the morning paper

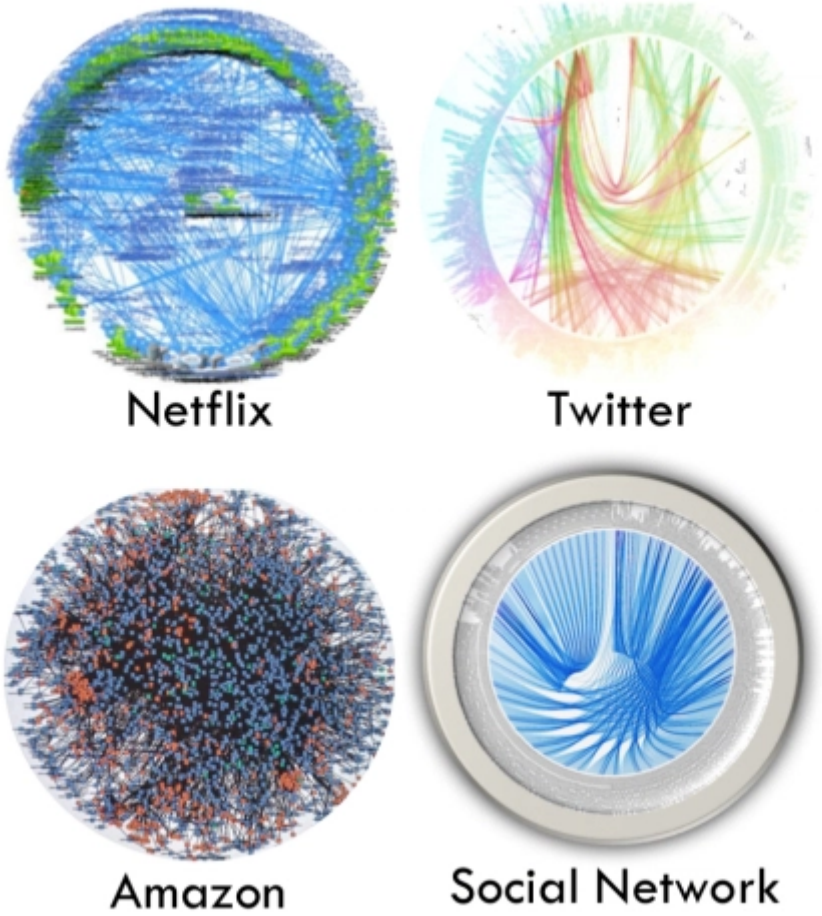
a random walk through Computer Science research, by Adrian Colyer

An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems

MAY 13, 2019MAY 13, 2019

An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems (http://www.csl.cornell.edu/~delimitrou/papers/2019.asplos.microservices.pdf) Gan et al., *ASPLOS'19*

Microservices are well known for producing 'death star' interaction diagrams like those shown below, where each point on the circumference represents an individual service, and the lines between them represent interactions.



Systems built with lots of microservices have different operational characteristics to those built from a small number of monoliths, we'd like to study and better understand those differences. That's where 'DeathStarBench' comes in: a suite of five different microservices-based applications (one of which, a drone coordination platform called Swarm has two variations – one doing most compute in the cloud, and one

offloading as much as possible to the edge). It's a pretty impressive effort to pull together and make available in open source (http://microservices.ece.cornell.edu/) (not yet available as I write this) such a suite, and I'm sure explains much of the long list of 24 authors on this paper.

The suite is built using popular OSS applications and representative technologies, deliberately using a mix of languages (C/C++, Java, Javascript, node.js, Python, Ruby, Go, Scala, …) and both RESTful and RPC (Thrift, gRPC) style service interfaces. There's a nice nod to the Weave Sockshop (https://microservices-demo.github.io/) microservices sample application here too.

Service	Total New LoCs	Comm. Protocol	LoCs for RPC/REST Handwritten	LoCs for RPC/REST Autogen	Unique Microservices	Per-language LoC breakdown (end-to-end service)
Social Network	15,198	RPC	9,286	52,863	36	34% C, 23% C++, 18% Java, 7% node.js, 6% Python, 5% Scala, 3% PHP, 2% Javascript, 2% Go
Movie Reviewing	12,155	RPC	9,853	48,001	38	30% C, 21% C++, 20% Java, 10% PHP, 8% Scala, 5% node.js, 3% Python, 3% Javascript
E-commerce Website	16,194	REST RPC	4,798 2,658	- 12,085	41	21% Java, 16% C++, 15% C, 14% Go, 10% Javascript, 7% node.js, 5% Scala, 4% HTML, 3% Ruby
Banking System	13,876	RPC	4,757	31,156	34	29% C, 25% Javascript, 16% Java, 16% node.js, 11% C++, 3% Python
Swarm Cloud	11,283	REST RPC	2,610 4,614	- 21,574	25	36% C, 19% Java, 16% Javascript, 14% node.js, 13% C++, 2% Python
Swarm Edge	13,876	REST	4,757	-	21	29% C, 25% Javascript, 16% Java, 16% node.js, 11% C++, 3% Python

Table 1. Characteristics and code composition of each end-to-end microservices-based application.

A typical architecture diagram for one of these services looks like this:

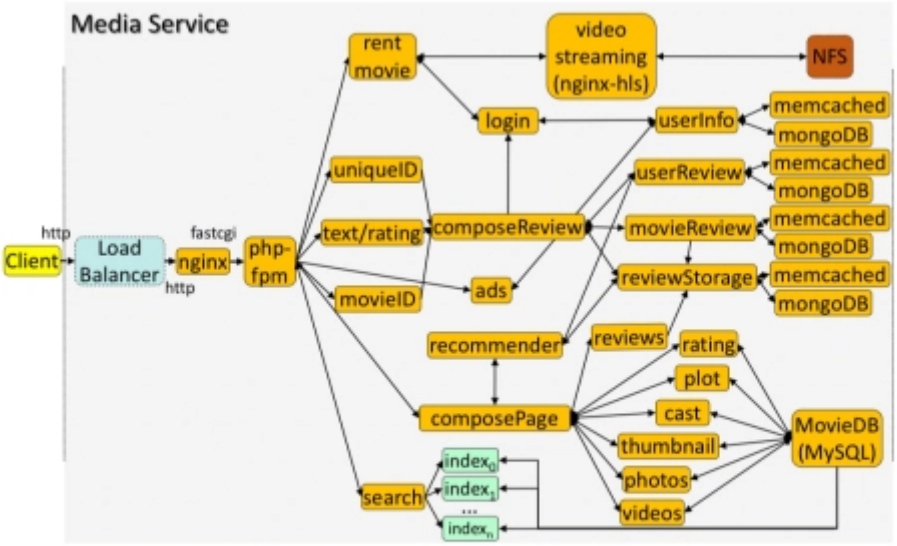


Figure 5. The architecture of the *Media Service* for reviewing, renting, and streaming movies.

Suitably armed with a set of benchmark microservices applications, the investigation can begin!

> 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.

The paper examines the implications of microservices at the hardware, OS and networking stack, cluster management, and application framework levels, as well as the impact of tail latency.

Hardware implications

> 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.

Smaller microservices demonstrated much better instruction-cache locality than their monolithic counterparts. The most prominent source of misses, especially in the kernel, was Thrift. It might also seem logical that these services would be good targets for running on simpler (lower power) cores. The following figure shows that this isn't necessarily the case though. The top line shows the change in tail latency across a set of monolithic applications as operating frequency decreases. The bottom line shows the tail latency impact in the microservices-based applications.

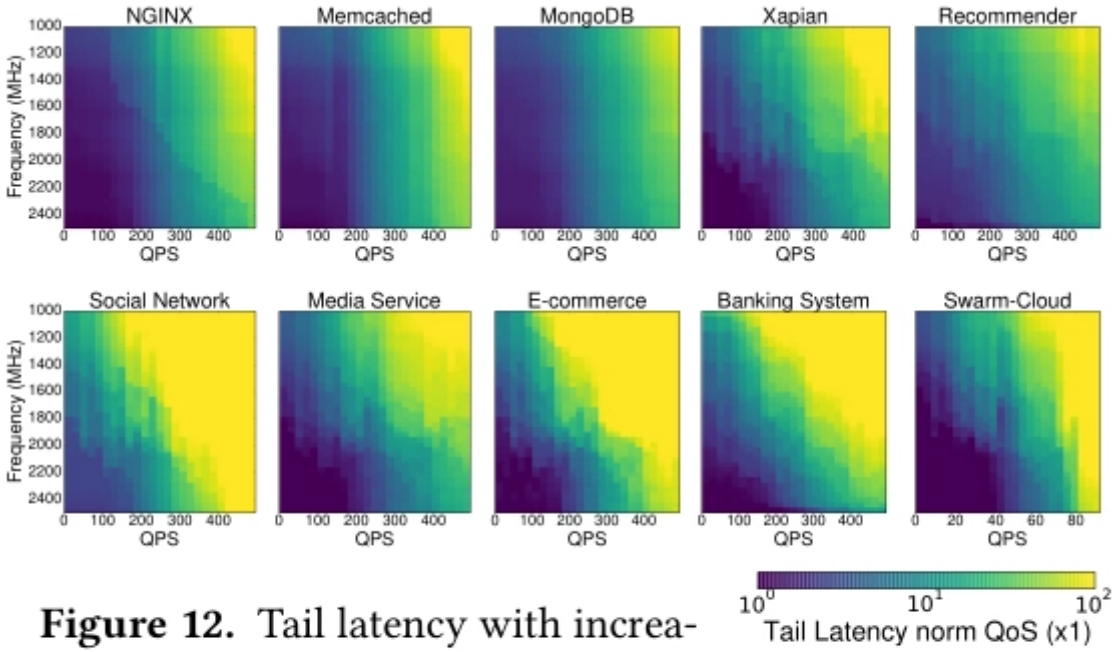


Figure 12. Tail latency with increasing load and decreasing frequency (RAPL) for traditional monolithic cloud applications, and the five end-to-end DeathStarBench services. Lighter colors (yellow) denote QoS violations.

> …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.

Operating system and network implications

Applications built in a microservices style spend a good amount of time in the kernel, as shown in the following figure:

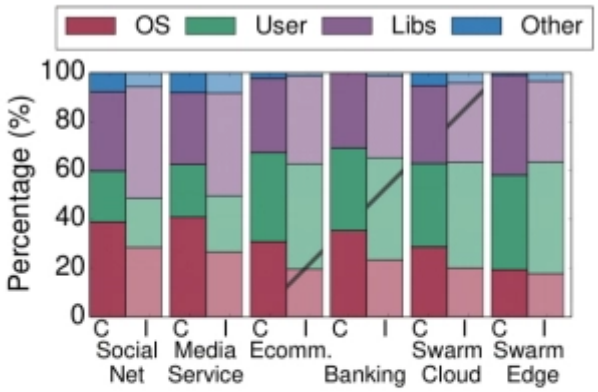
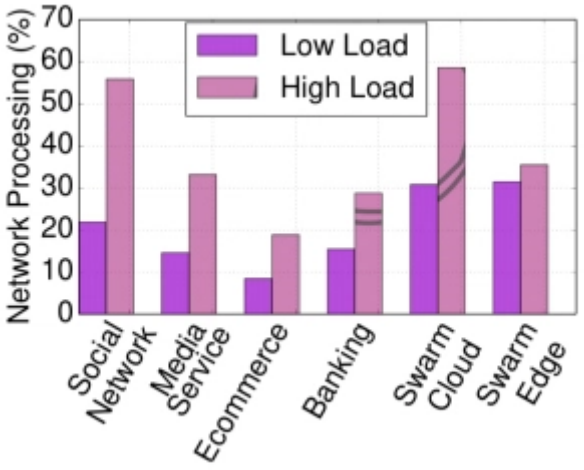


Figure 14. Time in kernel mode, user mode, and libraries for each service.

Included in this count though is the time that components such as MongoDB and memcached spend in the kernel handling interrupts, processing TCP packets, and activating and scheduling idling interactive services.

> Unlike monolithic services… microservices spend much more time sending and processing network requests over RPCs or other REST APIs.

This is most pronounced when under high loads, and becomes a very significant tail latency factor. This occurs both with HTTP and RPC-based services.



Microservices communication patterns and the pressure they put on network processing make network acceleration an interesting avenue of investigation. Introducing an FPGA-based offload improved network processing latency by 10-68x over native TCP, with end-to-end tail latency (what %-ile??) improving 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.

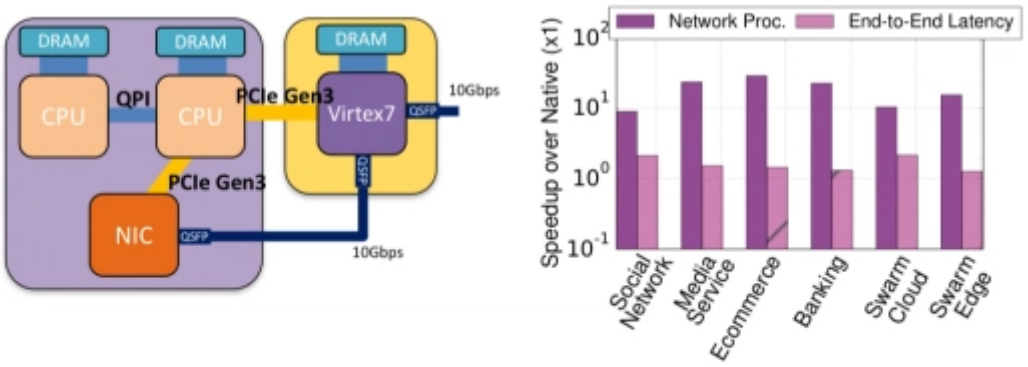


Figure 16. (a) Overview of the FPGA configuration for RPC acceleration, and (b) the performance benefits of acceleration in terms of network and end-to-end tail latency.

Cluster management implications

Each microservice may be individually simpler, but at least the essential complexity in the application had to go somewhere. And the place it goes is in the interactions between services, and interestingly, as this next section of the paper shows, *in the interactions between your services and the platform they run on*. We trade off code complexity for a new kind of emergent system complexity.

> Microservices significantly complicate cluster management. Even though the cluster manager can scale out individual microservices on-demand instead of the entire monolith, dependencies between microservices introduce back-pressure effects and cascading QoS violations that quickly propagate through the system, making performance unpredictable.

Consider the two scenarios shown in the following figure:

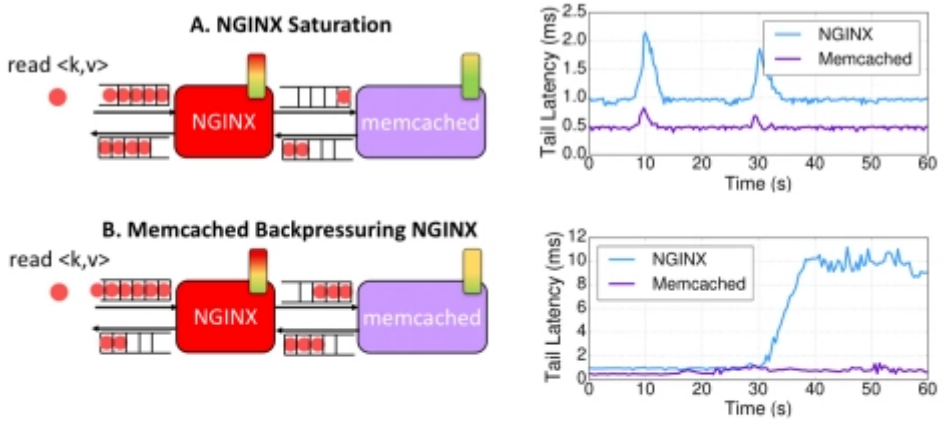


Figure 17. Example of backpressure between microservices in a simple, two-tier application. Case A shows a typical hotspot that autoscalers can easily address, while Case B shows that a seemingly negligible bottleneck in memcached can cause the front-end NGINX service to saturate.

In the top line (case A), nginx is reaching saturation as the client issues read requests causing long queues to firm in its inputs. Autoscaling systems can handle this by scaling out nginx. In the bottom line though (case B), we have blocking HTTP/1 requests causing memcached to only support one outstanding request per connection across tiers. Long queues again form at nginx, but scaling out nginx isn't going to make this situation any better, and may well make it worse.

Real-world applications are of course much more complex than this simple two-tier example. The following plots study cascading QoS violations in the *SocialNetwork* service. Bright areas in the LH plot show high latency, and in the RH plot high CPU usage. The ordering of services (top to bottom, back-end to front-end) is the same in both cases.

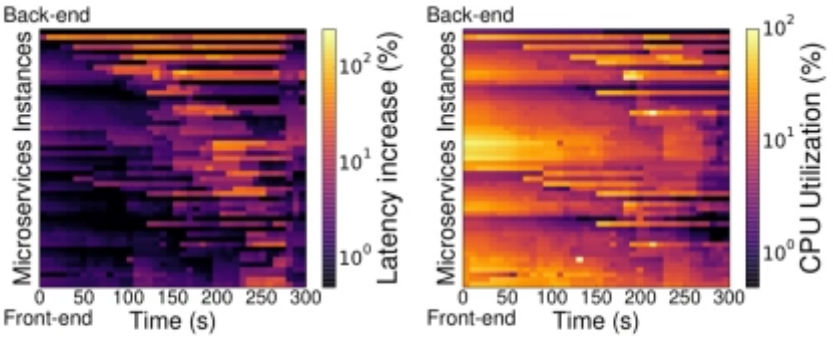


Figure 19. Cascading QoS violations in *Social Network* compared to per-microservice CPU utilization.

Once back-end services have high utilisation this propagates all the way to the front-end, and can put even higher pressure on microservices in the middle. There are also microservices that show relatively low utilisation but still have degraded performance (due to blocking).

The authors draw two conclusions from their analysis here:

- Cluster managers need to account for the impact dependencies have on end-to-end performance when allocating resources (but how will they learn this?)
- 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.

Bottleneck locations also vary with load.

Application and programming framework implications

In section 7, the authors evaluate the performance and cost of running each of the five microservices systems on AWS Lambda instead of EC2. Services were run for 10 minutes with a varying input load. Two versions of the Lambda implementation were compared: one using S3 for communication of state between functions, and one that uses the memory of four additional EC2 instances (DynamoDB or SQS feel like more natural serverless choices here, assuming the state is not generally large?).

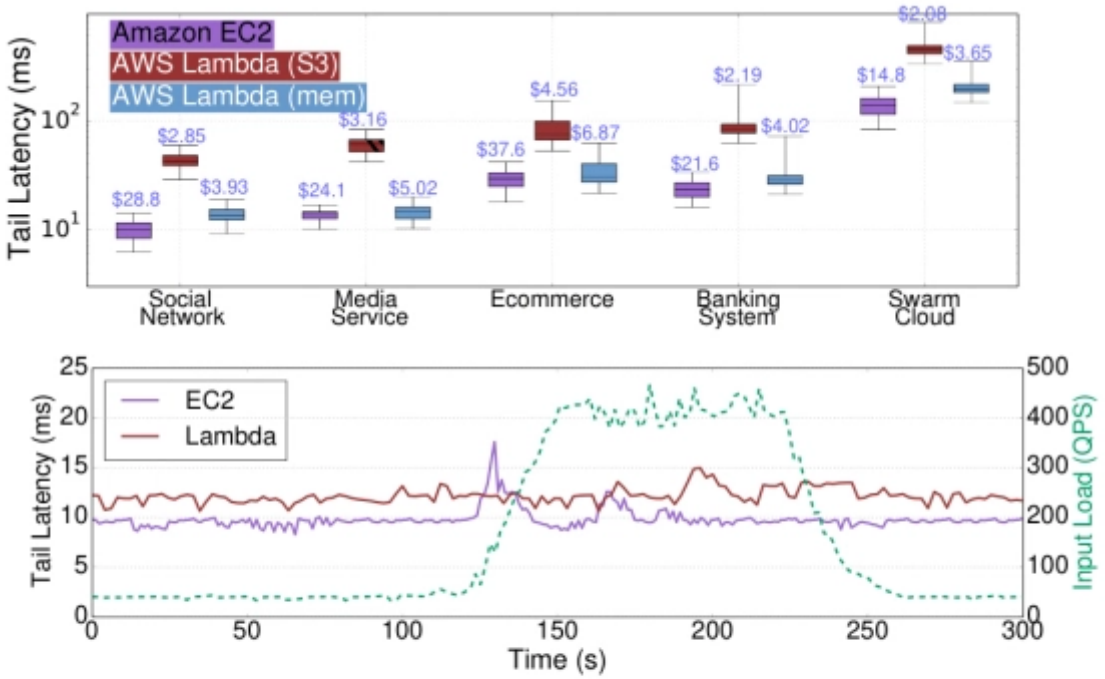


Figure 21. Performance and cost for the five services on Amazon EC2 and AWS Lambda (top). Tail latency for *Social Network* under a diurnal load pattern (bottom).

Lambda showed higher performance variability, but for these workloads cost was almost an order of magnitude lower, and Lambda adjusted resources to user demand more quickly.

The conclusion in this section is that microservices in a serverless world should remain mostly stateless, and leverage in-memory primitives to pass data between dependent functions (do you mean e.g. Redis???). I'd prefer to see either DynamoDB or SQS explored in an AWS context, and a move towards an asynchronous event-driven model rather than having functions call each other.

Tail at scale implications

> 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.

The increased sensitivity in the tail comes up in multiple places in the paper. Section 8 studies this phenomenon explicitly. The general finding is that the more complex the interaction graph of a microservices application, the more impactful slow services are as the probability that a service on the critical path will be degraded increases. Or as we learned a long time ago in 'The Tail at Scale (https://blog.acolyer.org/2015/01/15/the-tail-at-scale/)', if you call more things, you've got more chances of one of them being slow.

All of the applications from DeathStarBench are (will be) made available under a GPL license at http://microservices.ece.cornell.edu (http://microservices.ece.cornell.edu/).

DISTRIBUTED SYSTEMS PERFORMANCE

7 thoughts on “An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems”

1. **simbo1905**
   SAYS:
   MAY 13, 2019 AT 8:06 AM
   Broken link “[The Tail at Scale][TailAtScale”

   Reply
2. **simbo1905**
   SAYS:
   MAY 13, 2019 AT 8:09 AM
   Broken link “The Tail at Scale”

   Reply
   1. **adriancolyer**
      SAYS:
      MAY 13, 2019 AT 8:53 AM
      Fixed, thank you!

      Reply
3. Pingback: Seer: leveraging big data to navigate the complexity of performance debugging in cloud microservices | the morning paper
4. Pingback: RPCValet: NI-driven tail-aware balancing of µs-scale RPCs | the morning paper
5. **Alex Lowe**
   SAYS:
   MAY 20, 2019 AT 4:47 PM
   It appears that there is a typo in the middle of the post. But the results are very interesting, if somewhat expected. It’s nice to see the difficulty in scaling up microservices.

   Typo in this sentence: “In the top line (case A), nginx is reaching saturation as the client issues read requests causing long queues to **firm** in its inputs.”

   Reply
6. **Suman Sunderroy**
   SAYS:
   MAY 25, 2019 AT 11:57 AM
   It appears that microservices are replicating many of the same failure modes as microkernels. So when are we going to have a Torvalds-Tanenbaum debate?

   Reply