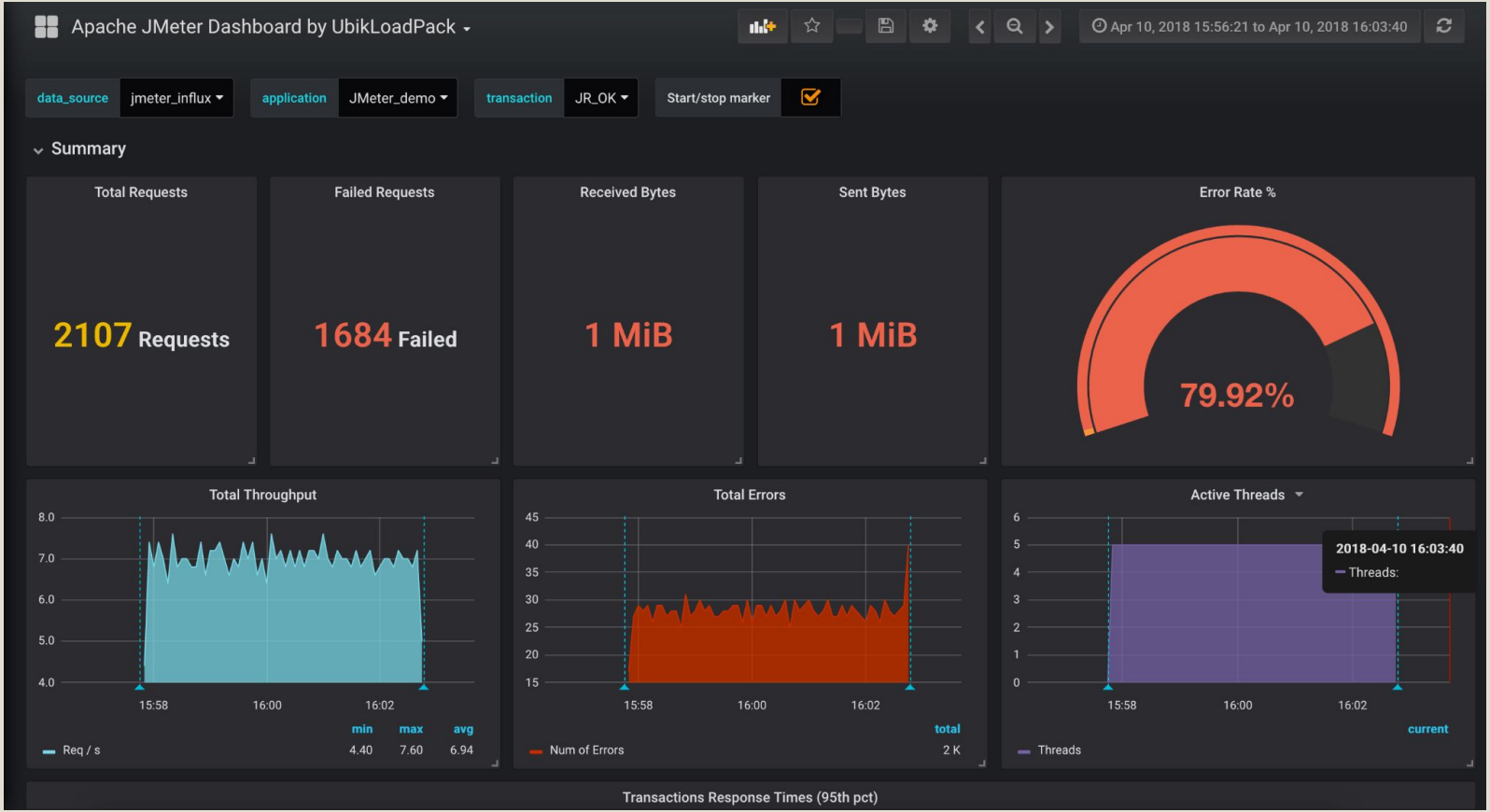
https://docs.locust.io/en/stable/quickstart.html



JMeter: Build Test Plan with UI



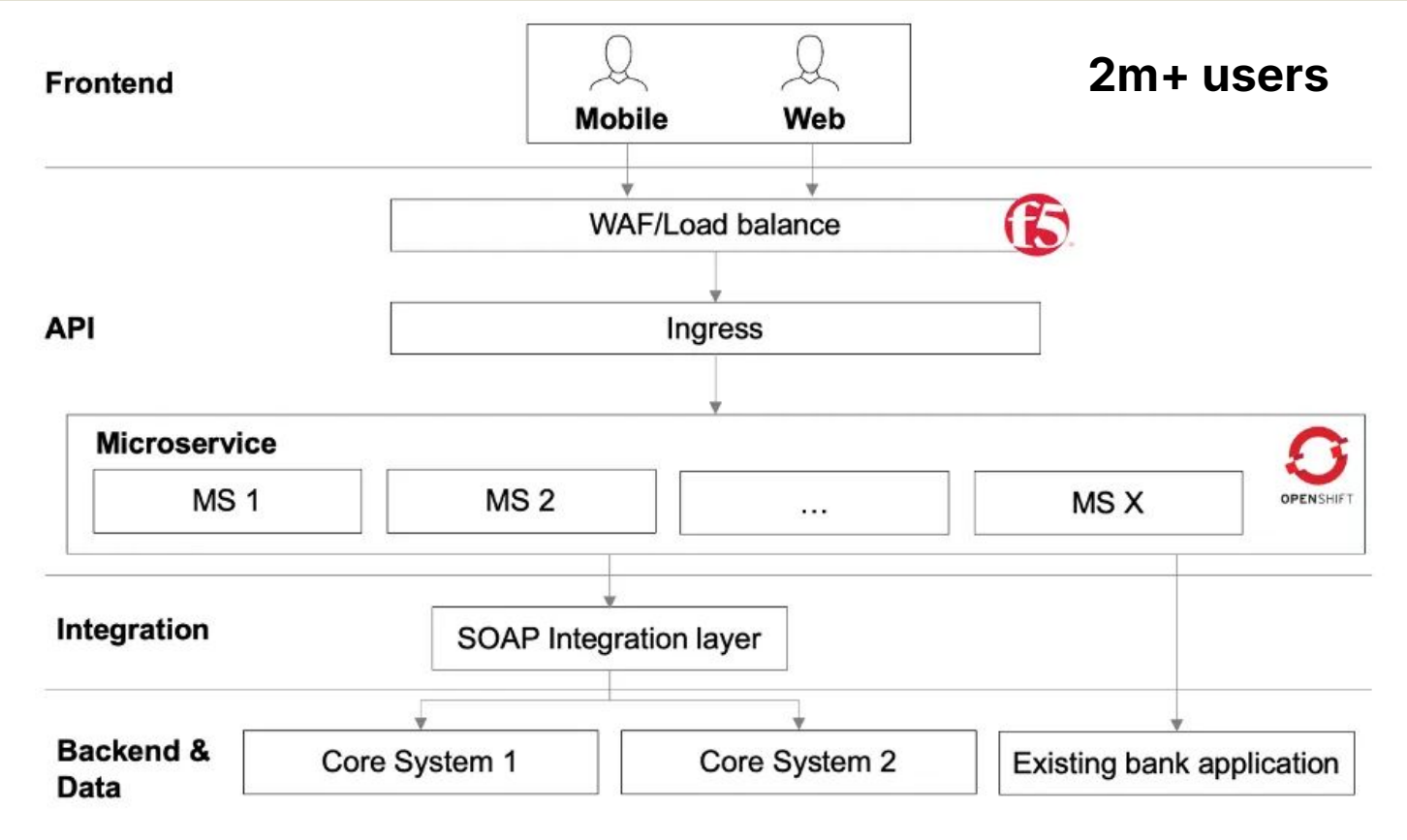
JMeter: Build Test Plan with UI -> Code



Load Testing Results with Grafana and Prometheus

https://www.ubik-ingenierie.com/blog/why-you-should-upgrade-to-jmeter-5-0/

6.2.3 case study



Case study on performance testing of a large-scale application

https://medium.com/@mohammad.assaf/case-study-on-performance-testing-of-a-large-scale-application-ca7a4e5f0c51

Load testing (Constant)

- Concurrent users: 6,000
- Duration: 1 hour

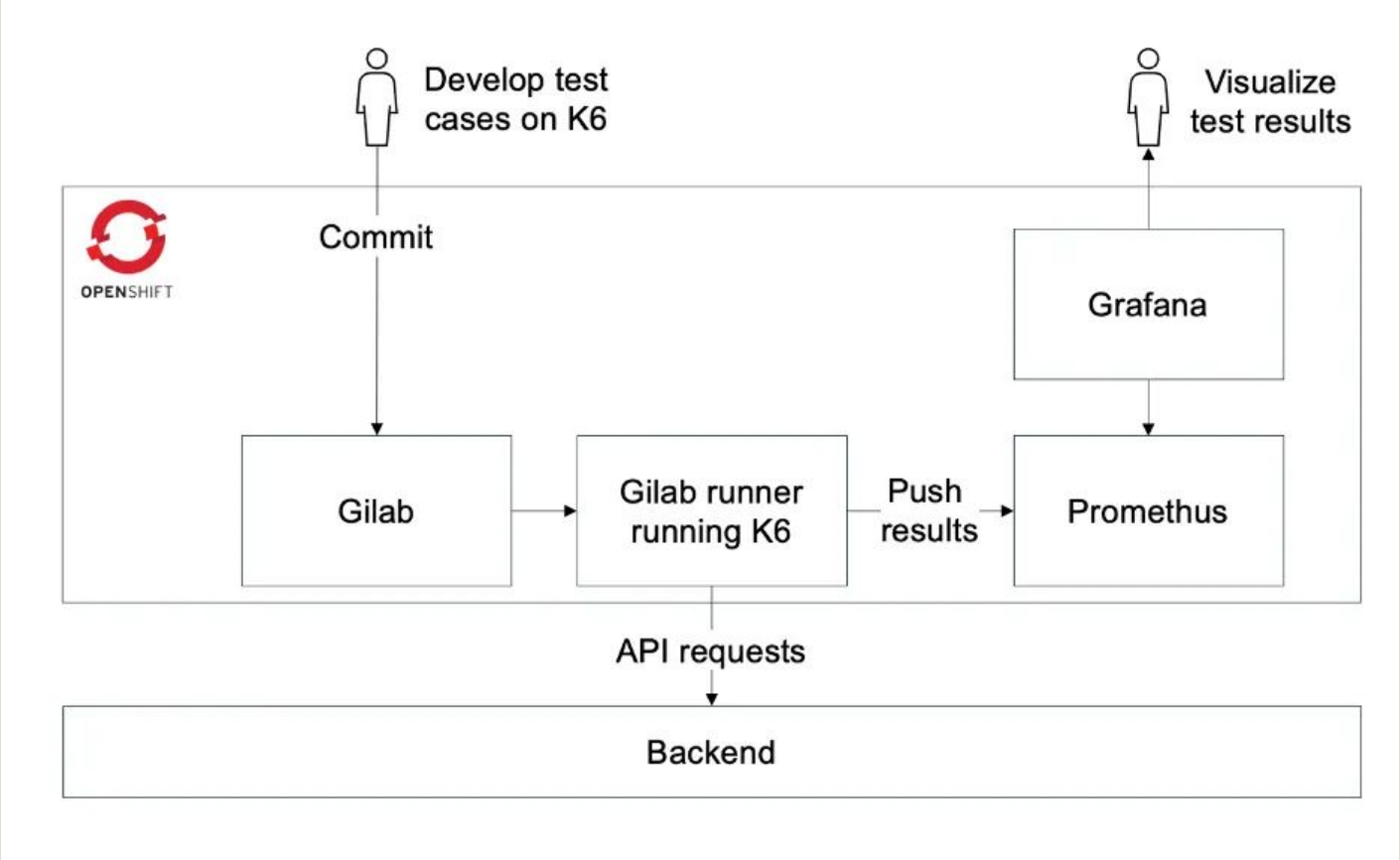
Endurance testing (Soak testing)

- Concurrent users: 3,000
- Duration: 12 hours

Tool used: K6 load testing (https://k6.io)

Case study on performance testing of a large-scale application

https://medium.com/@mohammad.assaf/case-study-on-performance-testing-of-a-large-scale-application-ca7a4e5f0c51



Gitlab CI runs K6, with metrics captured on Prometheus

https://medium.com/@mohammad.assaf/case-study-on-performance-testing-of-a-large-scale-application-ca7a4e5f0c51

Grafana used to chart load testing metrics

https://medium.com/@mohammad.assaf/case-study-on-performance-testing-of-a-large-scale-application-ca7a4e5f0c51

DB Response Time Spikes due to Connection Pool Exhaustion

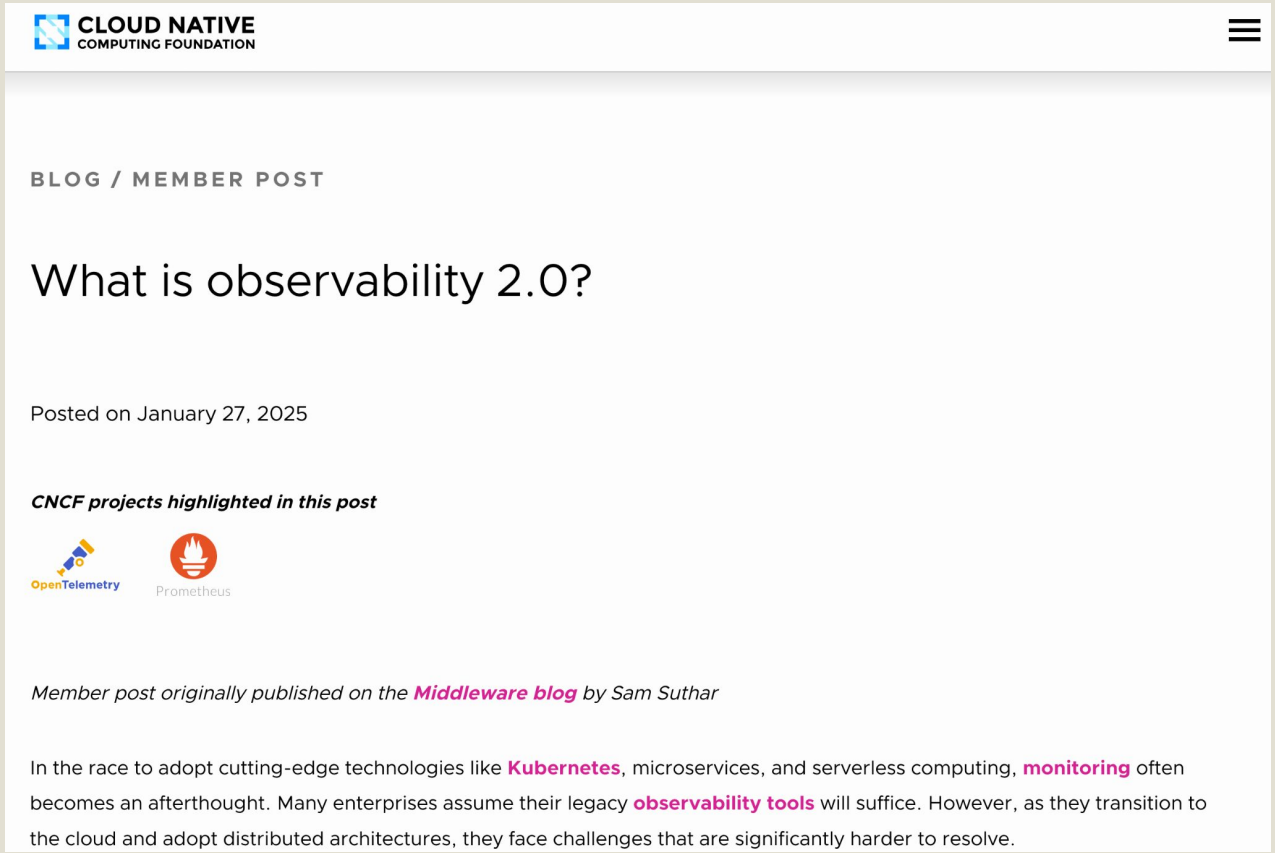
https://medium.com/@mohammad.assaf/case-study-on-performance-testing-of-a-large-scale-application-ca7a4e5f0c51

Challenge	Impact	Solution
Extra load on downstream systems	Sluggish API performance and user having to unnecessarily wait for pages to load	Follow a lazy information fetching approach, only get information when they are absolutely needed
Bottleneck in connection pools between systems (such as HTTPS and databases)	System reaching a specific number of concurrent users then spiking response times and timing-out	Fine tune connection pools (such as database configs) and keep-alive HTTPS connections whenever possible
Difficulty in identifying non-performant code	More time and resources are needed to debug performance issues and enhance application code	Implement tracing, visualized on appropriate tools (such as Kibana) which is critical to measure execution times on granular levels
Setup environment resources and configuration to be production-like	System not able to scale to production load, with insufficient physical resources or faulty software configurations	Inspect physical resources of all systems in the stack and cross-check configuration with production
Prepare testing data that are production-like	Errors and crashes during the performance tests due to users not fit for the journeys, or faulty test results due to bias in the data (for example users in the same category)	Get user data following well-defined criteria for test journeys, and mimic production statics in terms of user categories

Summary of Findings

https://medium.com/@mohammad.assaf/case-study-on-performance-testing-of-a-large-scale-application-ca7a4e5f0c51

p.s.
Current Trends



What is Observability 2.0?

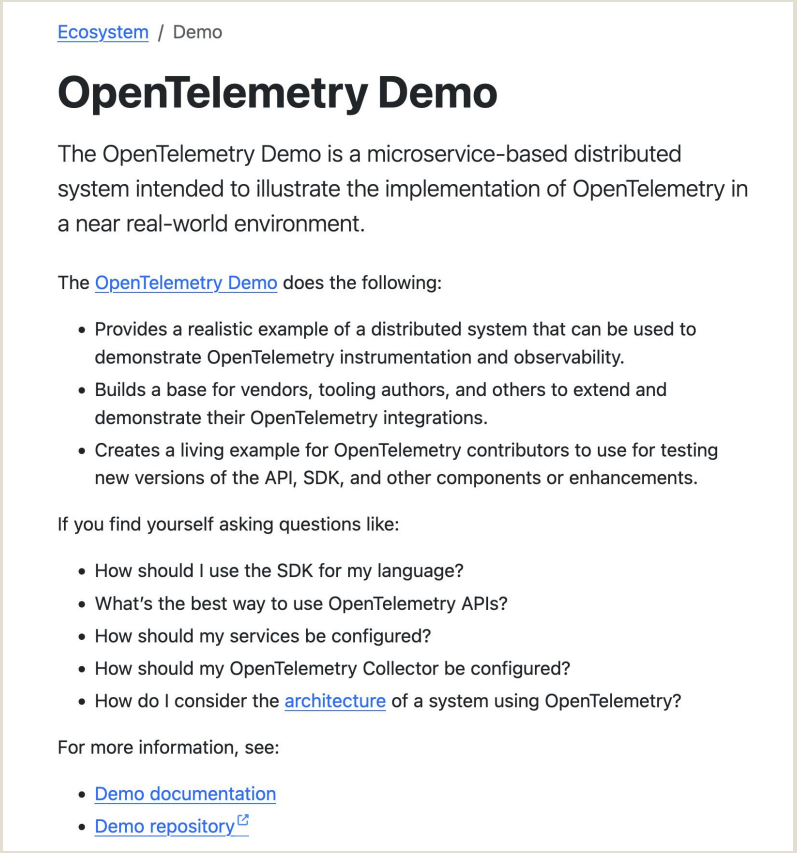
https://www.cncf.io/blog/2025/01/27/what-is-observability-2-0/

1. **Data handling**:
    - Traditional: Relies on separate tools for metrics, logs, and traces, creating silos and requiring manual correlation.
    - 2.0: Unifies telemetry data into a single platform, offering a comprehensive view of system health.
2. **Problem detection**:
    - Traditional Uses static thresholds and alerts that are often reactive and miss subtle issues.
    - 2.0: Employs AI and machine learning to identify anomalies in real-time, enabling proactive issue resolution.
3. **Focus on context**:
    - Traditional: Provides raw technical data without linking it to broader business outcomes.
    - 2.0: Maps telemetry data to business metrics, ensuring decisions align with organizational goals.
4. **Scalability and adaptability**:
    - Traditional Struggles with dynamic environments like Kubernetes and serverless, often requiring custom setups.
    - 2.0: Designed for dynamic scaling, adapts with ease to changes in cloud-native architectures.

Key differences between traditional observability and observability 2.0

https://www.cncf.io/blog/2025/01/27/what-is-observability-2-0/

optional readings



Ecosystem / Demo

OpenTelemetry Demo

The OpenTelemetry Demo is a microservice-based distributed system intended to illustrate the implementation of OpenTelemetry in a near real-world environment.

The OpenTelemetry Demo does the following:

- Provides a realistic example of a distributed system that can be used to demonstrate OpenTelemetry instrumentation and observability.
- Builds a base for vendors, tooling authors, and others to extend and demonstrate their OpenTelemetry integrations.
- Creates a living example for OpenTelemetry contributors to use for testing new versions of the API, SDK, and other components or enhancements.

If you find yourself asking questions like:

- How should I use the SDK for my language?
- What's the best way to use OpenTelemetry APIs?
- How should my services be configured?
- How should my OpenTelemetry Collector be configured?
- How do I consider the architecture of a system using OpenTelemetry?

For more information, see:

- Demo documentation
- Demo repository

The OpenTelemetry Demo does the following:

- Provides a realistic example of a distributed system that can be used to demonstrate OpenTelemetry instrumentation and observability.
- Builds a base for vendors, tooling authors, and others to extend and demonstrate their OpenTelemetry integrations.
- Creates a living example for OpenTelemetry contributors to use for testing new versions of the API, SDK, and other components or enhancements.

OpenTelemetry Demo

https://opentelemetry.io/ecosystem/demo/

Using Metrics and Traces to diagnose a memory leak

Application telemetry, such as the kind that OpenTelemetry can provide, is very useful for diagnosing issues in a distributed system. In this scenario, we will walk through a scenario demonstrating how to move from high-level metrics and traces to determine the cause of a memory leak.

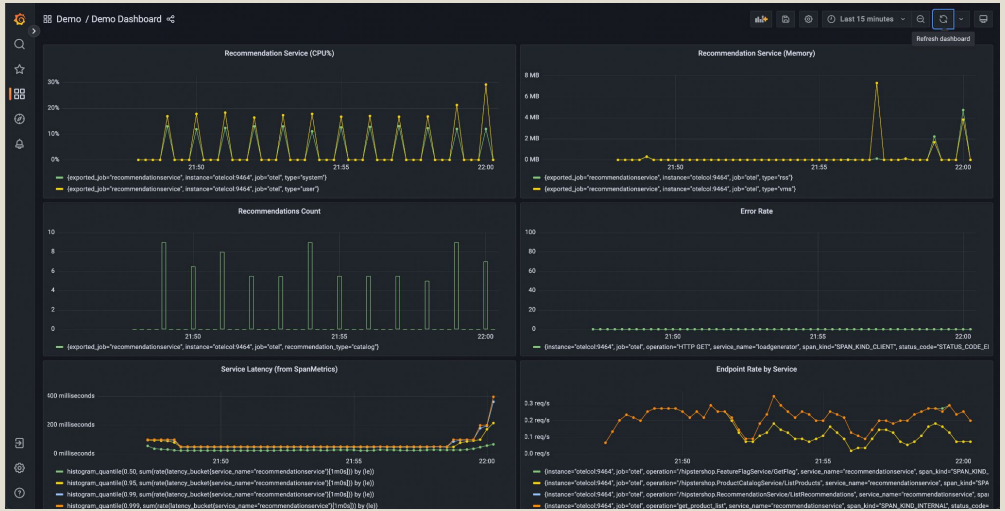
Setup

To run this scenario, you will need to deploy the demo application and enable the `recommendationServiceCacheFailure` feature flag. Let the application run for about 10 minutes or so after enabling the feature flag to allow for data to populate.

Diagnosis

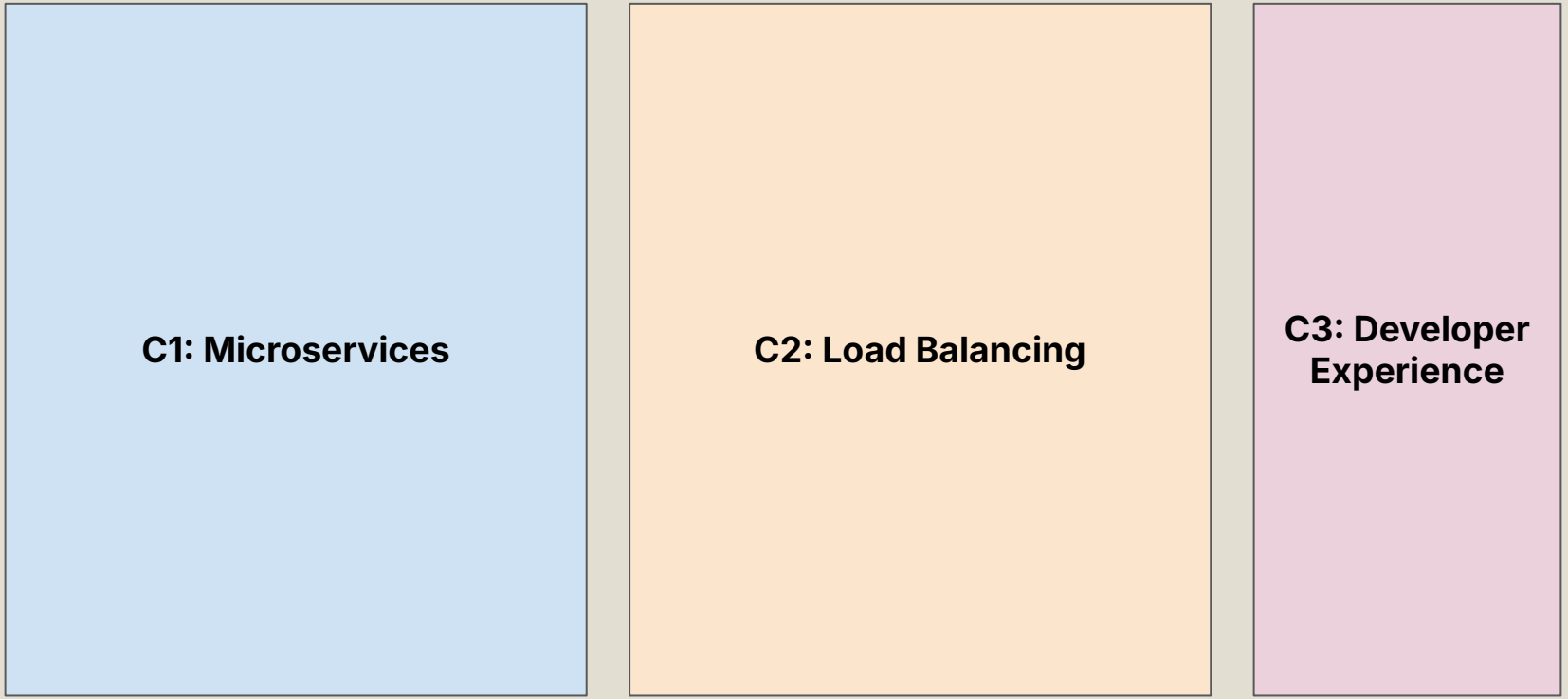
The first step in diagnosing a problem is to determine that a problem exists. Often the first stop will be a metrics dashboard provided by a tool such as Grafana.

A demo dashboard folder should exist after launching the demo with two dashboards; One is to monitor your OpenTelemetry Collector, and the other contains several queries and charts to analyze latency and request rate from each service.



Scenario: Using Metrics and Traces to diagnose a memory leak

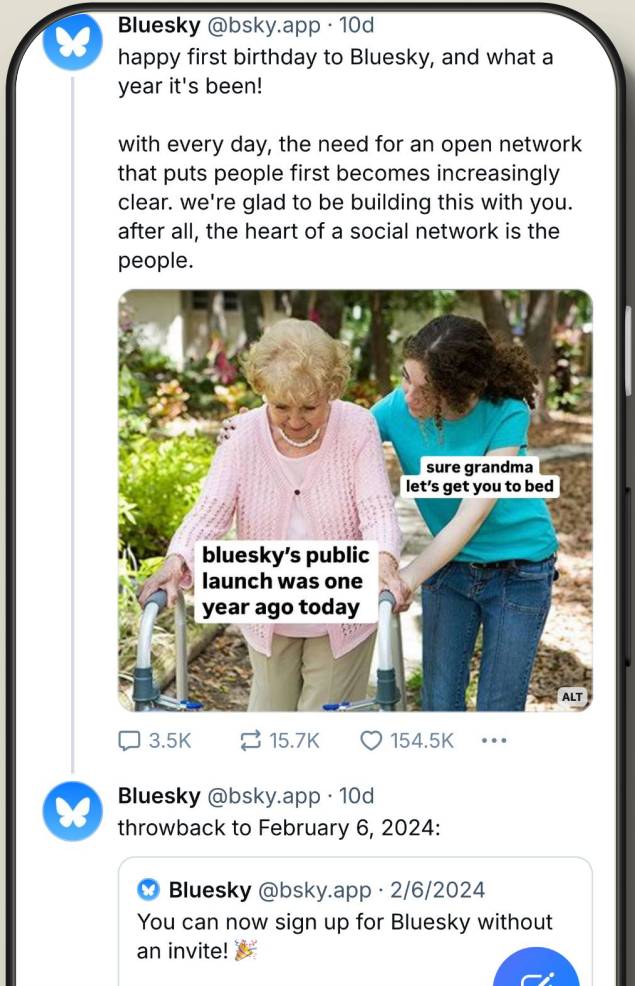
https://opentelemetry.io/docs/demo/scenarios/recommendation-cache/

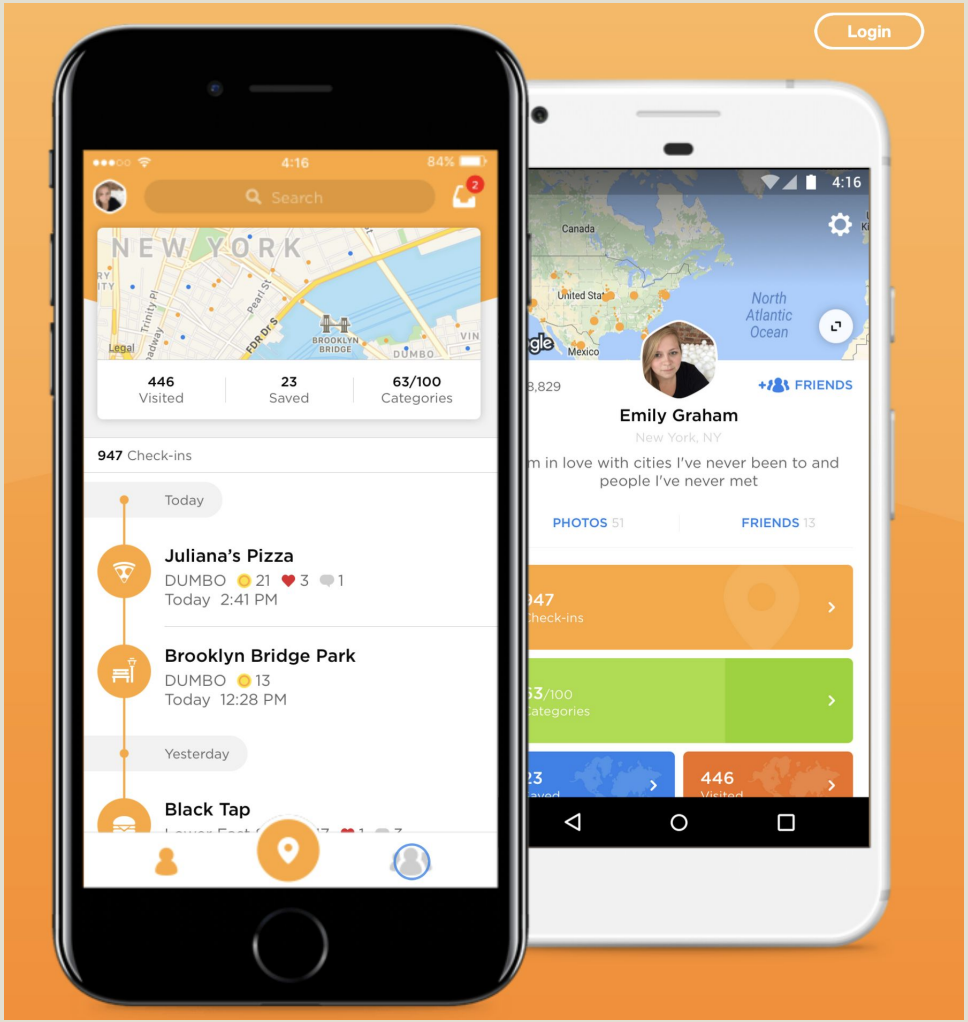


W7: 3 Case Studies

Assignment

Extend a Distributed Social Media Platform




Login
4:16
84%
Search
NEW YORK
446
Visited
23
Saved
63/100
Categories
947 Check-ins
Today
Juliana's Pizza
DUMBO 21 3 1
Today 2:41 PM
Brooklyn Bridge Park
DUMBO 13
Today 12:28 PM
Yesterday
Black Tap
4:16
Canada
North Atlantic Ocean
Mexico
FRIENDS
Emily Graham
New York, NY
PHOTOS 51
FRIENDS 13
446

Assignment

W1 → Distributed Social Media Research
W2 → Review W1 + Build PoC
W3 → Design 1st Draft of System + Sharing
W4 → Implementation
W5 → Implementation (Queues)
W6 → Implementation (Load Testing / Observability)
W7 → Technical Retrospective
W8 → Complete System Implementation

Assignment #5

- Choose one:
  (i) Implement observability in your project (metrics, logs, and traces), or
  (ii) Design a test plan and perform actual load testing on your system (isolated, full, or somewhere in between)

- Share your implementation in code, and share a link to your project (eg. GitHub or Gitlab)