The Hardware & Software Implications of Microservices and How Big Data Can Help

Christina Delimitrou
Cornell University

with Yu Gan, Yanqi Zhang, Shuang Chen, Neeraj Kulkarni, Shannon Joyner, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Ankitha Shetty, Nayan Katarki, Brett Clancy, Chris Colen, Dailun Cheng, Rick Lin, Anant Desai, Lavanya Kannan, Zhongling Liu, Jake Padilla, Han Li, Nicky Lim, Yang Liu, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Meghna Pancholi, Siyuan Hu

Stanford – October 11th 2018
Executive Summary

- **From monoliths to microservices:**
  - **Monolith:** all functionality in ~one service
  - **Microservices:** many single-concerned, loosely-coupled services
  - Modularity, specialization, accelerated development

- **Systems challenges & implications:**
  - SW design, management, server design
  - Motivate the need for data-driven approaches as system scale & complexity increases
From Monoliths to Microservices
Motivation

- Advantages of microservices:
  - Modular → easier to understand → less complex
  - Speed of development & deployment
  - On-demand provisioning/scaling
  - Debugging, error isolation & distinct failure domains
  - Language/framework heterogeneity

- Challenges of microservices:
  - Change server design assumptions → new bottlenecks
  - Complicate resource management → dependencies
  - Amplify tail-at-scale effects → performance unpredictability
  - No representative end-to-end apps with microservices
Challenge 1: Granularity

- Too few microservices $\rightarrow$ Too similar to monolith
- Too many microservices $\rightarrow$ High network overheads

Design guidelines:
- Single-function microservices
- Code needs to fit in one page (what font?)
- Code should not exceed 100 LoCs
- “Classitis” $\rightarrow$ few microservices are good, more are better
From Monoliths to Microservices

Class on Microservices Design

- Goal: given a monolith convert to microservices (social network)
- Audience: 28 undergrads (juniors-seniors)
- First offering: Summer 2018 (7 weeks)
- Next offering: Spring 2019 (10 weeks)

Week 1: Given monolith
Week 2: First working prototype
Week 3: Receive feedback
Week 4: Revise
Week 5: Present
Week 6: Update other group’s service
Week 7: Final presentation
Many creative pitfalls:
- Copy too much of the monolith
- Copy one microservice over and over (single point of failure)
- Shared database
- Synchronous communication, cyclic dependencies
- Too fine/Too coarse
- Too many/few languages

But in the end:
- All teams got a service working
- Almost all teams managed to add functionality to each other’s code

Improvements for next offering:
- Constrain languages/APIs & probably simplify service

Material will be publicly available on course website
Challenge 2: Network Implications

- From monoliths to microservices:
  - IPC $\rightarrow$ RPC/http
  - 100ms tail latency for monoliths $\rightarrow$ 100us per microservice
  - Network becomes a significant factor
    - A few us $\rightarrow$ significant increase in tail latency
    - Cascading effects across dependent microservices

- Similar OS overheads:
  - Context switch $\sim$20us
  - Interrupt handling
  - OS scheduling, synchronization
An End-to-End Suite for Cloud & IoT Microservices

- 6 end-to-end apps using popular open-source microservices → ~30-40 microservices per app
  - Social Network
  - Movie Reviewing/Renting/Streaming
  - E-commerce
  - Hotel Reservation
  - Secure Banking System (in progress)
  - Drone control service

- Programming languages and frameworks:
  - node.js, Python, C/C++, Java/JavaScript, Scala, PHP, and Go
  - Nginx, memcached, MongoDB, CockroachDB, Mahout, Xapian
  - Apache Thrift RPC, RESTful APIs
  - Docker containers
  - Lightweight RPC-level distributed tracing
Social Network

Social Network Service

Client

Load Balancer

nginx

php-fpm

followUser

readPost

blockedUsers

readTimeline

userInfo

usersStorage

writeTimeline

writeGraph

uniqueID

ads

recommender

video

image

text

userTag

favorite

search

index_0

index_1

index_n

urlShorten

login

http

fastcgi

http

Load Balancer

Add post
E-Commerce Service

E-commerce Service

Front-end (node.js)

socialNet

memcached

recommender

payment

authorization

transactionID

invoicing

accountInfo

wishlist

mongoDB

cart

queueMaster

orderQueue

shipping

media

catalogue

memcached

search

discounts

Order

Place order

Client

Load Balancer

http

http

memcached
Movie Streaming

- **Browse movie info** (movie plot, photos, videos, cast, stats, etc.)
- **ML widgets:**
  - Recommender for movies to watch
  - Recommender for ads
- **User authentication/Payment**
- **Search:**
  - Xapian: search movie DB
- **Analytics:**
  - Mahout: user analytics based on input stored in HDFS
  - Spark MLlib: in-memory ML analytics
Swarm Coordination (Centralized)

Controller

Image Recognition

Routing

Obstacle Avoidance

VideoDB

ImageDB

SensorCollection

Logging
Challenge 2: Network Implications

- **Computation : Communication ratio**
  - Monolithic service $\rightarrow$ 80:20 @ high load
  - Microservices $\rightarrow$ 40:60 @ high load
Challenge 2: Network Implications

- **Computation : Communication ratio**
  - Monolithic service → 80:20 @ high load
  - Microservices → 40:60 @ high load
  - RPC acceleration → FPGA offload (net filter)
**Challenge 3: Server Design**

Big vs. small servers:
- Power management using RAPL
- More pressure on single-thread performance, low tail latency
**Challenge 3: Server Design**

- **Big vs. small servers:**
  - Power management using RAPL
  - More pressure on single-thread performance, low tail latency

---

**Figure 1:** Network verification pressure on correctness verification. Edge devices also have limited resources, and may need to leverage the cloud for hardware acceleration to assist microservices. These services are written in languages for mobile computing.

**Figure 2:** Tail latency breakdown for different services. **Movie Streaming** shows the highest tail latency, whereas **Social Network** and **E-commerce** have lower tail latencies. **Banking System** has a moderate tail latency, and **Swarm-Cloud** has the lowest tail latency.

---

**ABSTRACT**

Microservices are appealing for several reasons. First, they help developers bring up services quickly and in isolation. Finally, the large number of available open-source services mapped to a physical host, improving mobility, and container, and containers of complementary resource requirements. A container-based datacenter, with a microservice per component, unlike monoliths, where resolving correctness issues often involves troubleshooting the entire application, debugging, as bugs can be isolated in specific applications. Services' design, with less effort. An increasing number of cloud providers, including Twitter, Netflix, AT&T, Amazon, and eBay have used microservices that is representative of large end-to-end services, modular and extensible. DeathStarBench includes a social network, a movie streaming service, an e-commerce services, revisit the big versus small core debate, and StarBench to study the architectural characteristics of microservices.

**INTRODUCTION**

Cloud computing services are governed by strict quality of service (QoS) constraints in terms of throughput and tail latency, as well as availability and reliability guarantees. In an effort to satisfy these, often contradicting constraints, latency, and may host safety-critical computation, which puts more pressure on single-thread performance, low tail latency. These services are written in languages for mobile computing. Cloud services have recently undergone a shift from monolithic services to microservices and containers. This includes a recent shift from monolithic services to microservices. The shift to microservices also has implications in server design. This includes determining whether big or small cores are required, only requiring a common cross-application model, there is a substantial risk of sacrificing both performance and resource utilization. These services are well suited for internet-of-things (IoT) applications. Despite their advantages, microservices change several assumptions we use to design and manage cloud systems. For monoliths and microservices.

**ISCA 2018 Submission**

**Power management using RAPL**

**More pressure on single-thread performance, low tail latency**
Challenge 3: Server Design

- **Big vs. small servers:**
  - Power management using RAPL
  - More pressure on single-thread performance, low tail latency
  - Low-power SoCs, e.g., Cavium ThunderX2
  - Similar latency, but earlier saturation
L1-i cache pressure:

- **Monoliths** → Large code footprints → L1i thrashing
- **Microservices** → Small footprint/microservice

- Assuming dedicated cores
Microservices dependencies:
- Hard to describe, infer impact of
- Change over time
- Create backpressure
- Cascading QoS violations
Challenge 4: Cluster Management

- Hard to describe dependencies:
  - Create backpressure
  - Cascading QoS violations
  - Change over time

- Different services scale differently with load
Dependencies & Backpressure
Autoscaling may help/penalize the wrong microservice

Difficult to infer dependency impact without knowing app semantics
Scalability Challenges
Challenge 4: Cluster Management

- **Social Network**

- **Microservices dependencies:**
  - Create backpressure
  - Cascading QoS violations
Challenge 4: Cluster Management

- Microservices dependencies:
  - Create backpressure
  - Cascading QoS violations
Challenge 5: Tail @ Scale Effects

- Monolith $\rightarrow$ $\sim$linear
- Microservices $\rightarrow$ $\sim$exp
- Amplify fanout impact
- Any bottlenecked service on the critical path degrades QoS
- Gets worse as cluster size increases

![Graph showing Max QPS at QoS vs Slow Servers (%)]
Using ML to Navigate Cloud Complexity

- DSL to Automatically Synthesize End-to-end Microservices
- Fault-Tolerant Coordination Control in Swarms
- Cluster Management to Meet End-to-end QoS
- Practical Performance Attacks for Microservices
- Resource Provisioning for Serverless Compute
- Proactive Performance Debugging for Microservices
Proactive Performance Debugging

- Dependencies → cascading QoS violations
  - Finding the culprit of a violation is difficult
  - Returning to nominal operation is slow

- Anticipating QoS violations & identifying culprits

- Seer: Data-driven Performance Debugging for Microservices
  - Combines lightweight RPC-level distributed tracing with hardware monitoring
  - Leverages scalable deep learning to signal QoS violations with enough slack to apply corrective action
Performance Implications
### Performance Implications

<table>
<thead>
<tr>
<th>Queue</th>
<th>CPU</th>
<th>Mem</th>
<th>Net</th>
<th>Disk</th>
</tr>
</thead>
</table>

The performance implications are visualized using a heat map, where different symbols represent various performance metrics such as queue, CPU, memory, network, and disk. The intensity of the symbols indicates the level of impact on each metric.
Leverage the massive amount of traces collected over time

1. **Distributed tracing**: Use ML to identify the culprit of an upcoming QoS violation in RPC-level traces

2. **Hardware probes**: Use per-server hardware monitoring to determine the root cause of a QoS violation

3. Take corrective action to prevent the QoS violation

- Need to predict 100s of msec – a few sec in the future
Tracing Framework

- RPC level tracing
- Based on Apache Thrift
- Timestamp start-end for each microservice
- Store in centralized DB (Cassandra)
- Record all requests → No sampling
- Overhead: <0.1% in throughput and <0.2% in tail latency
Deep Learning to the Rescue

- **Why?**
  - Architecture-agnostic
  - Adjusts to changes over time
  - High accuracy, good scalability & fast inference (within window of opportunity)

Deep Learning to the Rescue

- **Why?**
  - Architecture-agnostic
  - Adjusts to changes over time
  - High accuracy, good scalability & fast inference (within window of opportunity)
Seer Configuration

Which microservice will cause a QoS violation in the near future?

- Container utilization
- Latency
- Queue depths

Input signal

# Microservices (in dependencies order)

Dimensionality reduction

Near-future prediction

Output signal

QoS Violation Detection Accuracy (%)

- CNN
- LSTM
- Seer (CNN+LSTM)

Inference Time (ms)

0 2 4 6 8 10 12 14 16

56.1
70.75
79.43
Seer Configuration

Input signal
- Container utilization
- Latency
- Queue depths

Dimensionality reduction

Near-future prediction

Output signal
Which microservice will cause a QoS violation in the near future?

#Microservices (Pr for QoS violation)
Seer Configuration

- **Training** once: slow (hours - days)
  - Across load levels, load distributions, request types
  - Distributed queue traces, annotated with QoS violations
  - Weight/bias inference with SGD
  - Retraining in the background

- **Inference** continuously: streaming trace data

93% accuracy in signaling upcoming QoS violations
91% accuracy in attributing QoS violation to correct microservice
Seer Sensitivity

- Large increase in accuracy until ~50GB training set
  - Levels off afterwards
- Large increase in training time after that

- Frequency < 1/500ms → low accuracy
- Frequency > 1/200ms → no further improvement
Challenges:

- In large clusters inference too slow to prevent QoS violations
- Offload to TPU v2, 10-100x faster; 10ms for 90th %ile inference
- Fast enough for most corrective actions to take effect

Accuracy stable or increasing with cluster size
Experimental Setup

- 200 dedicated servers
- \(\approx 1000\) single-concerned containers
- Machine utilization 80-85%
- Inject interference to cause QoS violation
  - Using microbenchmarks (CPU, cache, memory, network, disk I/O)
Restoring QoS

- **Identify cause of QoS violation**
  - Private cluster: performance counters & utilization monitors
  - Public cluster: contentious microbenchmarks

- **Adjust resource allocation**
  - RAPL (fine-grain DVFS) & scale-up for CPU contention
  - Cache partitioning (CAT) for cache contention
  - Memory capacity partitioning for memory contention
  - Network bandwidth partitioning (HTB) for net contention
  - Storage bandwidth partitioning for I/O contention
Demo

Queue
CPU
Mem
Net
Disk

Seer

Default
Demo
Evaluation

- Post-detection, baseline system → dropped requests
- Post-detection, Seer → maintain nominal performance
Using ML to Design Better Systems

- ~2 months deployment
- ~600 users (~180 average daily users)
- Several bugs found (blocking RPCs, livelocks, shared data structs, cyclic dependencies, insufficient resources, etc.)
- Fewer QoS violations over time
Rethinking the Cloud Stack

Applications

Programming frameworks

Cluster management

Hardware design

CAL’18a

in submission

CAL’18b,

HotCloud’18,

in submission

in submission
Data-driven approaches can help navigate the increasing complexity of the cloud.