An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems

Yu Gan
Cornell University
yg397@cornell.edu

Yanqi Zhang
Cornell University
yz2297@cornell.edu

Dailun Cheng
Cornell University
dc924@cornell.edu

Ankitha Shetty
Cornell University
aas394@cornell.edu

Priyal Rathi
Cornell University
pr348@cornell.edu

Nayan Katarki
Cornell University
nk646@cornell.edu

Ariana Bruno
Cornell University
amb633@cornell.edu

Justin Hu
Cornell University
jh2625@cornell.edu

Brian Ritchken
Cornell University
bjr96@cornell.edu

Brendon Jackson
Cornell University
btj28@cornell.edu

Kelvin Hu
Cornell University
sh2442@cornell.edu

Meghna Pancholi
Cornell University
mp832@cornell.edu

Yuan He
Cornell University
yhe772@cornell.edu

Brett Clancy
Cornell University
bjc265@cornell.edu

Chris Colen
Cornell University
cdc99@cornell.edu

Fukang Wen
Cornell University
fw224@cornell.edu

Catherine Leung
Cornell University
chl66@cornell.edu

Siyuan Wang
Cornell University
sw884@cornell.edu

Leon Zaruvinsky
Cornell University
laz37@cornell.edu

Mateo Espinosa
Cornell University
me326@cornell.edu

Rick Lin
Cornell University
c12545@cornell.edu

Zhongling Liu
Cornell University
zlj682@cornell.edu

Jake Padilla
Cornell University
jsp264@cornell.edu

Christina Delimitrou
Cornell University
delimitrou@cornell.edu

Abstract

Cloud services have recently started undergoing a major shift from monolithic applications, to graphs of hundreds of loosely-coupled microservices. Microservices fundamentally change a lot of assumptions current cloud systems are designed with, and present both opportunities and challenges when optimizing for quality of service (QoS) and utilization.

In this paper we explore the implications microservices have across the cloud system stack. We first present DeathStarBench, a novel, open-source benchmark suite built with microservices that is representative of large end-to-end services, modular and extensible. DeathStarBench includes a social network, a media service, an e-commerce site, a banking system, and IoT applications for coordination control of UAV swarms. We then use DeathStarBench to study the architectural characteristics of microservices, their implications in networking and operating systems, their challenges with respect to cluster management, and their trade-offs in terms of application design and programming frameworks. Finally, we explore the tail at scale effects of microservices in real deployments with hundreds of users, and highlight the increased pressure they put on performance predictability.

CCS Concepts • Computer systems organization → Cloud computing; • Software and its engineering → n-tier architectures; Cloud computing.

Keywords cloud computing, datacenters, microservices, cluster management, serverless, acceleration, fpga, QoS

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASPLOS’19, April 13–17, 2019, Providence, RI, USA © 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-6240-5/19/04...$15.00
https://doi.org/10.1145/3297858.3304013

ACM Reference Format:
1 Introduction

Large-scale datacenters host an increasing number of popular online cloud services that span all aspects of human endeavor. Many of these applications are interactive, latency-critical services that must meet strict performance (throughput and tail latency), and availability constraints, while also handling frequent software updates [21, 28–34, 36, 44, 51, 61, 62, 65]. The effort to satisfy these often contradicting requirements has pushed datacenter applications on the verge of a major design shift, from complex monolithic services that encompass the entire application functionality in a single binary, to graphs with tens or hundreds of single-purpose, loosely-coupled microservices. This shift is becoming increasingly pervasive with large cloud providers, such as Amazon, Twitter, Netflix, Apple, and EBay having already adopted the microservices application model [6, 18, 19], and Netflix reporting more than 200 unique microservices in their ecosystem, as of the end of 2016 [18, 19].

The increasing popularity of microservices is justified by several reasons. First, they promote composable software design, simplifying and accelerating development, with each microservice being responsible for a small subset of the application’s functionality. The richer the functionality of cloud services becomes, the more the modular design of microservices helps manage system complexity. They similarly facilitate deploying, scaling, and updating individual microservices independently, avoiding long development cycles, and improving elasticity. Fig. 1 shows the deployment differences between a traditional monolithic service, and an application built with microservices. While the entire monolith is scaled out on multiple servers, microservices allow individual components of the end-to-end application to be elastically scaled, with microservices of complementary resources bin-packed on the same physical server. Even though modularity in cloud services was already part of the Service-Oriented Architecture (SOA) design approach [77], the fine granularity of microservices, and their independent deployment create hardware and software challenges different from those in traditional SOA workloads.

Second, microservices enable programming language and framework heterogeneity, with each tier developed in the most suitable language, only requiring a common API for microservices to communicate with each other; typically over remote procedure calls (RPC) [1, 7, 9] or a RESTful API. In contrast, monoliths limit the languages used for development, and make frequent updates cumbersome and error-prone.

Finally, microservices simplify correctness and performance debugging, as bugs can be isolated in specific tiers, unlike monoliths, where resolving bugs often involves troubleshooting the entire service. This makes them additionally applicable to internet-of-things (IoT) applications, that often host mission-critical computation, which puts more pressure on correctness verification [40, 43].

Despite their advantages, microservices represent a significant departure from the way cloud services are traditionally designed, and have broad implications ranging from cloud management and programming frameworks, to operating systems and datacenter hardware design.

In this paper we explore the implications microservices have across the cloud system stack, from hardware all the way to application design, using a suite of new end-to-end and representative applications built with microservices each. The DeathStarBench suite\footnote{Named after the DeathStar graphs that visualize dependencies between microservices [18, 19].} includes six end-to-end services that cover a wide spectrum of popular cloud and edge services: a social network, a media service (movie reviewing, renting, streaming), an e-commerce site, a secure banking system, and Swarm; an IoT service for coordination control of drone swarms, with and without a cloud backend.

Each service includes tens of microservices in different languages and programming models, including node.js, Python, C/C++, Java, Javascript, Scala, and Go, and leverages opensource applications, such as NGINX [13], memcached [39], MongoDB [12], Cylon [5], and Xapian [51]. To create the end-to-end services, we built custom RPC and RESTful APIs using popular open-source frameworks like Apache Thrift [1], and gRPC [9]. Finally, to track how user requests progress through microservices, we have developed a lightweight and transparent to the user distributed tracing system, similar to Dapper [76] and Zipkin [17] that tracks requests at RPC granularity, associates RPCs belonging to the same end-to-end request, and records traces in a centralized database. We study both traffic generated by real users of the services, and synthetic loads generated by open-loop workload generators.

We use these services to study the implications of microservices spanning the system stack, as seen in Fig. 2. First, we quantify how effective current datacenter architectures are at...
running microservices, as well as how datacenter hardware needs to change to better accommodate their performance and resource requirements (Section 4). This includes analyzing the cycle breakdown in modern servers, examining whether big or small cores are preferable [25, 35, 41, 42, 46–48], determining the pressure microservices put on instruction caches [37, 52], and exploring the potential they have for hardware acceleration [24, 27, 38, 49, 71]. We show that despite the small amount of computation per microservice, the latency requirements of each individual tier are much stricter than for typical applications, putting more pressure on predictably high single-thread performance.

Second, we quantify the networking and operating system implications of microservices. Specifically, we show that, similarly to traditional cloud applications, microservices spend a large fraction of time in the kernel. Unlike monolithic services though, microservices spend much more time sending and processing network requests over RPCs or other REST APIs. Fig. 3 shows the breakdown of execution time to network (red) and application processing (green) for three monolithic services (NGINX, memcached, MongoDB) and the end-to-end Social Network application. While for the single-tier services only a small amount of time goes towards network processing, when using microservices, this time increases to 36.3% of total execution time, causing the system’s resource bottlenecks to change drastically. In Section 5 we show that offloading RPC processing to an FPGA tightly-coupled with the host server, can improve network performance by 10–60x.

Third, microservices significantly complicate cluster management. Even though the cluster manager can scale out individual microservices on-demand instead of the entire monolithic, dependencies between microservices introduce back-pressure effects and cascading QoS violations that quickly propagate through the system, making performance unpredictable. Existing cluster managers that optimize for performance and/or utilization [29, 32, 33, 36, 45, 60–62, 64, 66–68, 73, 80, 84] are not expressive enough to account for the impact each pair-wise dependency has on end-to-end performance. In Section 6, we show that mismanaging even a single such dependency dramatically hurts tail latency, e.g., by 10.4x for the Social Network, and requires long periods for the system to recover, compared to the corresponding monolithic service. We also show that traditional autoscaling mechanisms, present in many cloud infrastructures, fall short of addressing QoS violations caused by dependencies between microservices.

Fourth, in Section 7, we identify microservices creating bottlenecks in the end-to-end service’s critical path, quantify the performance trade-offs between RPC and RESTful APIs, and explore the performance and cost implications of running microservices on serverless programming frameworks.

Finally, given that performance issues in the cloud often only emerge at large scale [28], in Section 8 we use real application deployments with hundreds of users to show that tail-at-scale effects become more pronounced in microservices compared to monolithic applications, as a single poorly-configured microservice, or slow server can degrade end-to-end latency by several orders of magnitude.

As microservices continue to evolve, it is essential for datacenter hardware, operating and networking systems, cluster managers, and programming frameworks to also evolve with them, to ensure that their prevalence does not come at a performance and/or efficiency loss. DeathStarBench is currently used in several academic and industrial institutions with applications in serverless compute, hardware acceleration, and runtime management. We hope that open-sourcing it to a wider audience will encourage more research in this emerging field.

2 Related Work

Cloud applications have attracted a lot of attention over the past decade, with several benchmark suites being released both from academia and industry [37, 44, 51, 81, 88]. Cloudsuite for example, includes both batch and interactive services, such as memcached, and has been used to study the architectural implications of cloud benchmarks [37]. Similarly, TailBench aggregates a set of interactive benchmarks, from web servers and databases to speech recognition and machine translation systems and proposes a new methodology to analyze their performance [51]. Sirius also focuses on intelligent personal assistant workloads, such as voice to text translation, and has been used to study the acceleration potential for interactive ML applications [44].

A limitation of these benchmark suites is that they focus on single-tier applications, or at most services with two or three tiers, which drastically deviates from the way cloud services are deployed today. For example, even applications like websearch, which is a classic multi-tier workload, are configured as independent leaf nodes, which does not capture correlations across tiers. As we show in Sections 4–7 studying the effects of microservices using existing benchmarks leads to fundamentally different conclusions altogether.

The emergence of microservices has prompted recent work to study their characteristics and requirements [55, 78, 79, 86]. µSuite for example quantifies the system call, context
We first describe the suite’s design principles, and then present DeathStarBench adheres to the following design principles:

### 3.1 Design Principles

DeathStarBench adheres to the following design principles:

- **Representativeness**: The suite is built using popular open-source applications deployed by cloud providers, such as NGINX [13], memcached [39], MongoDB [12], RabbitMQ [15], MySQL, Apache http server, ardrone-autonomy [2, 5], and the Sockshop microservices by Weave [16]. Most new code corresponds to interfaces between the services, using Apache Thrift [1], gRPC [9], or http requests.

- **End-to-end operation**: Open-source cloud services, such as memcached, can function as components of a larger service, but do not capture the impact of inter-service dependencies on end-to-end performance. DeathStarBench instead implements the full functionality of a service from the moment a request is generated at the client until it reaches the service’s backend and/or returns to the client.

- **Heterogeneity**: The software heterogeneity is both a challenge and opportunity with microservices, as different languages mean different bottlenecks, synchronization primitives, levels of indirection, and development effort. The suite uses applications in low- and high-level, managed and unmanaged languages including C/C++, Java, Javascript, node.js, Python, html, Ruby, Go, and Scala.

- **Modularity**: We follow Conway’s Law [4], i.e., the fact that the software architecture of a service follows the architecture of the company that built it in the design of the end-to-end applications, to avoid excessive two-way communication between any two dependent microservices, and to ensure they are single-concerned and loosely-coupled.

- **Reconfigurability**: Easily updating components of a larger service is one of the main advantages of microservices. Our RPC/HTTP API allows swapping out microservices for alternate versions, with small changes to existing components.

Table 1 shows the developed LoCs per service, and the LoCs for the communication protocol; hand-written, and auto-generated by Thrift, where applicable. The majority of new code for the Social Network, Media, E-commerce, and Banking services goes towards the cross-microservice API, as well as a few microservices for which no open-source framework existed, e.g., assigning ratings to movies. For the Swarm application, we show code breakdown for two versions; one where the majority of computation happens in a backend cloud (Swarm Cloud), and one where it happens locally on the edge devices (Swarm Edge). We also show the number of unique microservices for each application, and the breakdown per programming language. Unless otherwise noted, all microservices run in Docker containers.

### 3.2 Social Network

**Scope**: The end-to-end service implements a broadcast-style social network with uni-directional follow relationships.

**Functionality**: Fig. 4 shows the architecture of the end-to-end service. Users (client) send requests over http, which first reach a load balancer, implemented with nginx. Once a switch, and other OS overheads in microservices [78], while Ueda et al. [79] show the impact of compute resource allocation, application framework, and container configuration on the performance and scalability of several microservices. DeathstarBench differentiates from these studies by focusing on large-scale applications with tens of unique microservices, allowing us to study effects that only emerge at large scale, such as network contention and cascading QoS violations due to dependencies between tiers, as well as by including diverse applications that span social networks, media and e-commerce services, and applications running on swarms of edge devices.

### 3 The DeathStarBench Suite

We first describe the suite’s design principles, and then present the architecture and functionality of each end-to-end service.

<table>
<thead>
<tr>
<th>Service</th>
<th>Total New LoCs</th>
<th>Comm. Protocol</th>
<th>LoCs for RPC/REST</th>
<th>Unique Microservices</th>
<th>Per-language LoC breakdown (end-to-end service)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Social Network</td>
<td>15,198</td>
<td>RPC</td>
<td>9,286</td>
<td>52,863</td>
<td>34% C, 23% C++, 18% Java, 7% node.js, 6% Python, 5% Scala, 3% PHP, 2% Javascript, 2% Go</td>
</tr>
<tr>
<td>Movie Reviewing</td>
<td>12,155</td>
<td>RPC</td>
<td>9,853</td>
<td>48,001</td>
<td>30% C, 21% C++, 20% Java, 10% PHP, 8% Scala, 5% node.js, 3% Python, 3% Javascript</td>
</tr>
<tr>
<td>Website</td>
<td>16,194</td>
<td>REST</td>
<td>4,798</td>
<td>-</td>
<td>21% Java, 16% C++, 15% C, 14% Go, 16% Javascript, 7% node.js, 5% Scala, 4% HTML, 3% Ruby</td>
</tr>
<tr>
<td>Banking</td>
<td>13,876</td>
<td>RPC</td>
<td>4,757</td>
<td>31,156</td>
<td>29% C, 23% Javascript, 16% Java, 16% node.js, 11% C++, 3% Python</td>
</tr>
<tr>
<td>Swarm Cloud</td>
<td>11,283</td>
<td>REST</td>
<td>2,610</td>
<td>-</td>
<td>25</td>
</tr>
<tr>
<td>Swarm Edge</td>
<td>13,876</td>
<td>REST</td>
<td>4,757</td>
<td>-</td>
<td>21</td>
</tr>
</tbody>
</table>

Table 1. Characteristics and code composition of each end-to-end microservices-based application.
specific webserver is selected, also in nginx, the latter uses a php-fpm module to talk to the microservices responsible for composing and displaying posts, as well as microservices for advertisements, search engines, etc. All messages downstream of php-fpm are Apache Thrift RPCs [1]. Users can create posts embedded with text, media, links, and tags to other users. Their posts are then broadcasted to all their followers. Users can also read, favorite, and repost posts, as well as reply publicly, or send a direct message to another user. The application also includes machine learning plugins, such as ads and user recommender engines [22, 23, 53, 83], a search service using Xapian [51], and microservices to record and display user statistics, e.g., number of followers, and to allow users to follow, unfollow, or block other accounts. The service’s backend uses memcached for caching, and MongoDB for persistent storage for posts, profiles, media, and recommendations. Finally, the service is instrumented with a distributed tracing system (Sec. 3.7), which records the latency of each network request and per-microservice processing; traces are recorded in a centralized database. The service is broadly deployed at our institution, currently servicing several hundred users. We use this deployment to quantify the tail at scale effects of microservices in Section 8.

3.3 Media Service

Scope: The application implements an end-to-end service for browsing movie information, as well as reviewing, rating, renting, and streaming movies [18, 19].

Functionality: Fig. 5 shows the architecture of the end-to-end service. As with the social network, a client request hits the load balancer, which distributes requests among multiple nginx web servers. Users can search and browse information about movies, including their plot, photos, videos, cast, and review information, as well as insert new reviews in the system for a specific movie by logging into their account. Users can also select to rent a movie, which involves a payment authentication module to verify that the user has enough funds, and a video streaming module using nginx-hls, a production nginx module for HTTP live streaming. The actual movie files are stored in NFS, to avoid the latency and complexity of accessing chunked records from non-relational databases, while movie reviews are kept in memcached and MongoDB instances. Movie information is maintained in a shard and replicated MySQL database. The application also includes movie and advertisement recommenders, as well as a couple auxiliary services for maintenance and service discovery, which are not shown in the figure. We are similarly deploying Media Service as a hosting site for project demos at Cornell, which members of the community can browse and review.

3.4 E-Commerce Service

Scope: The service implements an e-commerce site for clothing. The design draws inspiration, and uses several components of the open-source Sockshop application [16].

Functionality: Fig. 6 shows the architecture of the end-to-end service. The application front-end in this case is a node.js service. Clients can use the service to browse the inventory using catalogue, a Go microservice that mines the back-end memcached and MongoDB instances holding information about products. Users can also place orders (Go) by adding items to their cart (Java). After they log in (Go) to their account, they can select shipping options (Java), process their payment (Go), and obtain an invoice (Java) for their order. Orders are serialized and committed using QueueMaster (Go). Finally, the service includes a recommender engine for suggested products, and microservices for creating an item wishlist (Java), and displaying current discounts.

3.5 Banking System

Scope: The service implements a secure banking system, which users leverage to process payments, request loans, or balance their credit card.
Functionality: Users interface with a node.js front-end, similar to the one in the E-commerce service to login to their account, search information about the bank, or contact a representative. Once logged in, a user can process a payment from their account, pay their credit card or request a new one, browse information about loans or request one, and obtain information about wealth management options. Most microservices are written in Java and Javascript. The back-end databases consist of in-memory memcached, and persistent MongoDB instances. The service also has a relational database (BankInfoDB) that includes information about the bank, its services, and representatives.

3.6 Swarm Coordination

Scope: Finally, we explore a different execution environment for microservices, where applications run both on the cloud and on edge devices. The service coordinates the routing of a swarm of programmable drones, which perform image recognition and obstacle avoidance.

Functionality: We explore two version of this service. In the first (Fig. 8a), the majority of the computation happens on the drones, including the motion planning, image recognition, and obstacle avoidance, with the cloud only constructing the initial route per-drone (Java service ConstructRoute), and holding persistent copies of sensor data. This architecture avoids the high network latency between cloud and edge, however, it is limited by the on-board resources. The Controller and MotionController are implemented in Javascript, while ImageRecognition is using jimp, a node.js library for image recognition [11], and ObstacleAvoidance in C++. Services on the drones run natively, and communicate with each other over IPC, while the cloud and drones communicate over http to avoid installing the heavy dependencies of Thrift on the edge devices.

In the second version (Fig. 8b), the cloud is responsible for most of the computation. It performs motion control, image recognition, and obstacle avoidance for all drones, using the ardrone-autonomy [2], and Cylon [5] libraries, in OpenCV and Javascript respectively. The edge devices are only responsible for collecting sensor data and transmitting them to the cloud, as well as recording some diagnostics using a local node.js logging service. In this case, almost every action suffers the cloud-edge network latency, although services benefit from the additional cloud resources. We use 24 programmable Parrot AR2.0 drones (a subset is seen in Fig. 8c), together with a backend cluster of 20 two-socket,
40-core servers. Drones communicate with each other and the cluster over a wireless router.

3.7 Methodological Challenges of Microservices
A major challenge with microservices is that one cannot simply rely on the client to report performance, as with traditional client-server applications. Resolving performance issues requires determining which microservice(s) is the culprit of a QoS violation, which typically happens through distributed tracing. We developed and deployed a distributed tracing system that records per-microservice latencies at RPC granularity using the Thrift timing interface. RPCs or REST requests are timestamped upon arrival and departure from each microservice by the tracing module, and data is accumulated by the Trace Collector, implemented similarly to the Zipkin Collector [17], and stored in a centralized Cassandra database. We additionally track the time spent processing network requests, as opposed to application computation using a similar methodology to [58]. We verify that the overhead from tracing is negligible, less than 0.1% on end-to-end latency in all cases, which is tolerable for such systems [26, 72, 76].

3.8 Provisioning & Query Diversity
Before characterizing the architectural behavior of microservices, we provision the end-to-end applications to ensure that microservices are used in a balanced way, and that no single microservice introduces early bottlenecks due to resource saturation. To do so, we start with a fair resource allocation for all microservices of an end-to-end workload, and upsize saturated microservices until all tiers saturate at about the same load. The ratio of resources between tiers varies significantly across end-to-end services, highlighting the need for application-aware resource management.

Different query types also achieve different performance in each service. For example, composePost requests in the Social Network vary in the media they embed in a message, ranging from text-only messages, to posts including image and video files (we keep videos within a few MBs, similar to the allowable video sizes in production social networks like Twitter). Reposting a post incurs the longest latency across query types for Social Network, as it must first read an existing post, prepend to it, and then propagate the message across the user’s followers’ timelines.

In E-commerce, on the other hand, placing an order, which includes adding an item to the cart, logging in to the account, confirming payment, and selecting shipping, takes 1-2 orders of magnitude longer than browsing the eshop’s catalogue. In reality, placing an order requires interaction with the end user; in our case we automate the client’s decisions so they incur zero delay, making latency server-dominated. The trends across query types are similar for the Media and Banking services, with processing payments, either to rent a movie, or to perform a transaction in a bank account, dominating latency and defining each service’s saturation point.

Finally, in Fig. 9, we compare the performance of the IoT application when computation happens at the edge versus the cloud. Since drones have to communicate with a wireless router over a distance of several tens of meters, latencies are significantly higher than for the cloud-only services. When processing happens in the cloud, latency at low load is higher, penalized by the long network delay. As load increases however, edge devices quickly become oversubscribed due to the limited on-board resources, with processing on the cloud achieving 7.8x higher throughput for the same tail latency, or 20x lower latency for the same throughput. Obstacle avoidance shows a different trade-off, since it is less compute-intensive, and more latency-critical. Offloading obstacle avoidance to the cloud at low load can have catastrophic consequences if route adjustment is delayed, which highlights the importance of latency-aware resource management between cloud and edge, especially for safety-critical computation.

4 Architectural Implications

Methodology: We first evaluate the end-to-end services on a local cluster with 20 two-socket 40-core Intel Xeon servers (E2699-v4 and E5-2660 v3) with 128-256GB memory each, connected to a 10Gbps ToR switch with 10Gbe NICs. All servers are running Ubuntu 16.04, and unless otherwise noted power management and turbo boosting are turned off.

Cycles breakdown and IPC: We use Intel VTune [10] to break down the cycles, and identify bottlenecks. Fig. 10
shows the IPC and cycles for each microservice in the Social Network and E-commerce services. We omit the figures for the other services, however the observations are similar.

Across all services a large fraction of cycles, often the majority, is spent in the processor front-end. Front-end stalls occur for several reasons, including long memory accesses and i-cache misses. This is consistent with studies on traditional cloud applications [37, 50], although to a lesser extent for microservices than for monolithic services (memcached, mongodb), given their smaller code footprint. The majority of front-end stalls are due to fetch, while branch mispredictions account for a smaller fraction of stalls for microservices than for other interactive applications, either cloud or IoT [37, 88]. Only a small fraction of total cycles goes towards committing instructions (21% on average for Social Network), denoting that current systems are poorly provisioned for microservices-based applications.

E-commerce includes a few microservices that go against this trend, with high IPC and high percentage of retired instructions, such as Search. Search (xapian [51]) is already optimized for memory locality, and has a relatively small codebase, which explains the fewer front-end stalls. The same applies for simple microservices, such as the wishlist for which i-cache misses are practically negligible. E-commerce also includes a recommender engine, whose IPC is extremely low; this is again in agreement with studies on the architectural behavior of ML applications [44]. The challenge with microservices is that although individual application components may be well understood, the structure of the end-to-end dependency graph defines how individual services affect the overall performance. For both services, we also show the cycles breakdown and IPC for corresponding applications with the same end-to-end functionality from the user’s perspective, but built as monoliths. In both cases, monoliths are developed in Java, and include all application functionality, except for the backend databases (memcached and MongoDB), in a single binary. The cycles breakdown is not drastically different for monoliths compared to microservices, although they experience slightly higher percentages of committed instructions, due to reduced front-end stalls, as they are less likely to wait for network requests to complete. IPC is also similar to microservices, and consistent with previous studies on cloud services [37, 51].

I-cache pressure: Prior work has characterized the high pressure cloud applications put on the instruction caches [37, 52]. Since microservices decompose what would be one large binary to many small, loosely-connected services, we examine whether previous results on i-cache pressure still hold. Fig. 11 shows the MPKI of each microservice for the Social Network and E-commerce applications. We also include the back-end caching and database layers, as well as the corresponding L1i MPKI for the monolithic implementations.

First, the i-cache pressure of nginx, memcached, MongoDB, and especially the monoliths remains high, consistent with prior work [37, 52, 88]. The i-cache pressure of the remaining microservices though is considerably lower, especially for E-commerce, an expected observation given the microservices’ small code footprints. Since node.js applications outside the context of microservices do not have low i-cache miss rates [88], we conclude that it is the simplicity of microservices which results in better i-cache locality. Most L1i misses, especially in the Social Network happen in the kernel, and are caused by Thrift. We also examined the LLC and D-TLB misses, and found them considerably lower than for traditional cloud applications, which is consistent with the push for microservices to be mostly stateless.

Brawny vs. wimpy cores: There has been a lot of work on whether small servers can replace high-end platforms in the cloud [25, 46–48]. Despite the power benefits of simple cores, interactive services still achieve better latency in servers that optimize for single-thread performance. Microservices offer an appealing target for simple cores, given the small amount of computation per microservice. We evaluate low-power machines in two ways. First, we use RAPL on our local cluster to reduce the frequency at which all microservices run. Fig. 12 (top row) shows the change in tail latency as load increases, and as the operating frequency decreases for five popular, open-source single-tier interactive services: nginx,
mencached, MongoDB, Xapian, and Recommender. We compare these against the five end-to-end services (bottom row).

As expected, most interactive services are sensitive to frequency scaling. Among the monolithic workloads, MongoDB is the only one that can tolerate almost minimum frequency at maximum load, due to it being I/O-bound. The other four single-tier services experience increased latency as frequency drops, with Xapian being the most sensitive [51], followed by nginx, and mencached. However, looking at the same study for the microservices reveals that, despite the higher tail latency of the end-to-end service, microservices are much more sensitive to poor single-thread performance than traditional cloud applications. Although initially counterintuitive, this result is not surprising, given the fact that each individual microservice must meet much stricter tail latency constraints compared to an end-to-end monolith, putting more pressure on performance predictability. Out of the five end-to-end services (we omit Swarm-Edge, since compute happens on the edge devices), the Social Network and E-commerce are most sensitive to low frequency, while the Swarm service is the least sensitive, primarily because it is bound by the cloud-edge communication latency, as opposed to compute speed.

Apart from frequency scaling, there are platforms designed with low-power cores to begin with. We also evaluate the end-to-end services on two Cavium ThunderX boards (2 sockets, 48 in-order cores per socket, 1.8GHz each, and a 16-way shared 16MB LLC) [25]. The boards are connected on the same ToR switch as the rest of our cluster, and their memory and network sub-systems are the same as the other servers. Fig. 13 shows the throughput at the saturation point for each application on the two platforms. We also show the performance of the Xeon server when equalizing its frequency to the Cavium board. Although ThunderX is able to meet the end-to-end QoS target at low load, all five applications saturate much earlier than on the high-end server. This is especially the case in Social Network, and Media Service because of their stricter latency requirements, and E-commerce, because it is more compute intensive. As with power management, Swarm does not suffer as much, because it is network-bound. Running the Xeon server at 1.8GHz, although worse than its performance at the nominal frequency, still outperforms the Cavium SoC considerably. Even though low power machines degrade performance in this case, they can still be used for microservices off the critical path, or those insensitive to frequency scaling.

5 OS & Networking Implications

We now examine the role of operating systems and networking under the new microservices model.

OS vs. user-level breakdown: Fig. 14 shows the breakdown of cycles (C) and instructions (I) to kernel, user, and libraries for each of the end-to-end services. For all applications, and especially Social Network and Media Service, a large fraction of execution is at kernel mode, skewed by the use of mencached for in-memory caching [57], and the high network traffic, with an almost equal fraction going towards libraries like libc, libgcc, libstdc, and libpthread. The breakdown is less skewed for E-commerce and Banking, whose microservices are more computationally-intensive, and spend more time in user mode, while Swarm, both in its cloud and especially edge configurations, spends almost half of the time in libraries.

The large number of cycles in the kernel is not surprising, given that applications like mencached and MongoDB spend most of their execution time in the kernel to handle interrupts, process TCP packets, and activate and schedule interacting services [57]. The large number of library cycles is also intuitive, given that microservices optimize for speed of development, and hence leverage a lot of existing libraries, as opposed to reimplementing the functionality from scratch. The overhead of general-purpose Linux has motivated a lot of simpler specialized kernels, such as Unikernel [63], which trade off compatibility for improved performance. Similar OS designs are also applicable to single-concerned microservices.

Computation:communication ratio: Fig. 15a shows the time spent processing network requests compared to application computation at low and high load for the microservices in Social Network. Fig. 15b shows the fraction of tail latency spent processing RPC requests for the remaining end-to-end services. At low load, RPC processing corresponds to 5-75% of execution time across the Social Network’s microservices, and 18% of end-to-end tail latency. This is caused by several microservices being too simple to involve considerable processing. In comparison, network processing accounts for a lower fraction of latency in E-commerce and Banking, primarily because their microservices are more computationally intensive. Finally, network processing accounts for over 30% of tail latency in both Swarm settings, even at low load.

At high load, network processing becomes a much more pronounced factor of tail latency for all end-to-end services, except for E-commerce, and Banking, as long queues build up in the NICs. This has a significant impact on tail latency,
with the Social Network experiencing a 3.2x increase in end-to-end tail latency. The large impact of network processing occurs regardless of whether microservices communicate over RPCs (Social Network, Media Service, Banking), or over HTTP (E-commerce, Swarm-Edge), although RPCs introduce considerably lower latencies at low load than HTTP. Finally, Fig. 15a also shows the time the monolithic Social Network application spends processing network requests. Both at low, and especially at high load the difference is dramatic, albeit justified, since monoliths are deployed as single binaries, with the majority of the network traffic corresponding to client-server communication.

Given the prominent role network processing has on tail latency, we now examine its potential for acceleration.

We use a bump-in-the-wire setup, seen in Fig. 16a, and similar to the one in [38] to offload the entire TCP stack [54, 69, 70, 74, 75] on a Virtex 7 FPGA using Vivado HLS. The FPGA is placed between the NIC and the top of rack switch (ToR), and is connected to both with matching transceivers, acting as a filter on the network. We maintain the PCIe connection between the host and the FPGA for accelerating other services, such as the machine learning models in the recommender engines, during periods of low network load. Fig. 16b shows the speedup from acceleration on network processing latency alone, and on the end-to-end latency of each of the services. Network processing latency improves by 10 – 68x over native TCP, while end-to-end tail latency improves by 43% and up to 2.2x. For interactive, latency-critical services, where even a small improvement in tail latency is significant, network acceleration provides a major boost in performance.

6 Cluster Management Implications

Microservices complicate cluster management, because dependencies between tiers can introduce backpressure effects, leading to system-wide hotspots [56, 59, 82, 85, 87]. Back-pressure can additionally trick the cluster manager into penalizing or upsizing a saturated microservice, even though its saturation is the result of backpressure from another, potentially not-saturated service. Fig. 17 highlights this issue for a simplified two-tier application consisting of a web-server (nginx), and an in-memory caching key-value store (memcached). In case A, as the client issues read requests, nginx reaches saturation, causing its latency to increase rapidly, and long queues to form in its input. This is a straightforward case, which autoscaling systems can easily tackle by scaling out nginx, as seen in the figure at $t = 14s$ and $t = 35s$.

Case B on the other hand, highlights the challenges of backpressure. When using HTTP1, requests within a single connection are blocking, i.e., there can only be one outstanding request per connection across tiers. Therefore, even though memcached itself is not saturated, it causes long queues of outstanding requests to form ahead of nginx, which in turn cause it to saturate. Current cluster managers cannot easily address this case, as a utilization-based autoscaling scheme would scale out nginx, which is busy waiting and appears saturated. As seen in the figure, not only does this not solve the problem, but can potentially make it worse, by admitting even more traffic into the system. Even without the connection blocking in HTTP1,
backpressure still occurs, as multi-tier applications are not perfect pipelines where tiers operate entirely independently.

Unfortunately real-world cloud applications are much more complex than this simple example suggests. Fig. 18 shows the microservices dependency graphs for three major cloud service providers, and for one of our applications (Social Network). The perimeter of the circle (or sphere surface) shows the different microservices, and edges show dependencies between them. Such dependencies are difficult for developers or users to describe, and furthermore, they change frequently, as old microservices are swapped out and replaced by newer services.

Fig. 19 shows the impact of cascading QoS violations in the Social Network service. Darker colors show tail latency closer to nominal operation for a given microservice in Fig. 19a, and low utilization in Fig. 19b. Brighter colors signify high per-microservice tail latency and high CPU utilization. Microservices are ordered based on the service architecture, from the back-end services at the top, to the front-end at the bottom. Fig. 19a shows that once the back-end service at the top experiences high tail latency, the hotspot propagates to its upstream services, and all the way to the front-end. Utilization in this case can be misleading. Even though the saturated back-end services have high utilization in Fig. 19b, microservices in the middle of the figure also have even higher utilization, without this translating to QoS violations.

Conversely, there are microservices with relatively low utilization and degraded performance, for example, due to waiting on a blocking/synchronous request from another, saturated tier. This highlights the need for cluster managers that account for the impact dependencies between microservices have on end-to-end performance when allocating resources.

Finally, the fact that hotspots propagate between tiers means that once microservices experience a QoS violation, they need longer to recover than traditional monolithic applications, even in the presence of autoscaling mechanisms, which most cloud providers employ. Fig. 20 shows such a case for Social Network implemented with microservices, and as a monolith in Java. In both cases the QoS violation is detected at the same time. However, while the cluster manager can simply instantiate new copies of the monolith and rebalance the load, autoscaling takes longer to improve performance. This is because, as shown in Fig. 20b, the autoscaler simply upsizes the resources of saturated services - seen by the progressively darker colors of highly-utilized microservices. However, services with the highest utilization are not necessarily the culprits of a QoS violation [61], taking the system much longer to identify the correct source behind the degraded performance and upsizing it. As a result, by the time the culprit is identified, long queues have already built up which take considerable time to drain.

7 Application & Programming Framework Implications

Latency breakdown per microservice: We first examine whether the end-to-end services experience imbalance across tiers, with some microservices being responsible for a disproportionate amount of computation or end-to-end latency, or being prone to creating hotspots. We examine each service at low and high load and obtain the per-microservice latency using our distributed tracing framework, and confirm it with Intel’s vTune. Both for the Social Network and Media Service latency at low load is dominated by the front-end (nginx), while the rest of the microservices are almost evenly distributed. MongoDB is the only exception, accounting for 8.5% and 10.3% of end-to-end latency respectively.

This picture changes at high load. While the front-end still contributes considerably to latency, overall performance is now limited by the back-end databases, and the microservices that manage them, e.g., writeGraph. The Ecommerce and Banking services experience similar fluctuations across load levels, and are additionally impacted by the fact that several of their services are compute intensive, and written in high-level languages, like node.js and Go. This affects execution time, with orders, catalogue, and payment accounting for the majority of end-to-end latency for Ecommerce, and payments and authentication for Banking. The back-end databases in this case contribute less to execution time, showing that the choice of programming language affects how hotspots evolve in the system. queueMaster also experiences high latency in E-commerce, as it uses synchronization to ensure that orders are serialized, processed, and committed in order, which constrains its scalability at high load.
Finally, the *Swarm coordination* service experiences different trade-offs when running on the cloud compared to the edge devices. While *image Recognition* dominates latency regardless of where the microservice is running, its impact on tail latency is more severe when running at the resource-limited edge, to the point of preventing the motion controller from engaging, due to insufficient resources.

This shows that not only bottlenecks vary across end-to-end services, despite individual microservices being same/similar, but that these bottlenecks additionally change with load, putting more pressure on dynamic and agile management.

**Serverless frameworks**: Microservices are often used in the context of serverless programming frameworks, i.e., frameworks where the application and data are managed by the cloud provider, and the user simply launches short-lived "functions", and is charged on a per-request basis [3]. Serverless is well-suited for applications with intermittent activity, where maintaining long-running instances is cost inefficient. Serverless additionally targets embarrassingly parallel services, which benefit from a massive amount of resources for a brief period of time. At the same time, serverless adds an extra level of indirection, as applications have to be instrumented (or re-written) to interface with the serverless framework [8, 14]. Additionally, since serverless functions are ephemeral, data has to be stored in persistent storage for subsequent functions to operate on it. On AWS Lambda the output of functions is stored in S3, which can introduce significant overheads compared to in-memory computation.

Fig. 21 (top) shows the performance and cost of each end-to-end service on traditional containers on Amazon EC2 versus AWS Lambda functions. Each microservice is instrumented to interface with Lambda's API. For a number of microservices written in languages that are not currently supported by Lambda, we also had to reimplement the microservice logic. In the case of EC2, each service uses between 20-64 m5.12xlarge instances. We run each service for 10 minutes. The margins of box plots show the 25th and 75th latency percentiles, while the whiskers show the 5th and 95th. In Lambda, we show performance and cost both for the default persistent storage (S3), and for a configuration that uses the memory of four additional EC2 instances to maintain intermediate state passed through dependent microservices.

Latency is considerably higher for Lambda when using S3, primarily due to the overhead and rate limiting of the remote persistent storage. This occurs even though the amount of data transferred between microservices is small, to adhere to the design principle that microservices should be mostly stateless [18]. The majority of this overhead disappears when using remote memory to pass state between dependent serverless functions. Even in this case though, performance variability is higher in Lambda, as functions can be placed anywhere in the datacenter, incurring variable network latencies, and suffering interference from external functions co-scheduled on the same physical machines (EC2 instances are dedicated to our services). Note that even in the EC2 scenario, dependent microservices are placed on different physical machines to ensure a fair comparison in terms of network traffic. On the other hand, cost is almost an order of magnitude lower for Lambda, especially when using S3, as resources are only charged on a per-request basis.

The bottom of Fig. 21 highlights the ability of serverless to elastically scale resources on demand. The input load is real user traffic in *Social Network*, which follows a diurnal pattern. In the interest of cost, we have compressed the load pattern to a shorter period of time and replayed it using our open-loop workload generator. Even though EC2 experiences lower tail latency than Lambda during low load periods, consistent with the findings above, when load increases, Lambda adjusts resources to user demand faster than EC2. This is because the increased number of requests translates to more Lambda functions without requiring the user to intervene. In comparison, in EC2, we use an autoscaling mechanism that examines utilization, and scales allocations by requesting extra instances, when it exceeds a pre-determined threshold (70% in this case, consistent with EC2 default autoscaler [20]). This has a negative impact on latency, since the system waits for load to increase substantially before employing additional resources, and initializing new resources is not instantaneous. For microservices to reach the potential serverless offers, they need to remain mostly stateless, and leverage in-memory primitives to pass data between dependent functions.

### 8 Tail At Scale Implications

We now focus on the *Social Network* service to study the tail at scale effects of microservices, i.e., effects that occur because of the large-scale of systems and applications [28]. We *The Social Network* has several hundred registered users, and 165 active daily users on average. The input load for this
Impact of slow servers: Fig. 22c shows the impact a small number of slow servers has on overall QoS as cluster size increases. We purposely slow down a small fraction of servers by enabling aggressive power management, which we already saw is detrimental to performance (Sec. 4). For large clusters (>100 instances), when 1% or more of servers behave poorly, the goodput is almost zero, as these servers host at least one microservice on the critical path, degrading QoS. Even for small clusters (40 instances), a single slow server is the most the service can sustain and still achieve some QPS under QoS. Finally, we compare the impact of slow servers in clusters of equal size for the monolithic design of Social Network. In this case goodput is higher, even as cluster sizes grow, since a single slow server only affects the instance of the monolith hosted on it, while the other instances operate independently. The only exception are back-end databases, which even for the monolith are shared across application instances, and sharded across machines. If one of the slow servers is hosting a database shard, all requests directed to that instance are degraded. In general, the more complex an application’s microservices graph, the more impactful slow servers are, as the probability that a service on the critical path will be degraded increases.

9 Conclusions
We have presented DeathStarBench, an open-source suite for cloud and IoT microservices. The suite includes representative services, such as social networks, video streaming, e-commerce, and swarm control services. We use DeathStarBench to study the implications microservices have across the cloud system stack, from datacenter server design and hardware acceleration, to OS and networking overheads, and cluster management and programming framework design. We also quantify the tail-at-scale effects of microservices as clusters grow in size, and services become more complex, and show that microservices put increased pressure in low tail latency and performance predictability.

DeathStarBench Release
The applications in DeathStarBench are publicly available at: http://microservices.ece.cornell.edu under a GPL licence. We welcome feedback and suggestions, and hope that by releasing the benchmark suite publicly, we can encourage more work in this emerging field.

Acknowledgements
We sincerely thank Christos Kozyrakis, Daniel Sanchez, David Lo, as well as the academic and industrial users of the benchmark suite, and the anonymous reviewers for their feedback on earlier versions of this manuscript. This work was supported in part by NSF grant NeTS CSR-1704742, a Facebook Faculty Research Award, a John and Norma Balen Sesquicentennial Faculty Fellowship, and generous donations from Google Compute Engine, Windows Azure, and Amazon EC2.