



Cornell University
Computer Systems Laboratory



Leveraging Machine Learning to Detect Performance Issues in Cloud Applications

Meghna Pancholi, Yuan He, Siyuan Hu

1. Motivation

Why and how to predict performance issues in cloud applications.

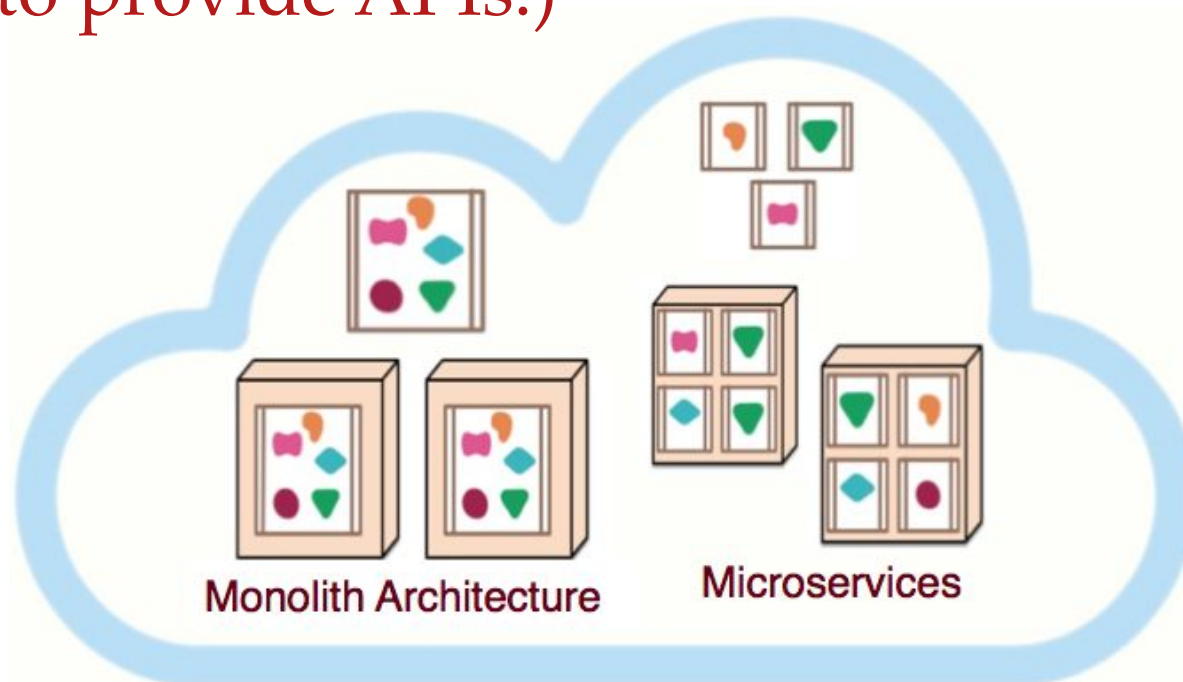
2. Solution

How we approach the problem by obtaining **end-to-end tracing, performance measurements, resource allocation configuration**, and implement **deep learning** on top of the collected data to predict QoS violations in the near future.

MONOLITHIC VS. MICROSERVICES



- **Monolithic applications:** Built as a single unit
- **Microservices applications:** Broken into several *loosely-coupled* self-contained components that have *bounded contexts*. (Easier to update and deploy, only need to provide APIs.)

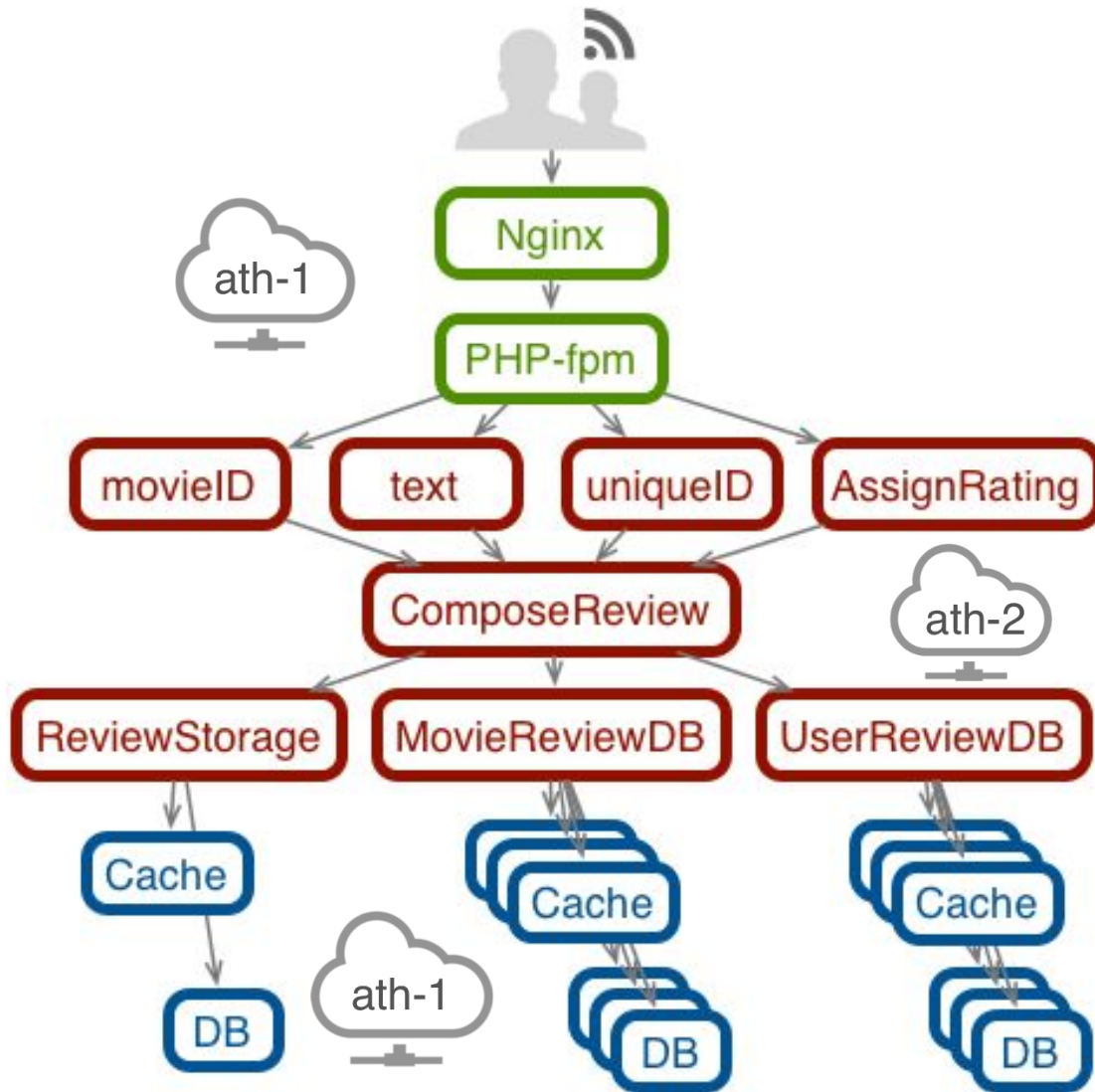


- **Difficulty:** Different languages, spanning multiple servers, large volume of requests, separate DBs. Need to trace the **evolution of a request through an application's life cycle** and monitor performance.
- **Solution:** Distributed Tracing System(*Track requests as they pass through different microservices*)
- **Goal:** Use distributed tracing to predict performance (QoS violations) or resource saturation issues before they actually occur.

- Cloud apps are governed by strict **QoS constraints** in terms of throughput and **tail latency**, availability and reliability.
- Vital to monitor the cloud application with tracing to **troubleshoot the services that are possible to have a QoS violation**.
- Ultimately, we want to be able to **dynamically re-allocate resources** to prevent QoS violations.

- Understand the **interaction between microservices** and end-to-end applications.
- Design and use a **distributed end-to-end tracing** system to collect performance and utilization statistics.
- Use the tracing data to **anticipate QoS violations** before they could occur with a ML model.

SYSTEM ARCHITECTURE: MOVIE STREAMING SERVICE



NGINX



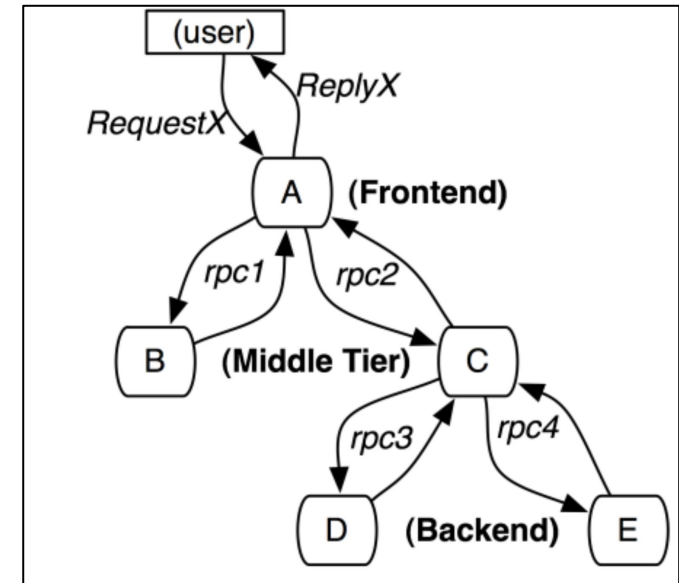
Fig. 1: Compose Movie review

- Popular tracing systems: Zipkin, SpringCloud, OpenTracing
- Gather latency data to determine bottlenecks and troubleshoot performance, efficiency, and security problems
- Create Gantt charts to visualize request latency



■ Dapper

- Google's Large-Scale Distributed Systems Tracing Infrastructure
- Created to understand system behaviors for each request through a distributed system.



Path taken by a request X through a simple system.

RELATED WORK

■ Zipkin

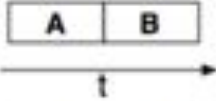
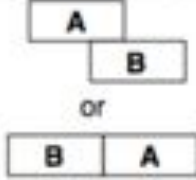
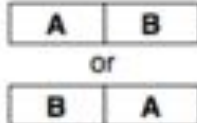
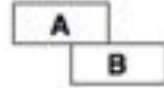
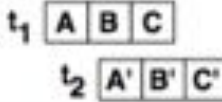
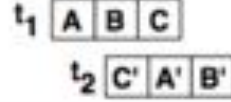
- Distributed Tracing System
- Collects timing data from microservices applications to troubleshoot latency problems and analyze performance.
- Based off of the Google Dapper paper.
- Supports various languages as well as RPC debugging.
- Creates Gantt charts for requests and shows dependency diagrams of how many traced requests went through each application.



■ The Mystery Machine Paper

- End-to-end performance analysis of large-scale Internet Services
- Methodology to interpret performance tracing results of millions of requests through the Facebook Web pipeline.
- Uses this data to develop tools to optimize scheduling and improve latencies of Facebook requests.
- Provides preliminary data about hypothesized methods before implementation.

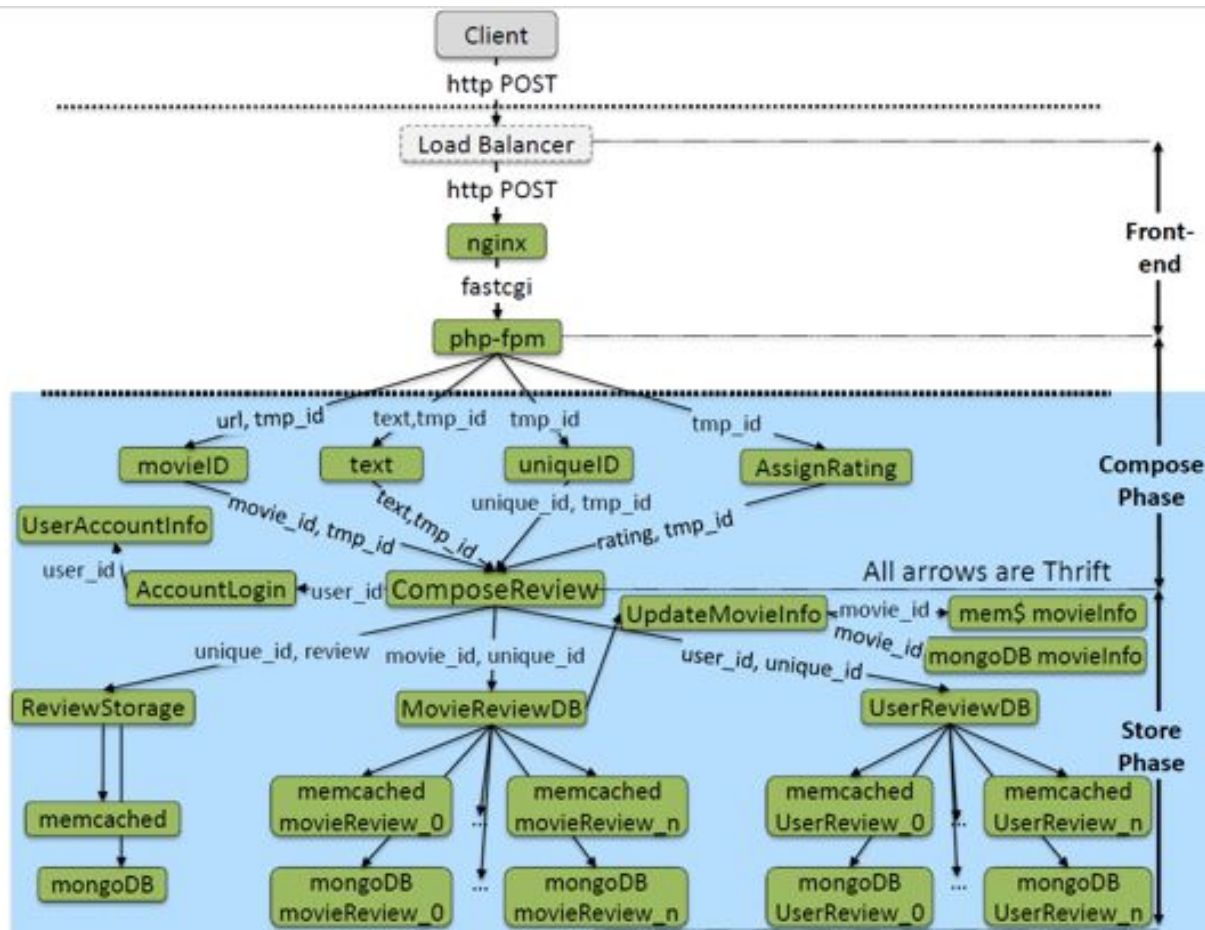
The Mystery Machine Paper

Relationship	Example	Counterexample
Happens Before		
Mutual Exclusion		
Pipeline		

Uses traces to create causal relationship models:

- **Happens-Before** – time stamp for B is greater than or equal to time stamp of A.
- **Mutual Exclusion** – time intervals for A and B do not overlap.
- **Pipeline** – a data dependency between pairs of segments in two tasks t_1 and t_2 .

STEP 1: GANTT TRACING



- Leverage Thrift logger to insert timestamps around each sub-service
- **Compose Review** is a one way request (client to databases)

STEP 1: GANTT TRACING



- Each request receives a **unique identifier**
 - Organizes each request as it passes through the Thrift microservices, MongoDB, Memcached, and the Nginx web server
- Each microservice creates a log with start/end time of its requests
- Logs are interleaved and composed in the end

STEP 1: CHALLENGES



- **Server synchronization:**

- All requests have time stamps from the same logical clock
- Standardizes start and end times from different machines using TimeSync.
 - » $\text{Time Skew} = \text{Time Client} - \text{Time Server}$

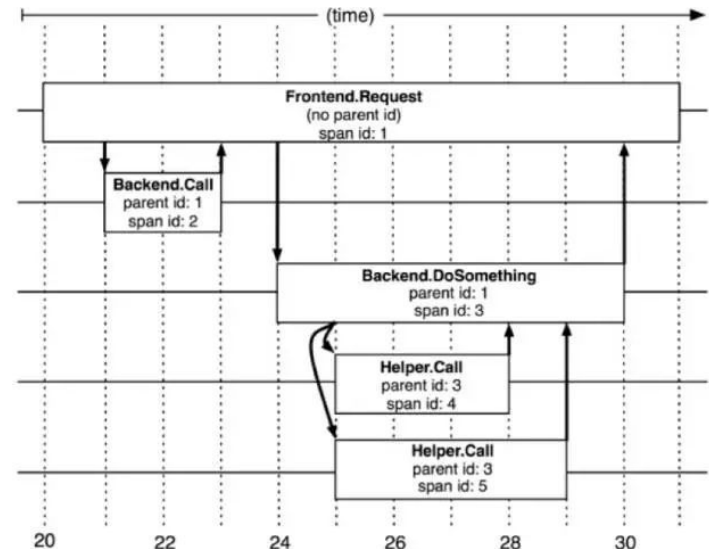
- Allows us to see relationships between requests

- **Not all microservices communicate over RPC (e.g., memcached, mongodb, nginx)**

- Memcached + mongodb don't have a request ID
- Rely on tcpdump to log latency in these services

STEP 1: TRACING METRICS AND ITS VISUALIZATION

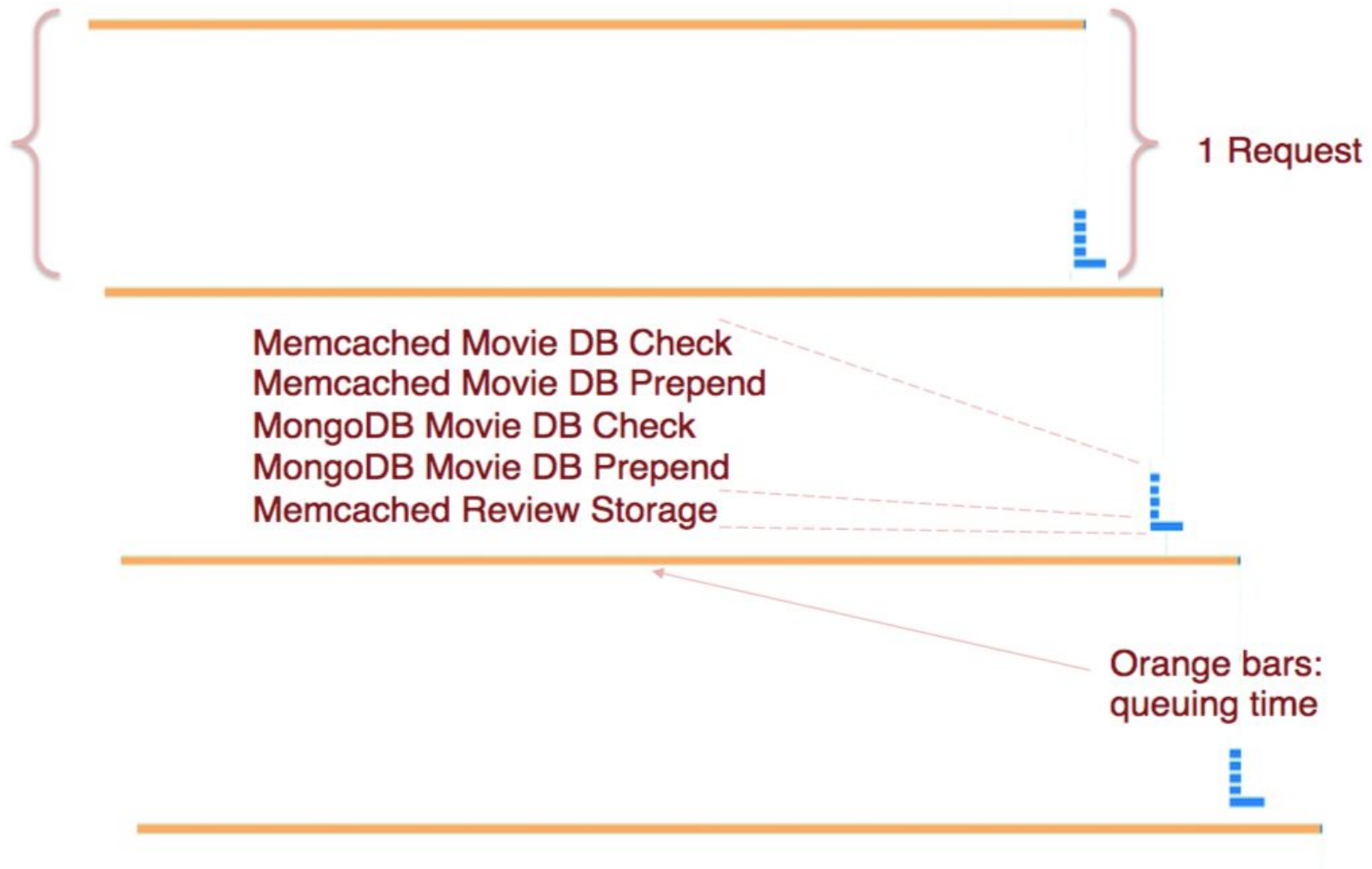
- For each request, start and end times are gathered and processed to create **Gantt chart** visualizations (horizontal bar charts we can use to visualize the requests)
- Gantt charts show:
 - Queueing times
 - Sequence of microservices called
 - Time spent in each microservice
 - Relationships between requests
 - How we may need to modify resource allocations



STEP 1: TRACING NETFLIX COMPOSE REVIEW



of clients = 4, # of requests = 100,000, QPS=5000



STEP 1: NETFLIX COMPOSE REVIEW – INCREASING QPS

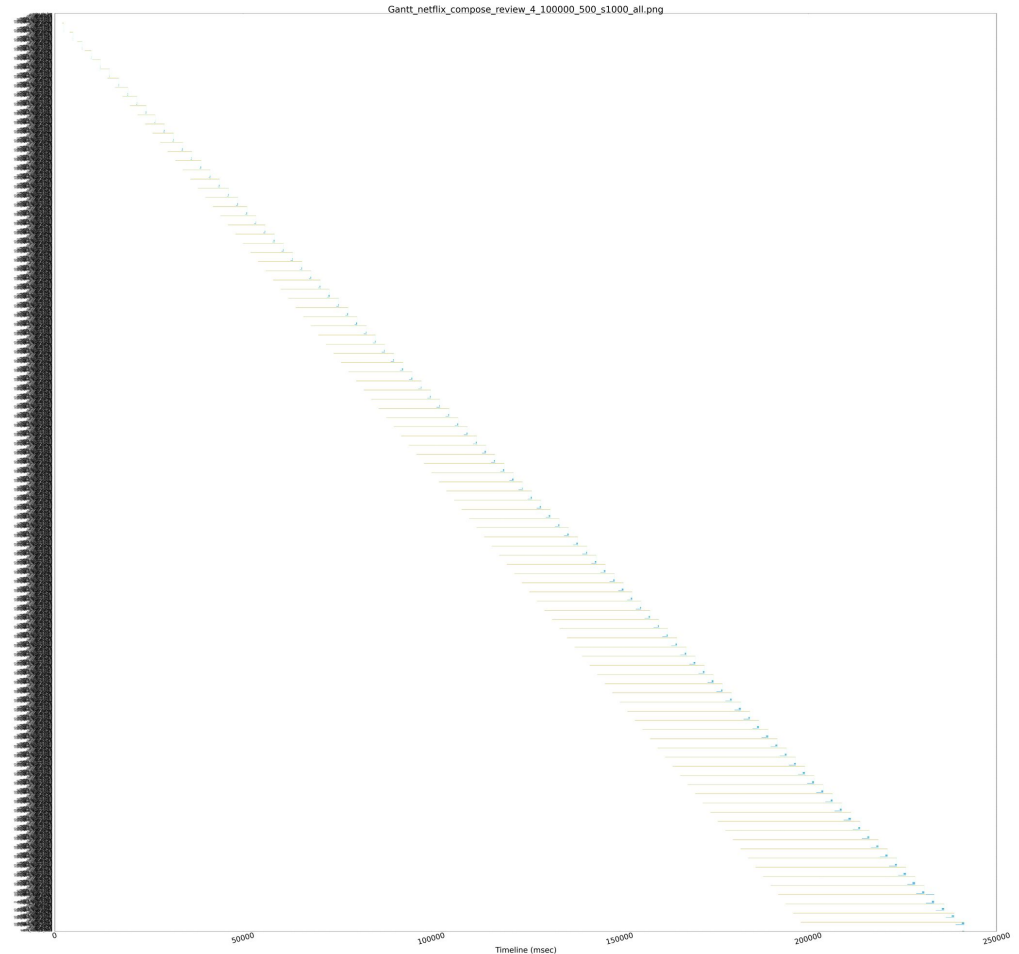
Number of clients: 4

Number of requests: 100,000

QPS: 500

The widening of the orange bars indicates the growth of the queues as the simulator receives more and more requests.

More queries per second also show much larger queues than fewer queries per second.



STEP 1: NETFLIX COMPOSE REVIEW – INCREASING QPS

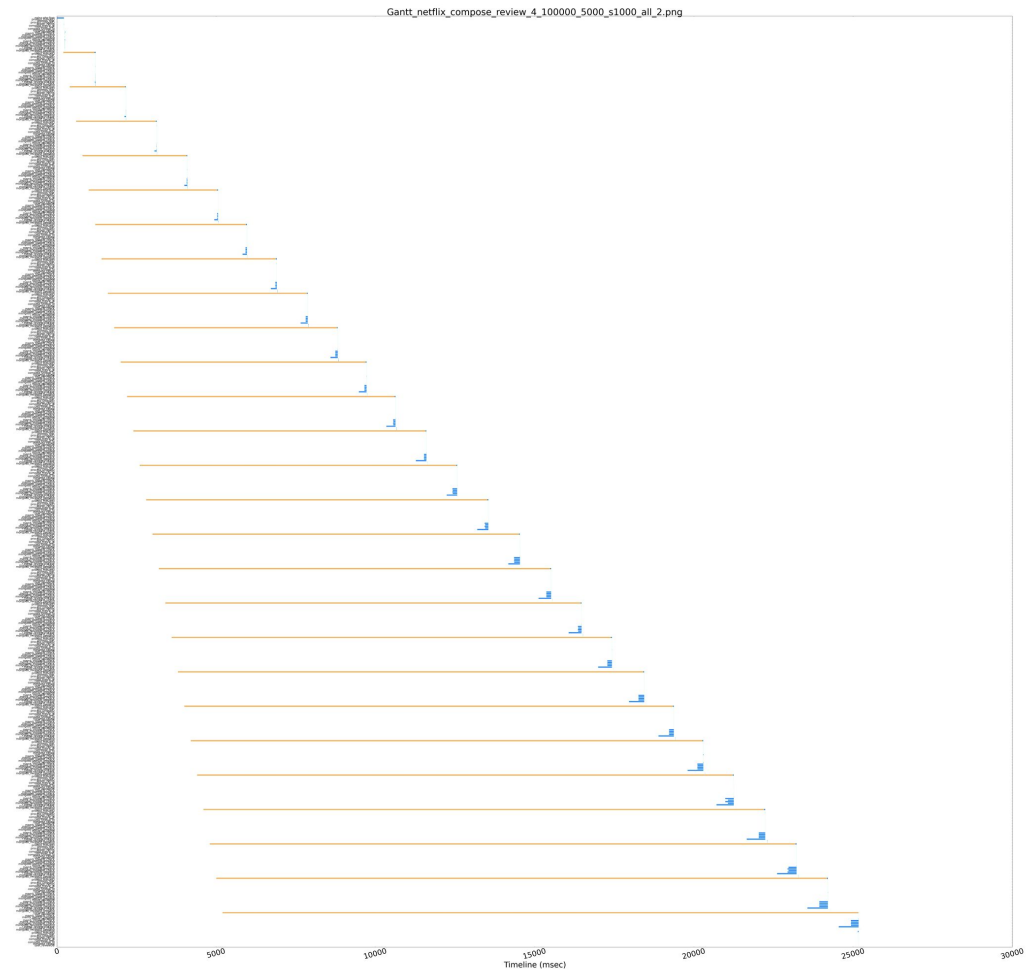
Number of clients: 4

Number of requests: 100,000

QPS: 5,000

Much larger queuing times than smaller queries per second experiments as indicated by long orange bar.

Waterfall format shows relationship between requests.



- As QPS increases, queuing time increases drastically.
- While processing, most of the time is spent in MongoDB and Memcached.

STEP 2: PERFORMANCE & UTILIZATION STATISTICS

Measurement

- **System performance:** Measured with **tail latencies** (95th, 99th).
- **Resource utilization:** Measured with **CPU%**.

Plan

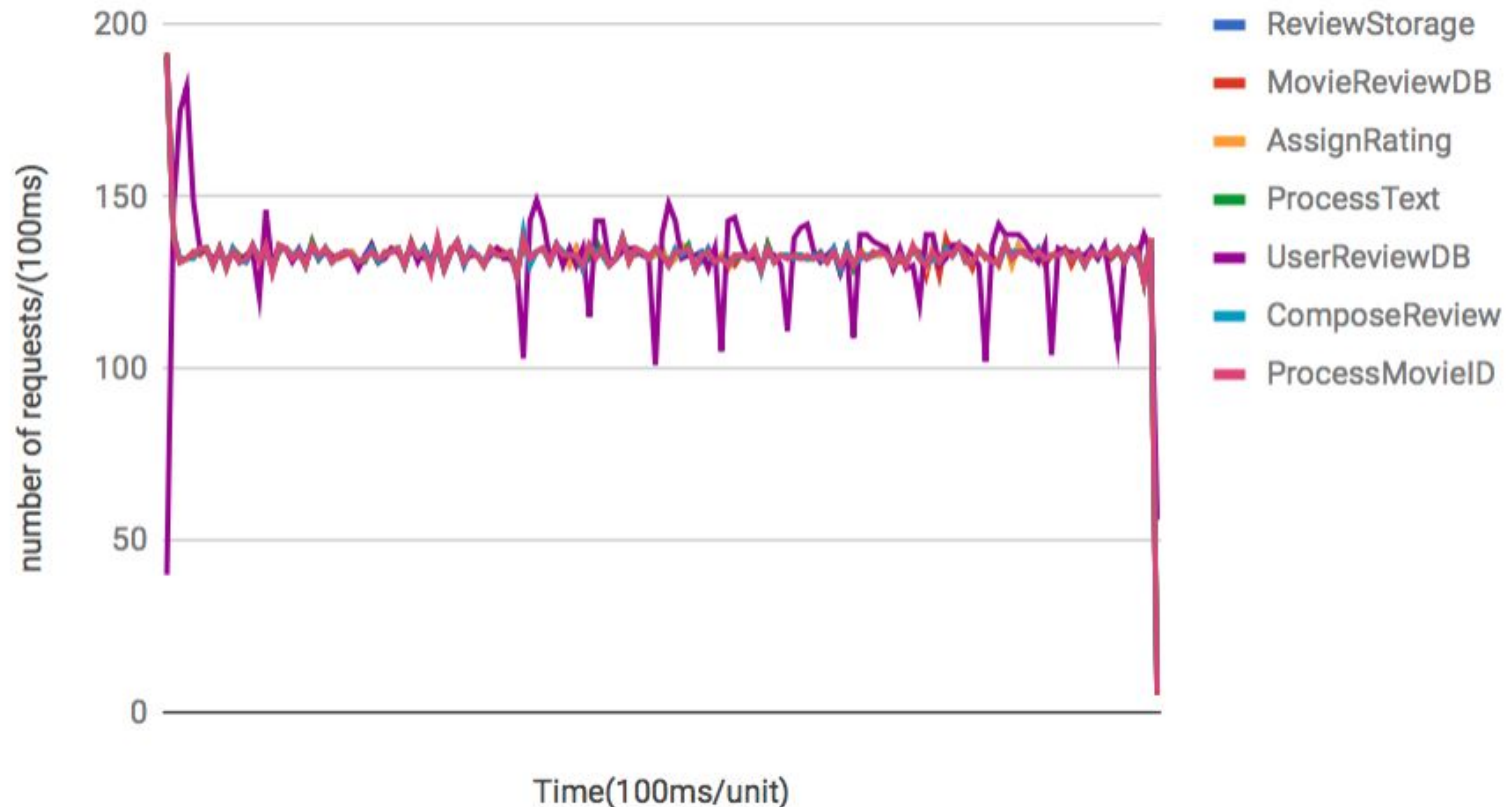
1. **Scale out** the system (using CPU%) to find an **ideal core configuration** for each service that services are equally loading/bottlenecking the system.
2. Understand how tail latency is **correlated** with CPU%.

STEP 2: ISSUES AND THEIR SOLUTIONS



- **Issue 1:** Tail latency graphs are having periodic spikes
- The spikes are presumably due to a non-uniformly distributed query input.

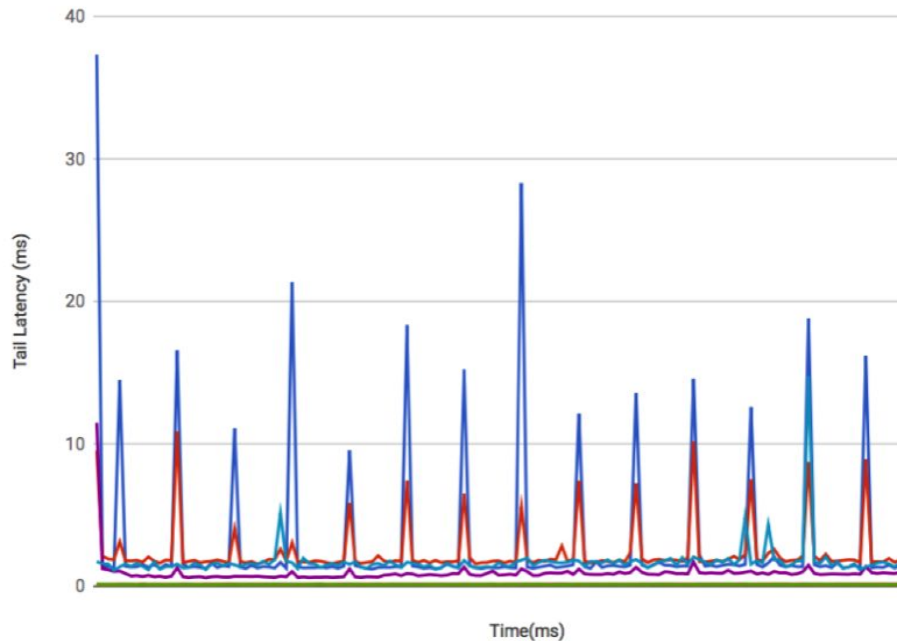
Request's distribution



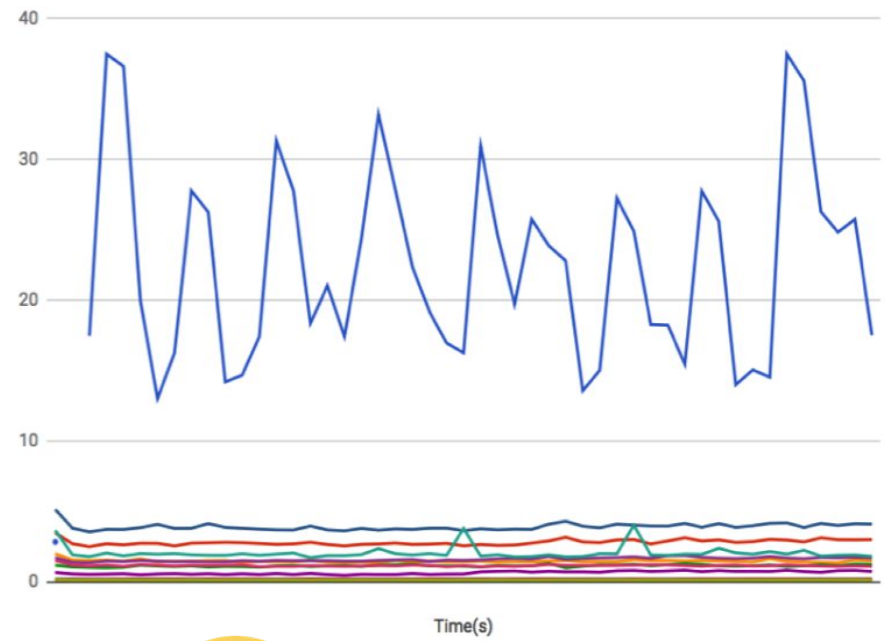
STEP 2: ISSUES AND THEIR SOLUTIONS



- **Solution:** Switch the client side script from the custom loader to **wrk2** for stable input distribution.
- Fig. 1&2: Nginx 6 - Memcached 1 - MongoDB 5, QPS = 2000, 95th Tail



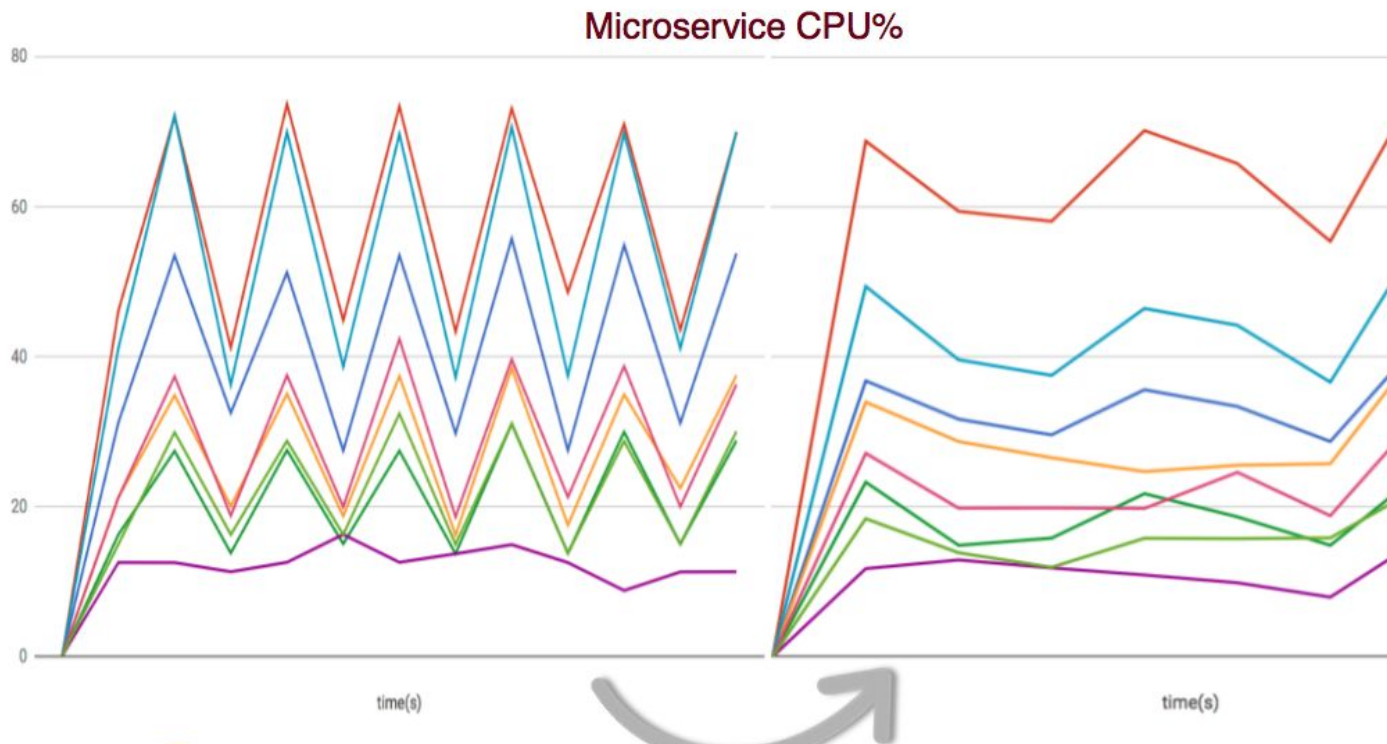
Erring Periodic Spike



No Periodic Spike

STEP 2: ISSUES AND THEIR SOLUTIONS

- **Issue 2:** Microservices are prone to dropping requests.
- **Solution:** Double the cores for microservices that deals with the review storing phase.
- Fig. 1&2: Ngx 6 - MMC 1 - MDB 5, QPS = 2000, 95th Tail



Large Queues:
Prone to drop requests



Smaller queues:
Smoother CPU%

STEP 2: IDEAL RESOURCE CONFIGURATION



- For each application, determining the **ideal resource configuration** (the ratio of resources allocated to each microservice) so that all microservices saturate relatively at the same time.
- This allows us to **efficiently utilize** our resources in **a minimal way**.
- Important to look at both **CPU utilization** and **latency**.
- Use this as starting point before adding interference, ensuring QoS violation is due to external interferences, instead of caused by the system itself.

STEP 2: ADDING INTERFERENCE



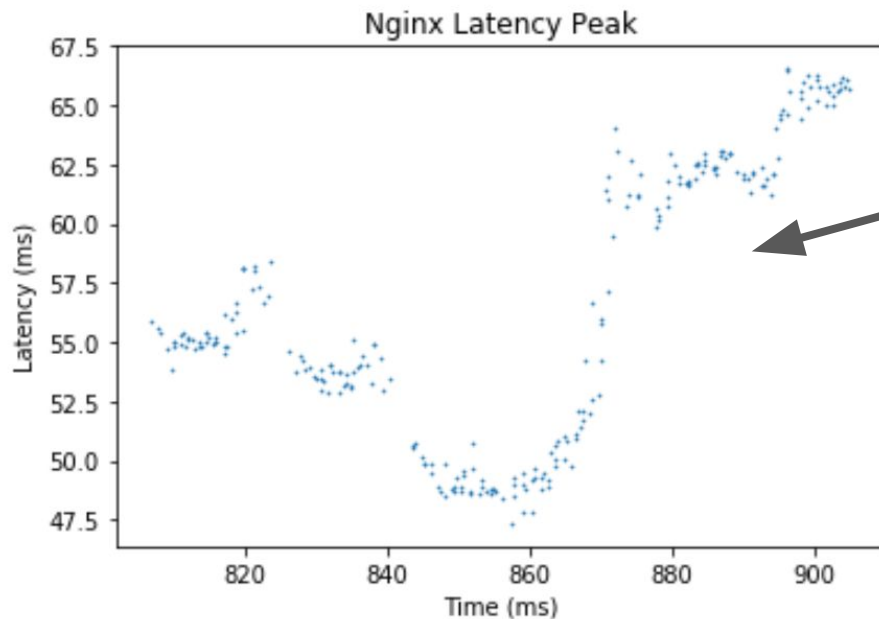
- Understanding, reducing, and managing interference between **co-scheduled processes** can significantly impact the design of a large-scale system.
- Therefore, we try to **inject interference into each microservice**, and record their performance and resource utilization accordingly.

- Using **iBench**, a workload suite that helps **quantify the interfering intensity** (linearly increasing to 100%). Interference sources span from **CPU, cache hierarchy, memory, storage, and network** subsystems.
- Now experimenting with CPU interference.

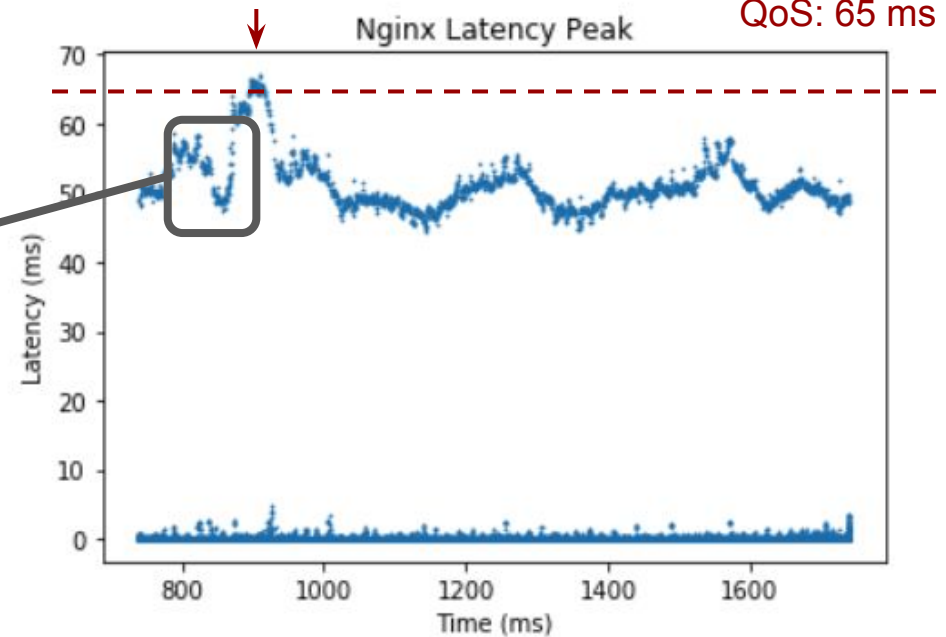
STEP 3: MACHINE LEARNING

- Idea: Use a sliding window to predict a future QoS violation

Number of requests: 3000
Time range: 98 ms



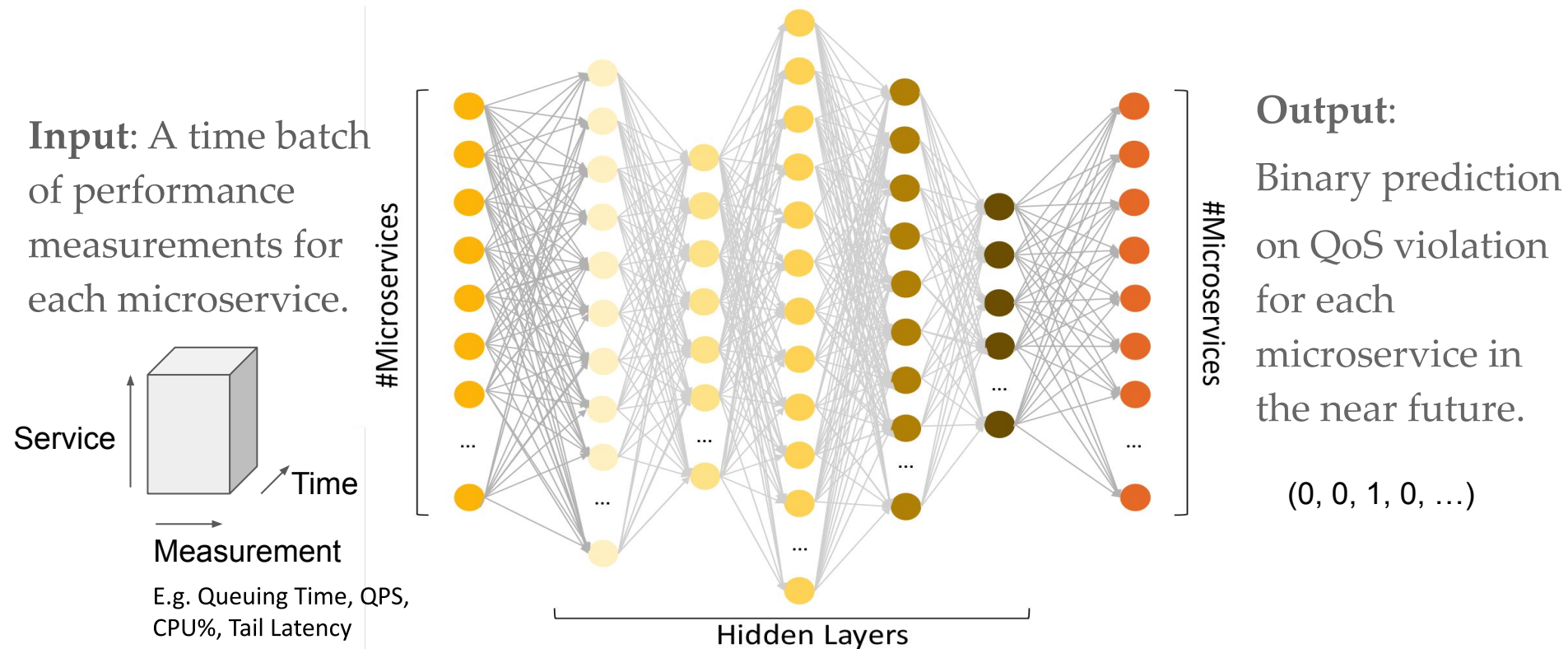
Number of requests: 30000
Time range: 1001 ms



QPS=30K

STEP 3: MACHINE LEARNING

- Plan: Use a deep neural network to predict which microservices are likely to cause a QoS violation in the near future, and re-allocate resource/re-balance requests accordingly to avoid the violation.



- Continue to run experiments to refine the ideal resource configuration.
- Inject interference into the microservices to collect spiky training data with QoS violations.
- Continue with the deep learning experiment, and tune the input dimension and neural network architecture to generate reasonable predictions.

- Google Dapper Paper
- The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services
- Distributed Tracing For Polyglot Microservices
- Microservices in Production: 5 Challenges You Should Know

THANK YOU!