Ampere® Altra® Family 64-Bit Multi-Core Processors

Web Services Efficiency – Reference Architecture

May 10, 2023

Document Issue 1.00

AMPERE®

Contents

AMPERE®

Contents (continued)

Figures

Tables

1. About Web Services

Any software application that provides a standardized way for machine-to-machine communication over a network can be considered a web service. Web services use standard protocols and data formats to allow systems and applications to interoperate to serve a variety of purposes. These include web hosting, social media integrations, data access and management, facilitating e-commerce transactions and many more.

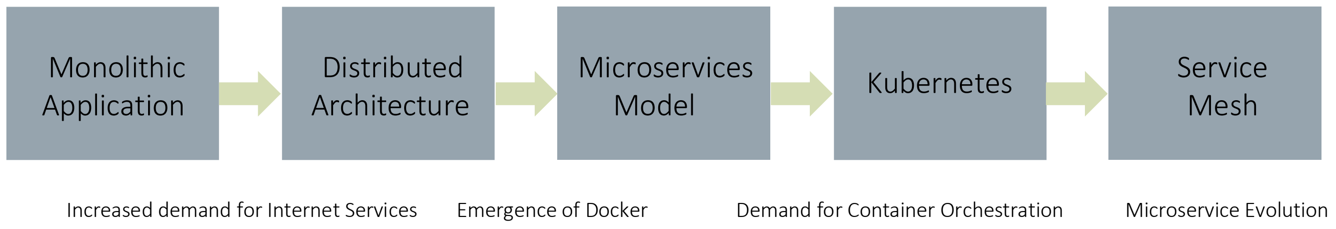
Initially, web services were deployed on servers hosted in datacenters, and specific servers were dedicated to specific applications. Over time, virtualization became popular, which meant multiple applications could now be hosted on the same server. However, hosting providers still needed to manage and maintain the servers in their own datacenters, which soon became expensive as well as difficult to scale. This general trend gave birth to the era of Cloud Computing.

As cloud adoption increased, challenges with getting certain applications to run in a cloud-based deployment became evident. The “lift and shift” method generally worked but it really didn’t leverage all of the benefits of “being in the cloud”. Applications needed to be rearchitected from the ground up. Monolithic codebases gave way to microservices, an approach where individual microservices are scaled and orchestrated together to deliver the desired end service. This method optimizes resource utilization, increases manageability and ease of deployment. Further, individual services can be ported to run on different platforms quickly. Eventually, containerization and virtualization became popular methods for deployment and management of microservices, and more companies started to integrate tools like Docker and Kubernetes into their infrastructure.

Most modern web services are built using a microservices architecture and make use of several cloud-native technologies, whether they are deployed on-premises in a data center or in the cloud.

This evolution is simplified and depicted in *Figure 1*.

Figure 1: Evolution of Web Services



1.1 Scope and Audience

In this document, we showcase how to deploy and scale a real-world, microservice based, end-to-end Web Service application as modeled by *DeathStarBench:socialNetwork* on Ampere® Altra® Family Cloud Native Processor systems. This application suite implements a broadcast-style social network with unidirectional follow relationships similar to Twitter or Facebook. We also demonstrate how the social network application can be scaled to a multi-node Ampere Altra Max cluster using Kubernetes to allow more user requests per second (RPS) without compromising response times. Finally, we measure the maximum performance delivered under load on a multi-node scale-out Ampere Altra Max cluster and compare the performance to a similar setup using x86 based systems.

This document is a case study that shows sales engineers, cloud service providers, IT Planners, cloud architects, and end-users how to deploy and manage modern microservice based web services on Ampere processors. Scaling out the web service deployment beyond a cluster to the rack level shows how organizations can increase compute capacity in every rack and decrease the number of racks required to run complex web services on Ampere Arm-based servers.

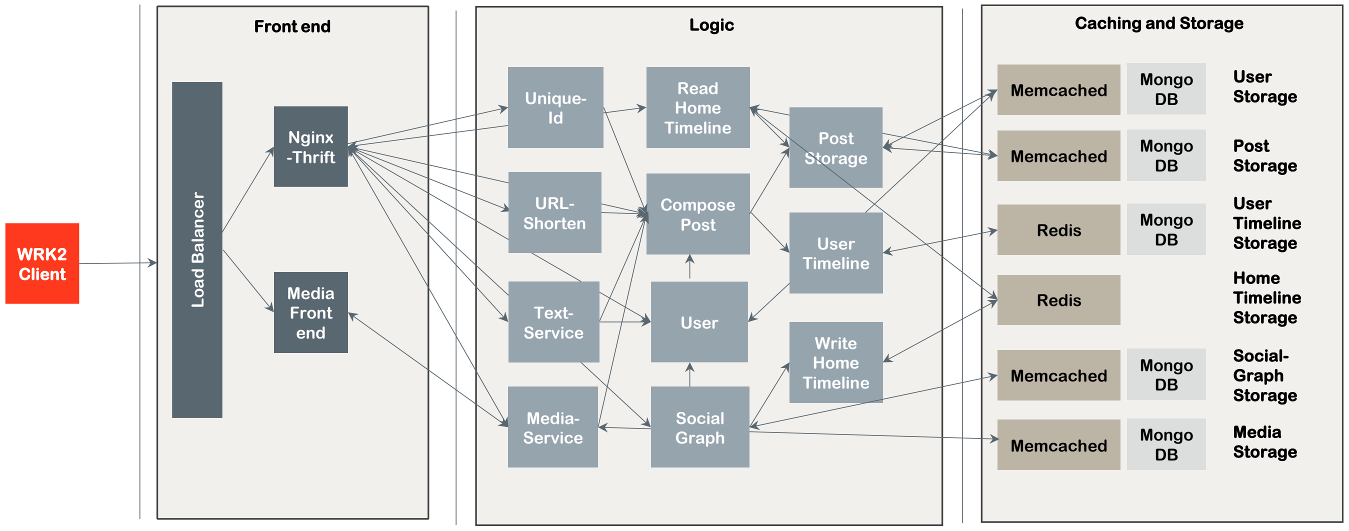
2. DeathStarBench/socialNetwork Architecture

The DeathStarBench/socialNetwork suite is built using popular open-source applications (such as NGINX, Memcached, Redis and MongoDB) commonly deployed by cloud service providers. The application is implemented with loosely coupled microservices communicating with each other via Thrift RPCs and written in various languages including C/C++, Python and Lua Scripts.

Users (clients) send requests over http, which first reach a load balancer, which is implemented using NGINX. Once NGINX selects the specific web server, it uses a Lua module to talk to the microservices responsible for composing and displaying posts. The service backend uses Memcached/Redis for caching, and MongoDB for persistent storage for posts, profiles, media, and recommendations. Finally, the service is instrumented with a distributed tracing system, which records the latency of each network request and per-microservice processing.

For benchmarking, this deployment uses WRK2 as the load generator to simulate users connecting to the social network website. *Figure 2* depicts the entire architecture.

Figure 2: Social Network Application Components



2.1 DeathStarBench/socialNetwork Components

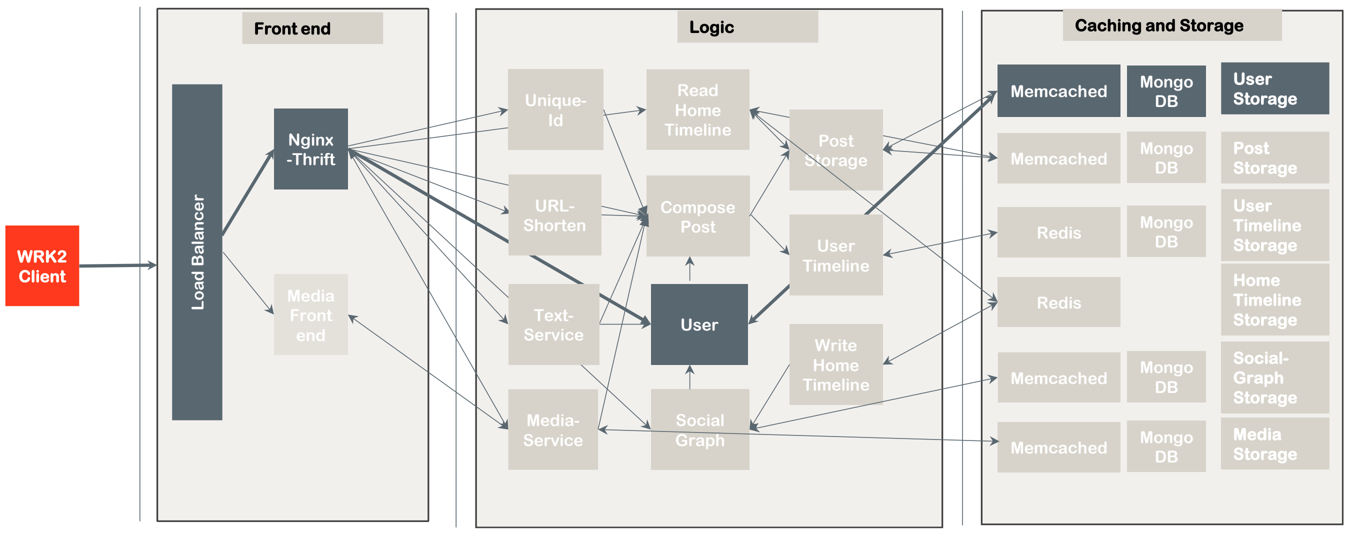
2.1.1 Web Frontend: NGINX

NGINX is an open source, high performance HTTP server and reverse proxy with many other web service-related features bundled. It is often used as load balancer in the cloud. NGINX implements event-driven architecture to handle incoming requests. It is built to offer a low memory footprint and high concurrency. NGINX is the most popular web server among high-traffic websites based on a 2023 netcraft survey.

The social network application in the DeathStarBench (DSB) suite uses OpenResty which is a higher-level application and gateway platform with NGINX as a component. OpenResty integrates an enhanced version of the NGINX core, an enhanced version of LuaJIT, Lua libraries, third-party NGINX modules, and most of their external dependencies. It is designed to help developers easily build scalable web applications, web services, and dynamic web gateways. NGINX provides the web frontend, where users (clients) send requests over http and these requests are sent using the nginx-lua module to the microservices responsible for composing, displaying, and storing posts.

Figure 3 shows the logical flow of the user signup process.

Figure 3: User Signup and Login

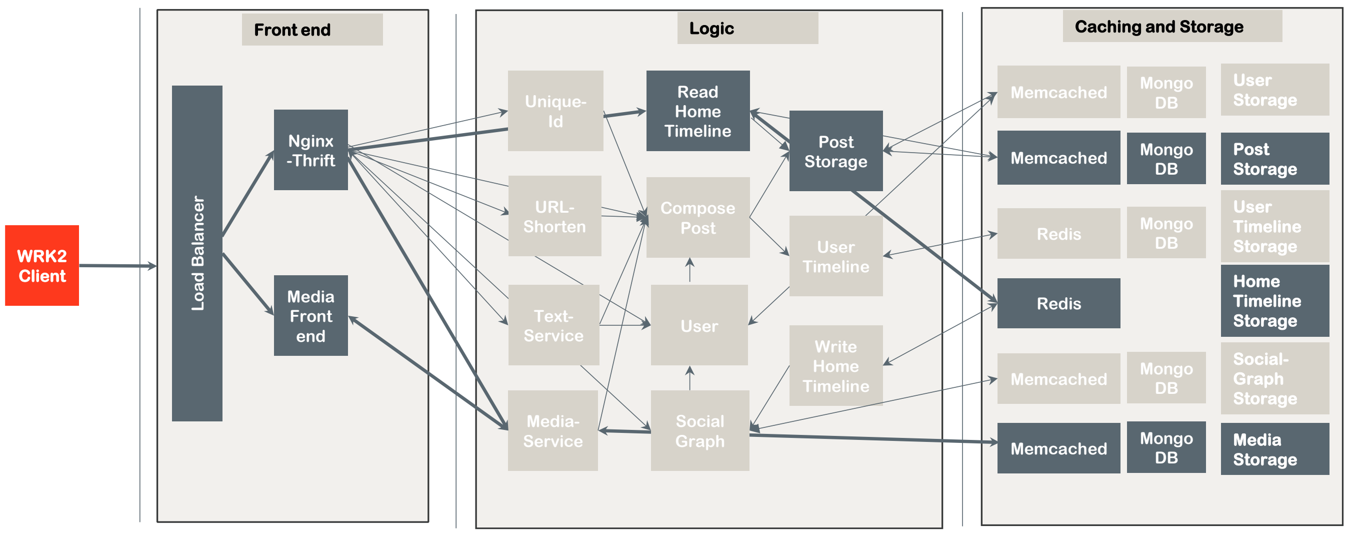


2.1.2 Caching Layer: REDIS

Redis is an open source, in-memory, key-value data store typically used as a database or a cache. It uses an in-memory dataset, but data can persist through periodic writes or appends to disk. Being in-memory, Redis is very fast, and can deliver high throughput at sub-millisecond latencies. It continues to rank highly in popularity among key value stores in the cloud, according to DB-engines.

Redis is used as the caching layer for the Home Timeline Service for the DSB:socialNetwork. This service lists the status messages that have been posted by a specific user and all the people they are following on their home page. As the landing page for users, this data should be as easy to retrieve as possible. Redis stores this information as a ZSET of status ID/timestamp pairs. Timestamp information provides the sort order, and the status ID is used to fetch the status message data. *Figure 4* shows the workflow for the home timeline service.

Figure 4: Home Timeline Service



2.1.3 Caching Layer: MEMCACHED

Memcached is a well-known, simple, in-memory cache solution and is used by several large companies such as Facebook, Twitter, and Pinterest. The main use case for Memcached is look-aside caching, to reduce the load on the database. As a result, users of a social network – for instance – enjoy lower latency and response times when using the platform.

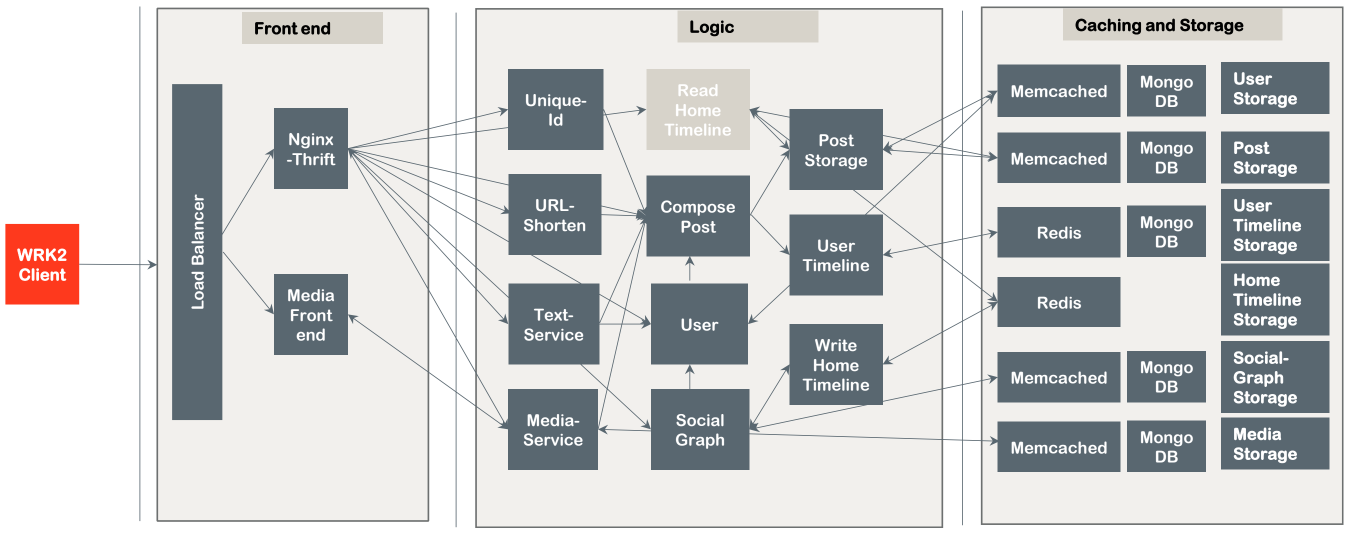
The social network application persists all posts to disk when they are created, but most posts requested by users need to be served out of memory for performance reasons. This application uses Memcached to store recent and frequently accessed posts, and as a result optimizes disk usage.

Unlike Redis, Memcached, known for its simplicity, does not offer any in-built high availability (HA) features. There have been a few open-source Memcached HA solutions (the most popular being Mcrouter by Facebook), but in this case study we decided to run Memcached in default standalone mode. Separate Memcached pods are deployed and used by different services. Scalability is limited and when more replicas of Memcached are running, cached data is split across all pods.

2.1.4 Databases: MONGODB

MongoDB is the pioneer of NoSQL databases, which developed because RDBMS systems based on SQL did not support the scale or rapid development cycles needed for creating modern applications. Instead of storing data in tables of rows or columns like SQL databases, each record in a MongoDB database is a document described in BSON, a binary representation of the data. Applications can then retrieve this information in a JSON format. MongoDB can also handle high volume and can scale both vertically and horizontally to accommodate large data loads.

Figure 5: MongoDB Workflow



The social network service's backend uses MongoDB for persistent storage for user information, posts, profiles, media etc. as shown in *Figure 5* above.

2.1.5 Application Layer: PYTHON, C/C++, LUA

Microservices-based architecture allows each of the many services that are part of the social network application to be written in different languages and programming models, including C/C++, Python, GO etc. Most of these can be easily ported to the Arm architecture without any modifications. All major Linux distributions support Arm and provide an extensive library of common Linux packages built for AArch64. Applications and their dependencies can be recompiled using compilers like GCC which is fully supported.

2.1.6 Workload Generators: WRK2

The DeathStarBench suite comes with a modified workload generator WRK2, which is based on WRK. This is an open-loop implementation of WRK2 which sends out requests according to the schedule regardless of any delay in response for previously sent requests. Any traffic on the server side will be directly reflected in the real latency results. Latency reporting is the same as WRK.

WRK2 uses LuaJIT script to perform HTTP request generation, response processing, and custom reporting. There are 4 different scripts that support the following actions:

- Compose Post (text, media, images, shortened URL etc.)
- Read entire user timeline
- Read home timeline
- Mixed workload comprised of all 3 of the above actions

3. Deployment

3.1 Hardware Configuration

3.1.1 Ampere Servers

Table 1 lists the details of the configuration of the Ampere servers.

Table 1: Ampere Servers BoM (Quantity: 3)

PARAMETER	DESCRIPTION
OEM Model	Gigabyte Mt. Snow (*where to buy*)
Motherboard	MP32-AR1 (single socket)
BMC Firmware	12.60.10
Sockets	1
CPU/Cores/Threads	Ampere Q128-30: • 128 cores / 128 threads • 3.0 GHz • L1d: 8 MiB (128 instances), L1i: 8 MiB (128 instances) • L2:128 MiB (128 instances), L3:16 MiB (1 instance)
Memory	64GB DDR4, 3200 MT/s
NIC	Mellanox ConnectX-4 Lx: • MT27710 Family • 8 GT/s x4 • 25 GbE
Storage	Samsung NVMe 960.2 GB

3.1.2 x86 Servers

Table 2 lists the details of the configuration of the x86 servers.

Table 2: x86 Servers BoM (Quantity: 3)

PARAMETER	DESCRIPTION
OEM Model	Dell PowerEdge R650
Motherboard	PowerEdge R650 Motherboard (dual socket)
BMC Firmware	1.5.4
Sockets	2
CPU/Cores/Threads	Intel Xeon Gold 6342: • 24 cores / 48 threads • 2.8 GHz / 3 .5 GHz • L1d: 2.3 MiB (48 instances) L1i: 1.5 MiB (48 instances) • L2: 60 MiB (48 instances), L3: 72 MiB (2 instances)
Memory	64 GB DDR4, 3200 MT/s

PARAMETER	DESCRIPTION
NIC	Mellanox ConnectX-6 Dx: • MT2892 Family • 16 GT/s x16 • 100 GbE
Storage	Dell DC NVMe 960 GB

3.2 Software Configuration

Table 3 lists the details of the software configuration of both the Ampere and x86 servers.

Table 3: Software Configuration

PARAMETER	DESCRIPTION
Operating System	Ubuntu 22.04.1 LTS 5.15.0-60-generic #64-Ubuntu
Docker	23.0.0
Kubernetes	1.23.16

3.3 Prerequisites

- Docker
- Kubeadm, kubelet
- Python 3.5+ (with asyncio and aiohttp)
- libssl-dev (apt-get install libssl-dev)
- libz-dev (apt-get install libz-dev)
- lua5.1
- luarocks (apt-get install luarocks)
- luasocket (luarocks install luasocket)

Make sure the following ports are accessible:

- 8080 for NGINX frontend
- 16686 for Jaeger

3.4 DeathStarBench/socialNetwork on GitHub

The DeathStarBench open-source benchmarking suite is available as a free software from GitHub at:

https://github.com/delimitrou/DeathStarBench

This repo uses x86 based images from docker hub for the various microservices that are part of the social network application. Before deploying on Ampere Altra or any other Arm-based system, the application images need to be rebuilt for AArch64.

We have ported the DeathStarBench/socialNetwork benchmark to run on AArch64. The AArch64 version can be downloaded from the Ampere Computing GitHub repository at:

https://github.com/AmpereComputing/deathstarbench-ah/tree/arm64-port

3.5 Building AArch64 Images

The social network application uses many cloud-native applications like NGINX, Redis, MongoDB etc. The AArch64 images of these components are available from docker hub. The versions used for this benchmarking are listed in *Table 4*.

Table 4: Versions for Benchmarking

SOFTWARE	VERSION	IMAGE LOCATION
Openresty with NGINX	1.15.8.1rc1	https://openresty.org/download/openresty-1.15.8.1rc1.tar.gz
Redis	6.2.4	docker pull redis:6.2.4
Memcached	1.6.7	docker pull memcached:1.6.7
MongoDB	4.4.6	docker pull mongo:4.4.6

The application layer itself is written in C/C++, Python and so can be easily recompiled to run on AArch64 in a few simple steps as shown below:

```
cd DeathStarBench/socialNetwork/docker/openresty-thrift/
docker build --no-cache -t arm64/openresty-thrift -f xenial/Dockerfile
cd DeathStarBench/socialNetwork/thrift-microservice-deps/
docker build --no-cache -t arm64/thrift-microservice-deps -f cpp/Dockerfile .
cd DeathStarBench/socialNetwork/media-frontend/
docker build --no-cache -t arm64/media-frontend -f xenial/Dockerfile .
```

Next, we need to edit the Dockerfile in the socialNetwork folder to use the new image as builder:

```
cd DeathStarBench/socialNetwork
FROM arm64/thrift-microservice-deps:latest AS builder
docker build --no-cache -t arm64/social-network-microservices -f Dockerfile .
```

Detailed instructions for porting the application to AArch64 are provided in the *Transition and Tuning Guide* available here.

3.6 Deploying the Social Network Application

The DeathStarBench/socialNetwork application can be easily deployed on a Kubernetes cluster using helm charts. For benchmarking on bare metal servers, we use the Kubeadm tool to set up a minimum viable cluster that conforms to best practices. The cluster is comprised of 3 nodes: the control-plane and 2 worker nodes. The control-plane node is also configured to act as a worker node and can schedule pods. We deployed the flannel container network interface (CNI) network add-on for the pods to communicate. All the nodes are connected on an internal network. For this benchmarking, we used Ubuntu 22.04 as the operating system on all servers.

In this application, every microservice has its own helm chart, which is then assembled under a main helm chart. The main helm chart contains global values that apply to all microservices and can be overridden by the values.yaml file for individual microservices.

To deploy using helm charts, start by installing helm on one of the nodes (https://helm.sh). Use the following command to create a default deployment of the Social Network application:

```
cd DeathStarBench/socialNetwork/helm-chart
kubectl create namespace social-network
helm install social-network ./socialnetwork/ -n social-network
```

3.7 Standalone vs Clustered Deployment

As the number of clients simultaneously connecting to a web service deployed on a single server increases, the server will eventually run out of resources and cease performance scaling. The solution is to scale the resources available either by adding more resources to the servers (vertical scaling, or “scaling up”), adding additional servers (horizontal scaling, or “scaling out”) or a combination of both. When the cluster is scaled out to multiple servers, the web service must be configured to ensure the load is evenly distributed across all the servers. The easiest way to do this for the front end and application tier is to replicate the service on multiple servers. Scaling the database tier is more complicated and requires running additional services that allow database sharding and clustering.

There are two ways to deploy the different databases and caching layers that are part of the social network application.

3.7.1 Standalone Mode (Default)

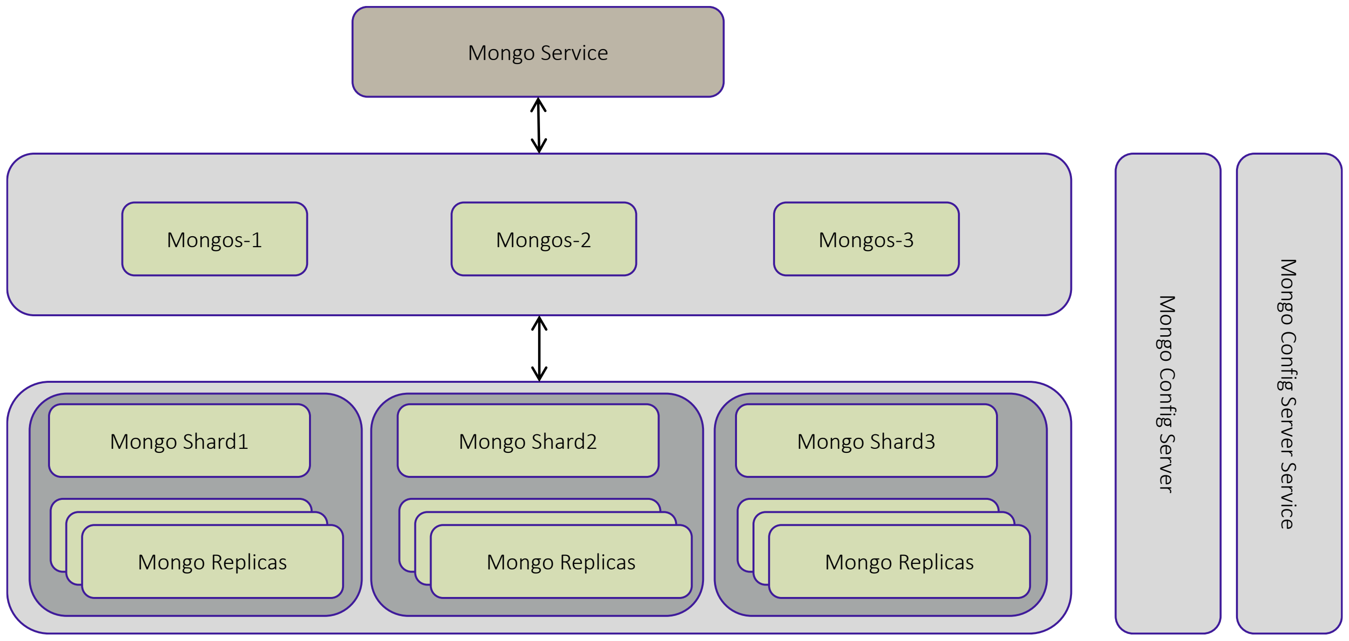
Separate pods for each of the database services – MongoDB, Redis and Memcached – are deployed and used by different services. This type of deployment works on a single node but has limited scalability.

3.7.2 Sharded, Replicated Setup

3.7.2.1 MongoDB Sharding

In this type of deployment, a single MongoDB instance is deployed with support for sharding, replication and persistent storage. This type of setup reflects real-world usage scenarios. Large data sets are divided into smaller chunks that are handled separately by individual shards. Each shard has replication enabled to eliminate a single point of failure. The sharded MongoDB setup can be scaled to run on multi-node clusters. The helm chart for deploying sharded MongoDB has been integrated with Bitnami’s MongoDB sharded package (see *Figure 6*).

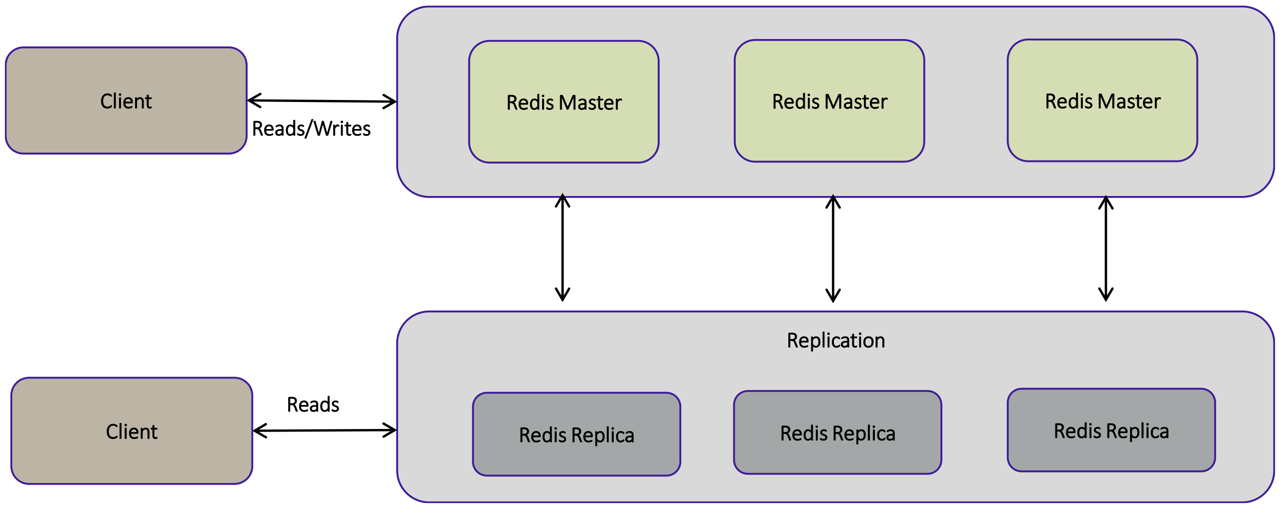
Figure 6: MongoDB Sharded Deployment



3.7.2.2 Redis Cluster

In this mode, master, and multiple replicas of Redis are deployed. It allows scaling read operations by spreading the load across all replicas. Applications can route read operations to a dedicated Redis service (Redis-Replica) and all insert and update operations to Redis-master. It is recommended to serve writes through Redis Master and reads through Redis Replicas. Redis Master replicates writes to one or more Redis Replicas. The master-replica replication is done asynchronously (see *Figure 7*).

Figure 7: Redis Clustered Deployment



3.7.2.3 Memcached Cluster

Memcached replicas are deployed along with mcrouter service to achieve replication and data consistency between Memcached pods. In this scenario, applications can benefit from parallel reads from all Memcached replicas without a risk of hitting cache miss.

Note: During our testing, we used Memcached in non-clustered mode and plan to explore Memcached clustering in the next phase. We used the default setup for Memcached, where separate pods are deployed and used by separate services.

3.8 Running the Load Generator

We used a dedicated server that is not part of the Kubernetes cluster as our client system to run the load generator WRK2. The client system also runs Ubuntu 22.04 as the operating system and connect to the test Kubernetes cluster on an internal network.

Every test run starts by initializing the dataset. This script registers users and constructs the social graph for the application:

```
python3 scripts/init_social_graph.py --graph=socfb-Reed98
```

We used the mixed-workload.lua script to simulate a mix of read/writes on the social network application:

```
cd DeathStarBench/socialNetwork/wrk2
```

```
./wrk -t <num-threads> -c <num-conns> -d <duration> -L -s ./scripts/social-network/mixed-
workload.lua http://localhost:8080 -R <requests-per-sec>
```

Start the test with a thread count = 10, number of connections = 1000 and duration of 5 minutes. The requests per second are set to an initial constant throughput (R value) based on the size of the cluster and increased in steps of 500. Each test is repeated five times for every R value to account for run-to-run variance. There are no resource limits configured for any containers and containers can use as much of the CPU or memory that the scheduler allows.

```
cd wrk2
```

```
./wrk -t 10 -c 1000 -d 5m -L -s ./scripts/social-network/mixed-workload.lua
http://localhost:8080 -R 1000
```

3.9 Monitoring Using Grafana and Prometheus

Prometheus is an open-source systems and service monitoring solution that collects and stores its metrics as time series data. Grafana or other API consumers can be used to visualize the collected data.

Kube-prometheus is a GitHub repository that collects Kubernetes manifests, Grafana dashboards, and Prometheus rules combined with documentation and scripts to provide easy to operate end-to-end Kubernetes cluster monitoring with Prometheus using the Prometheus Operator.

To install kube-prometheus on the Kubernetes cluster, download/clone the repository using the following commands:

```
git clone https://github.com/prometheus-operator/kube-prometheus.git

cd kube-prometheus

kubectl apply --server-side -f manifests/setup

kubectl apply -f manifests/

kubectl get svc -n monitoring
```

The Grafana dashboard is accessible via http://<grafana_service_ip>:3000 using the default Grafana credentials — user: admin, password: admin

Note: The monitoring services are not run during the benchmarking process and are only used for analyzing cpu-utilization, I/O, and network usage.

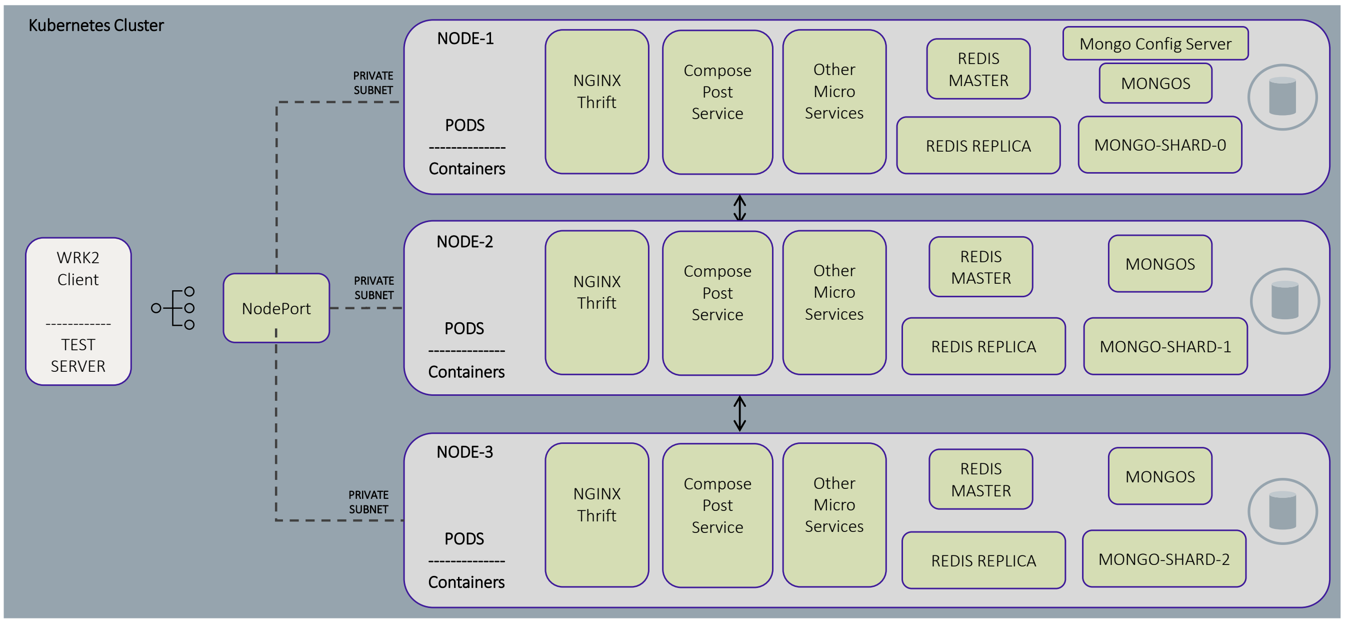
4. Benchmarking

The DeathStarBench/socialNetwork project allows deploying the social network application on a Kubernetes cluster natively using helm charts. The front-end and application layers of this application run as pods on a Kubernetes cluster allowing multiple replicas of the pods to be deployed to handle horizontal scaling of the workload.

4.1 Benchmarking Configuration

Figure 8 shows the DeathStarBench/socialNetwork application deployed on a 3-node Kubernetes cluster:

Figure 8: DeathStarBench/socialNetwork Application Deployed on a 3-Node Kubernetes Cluster



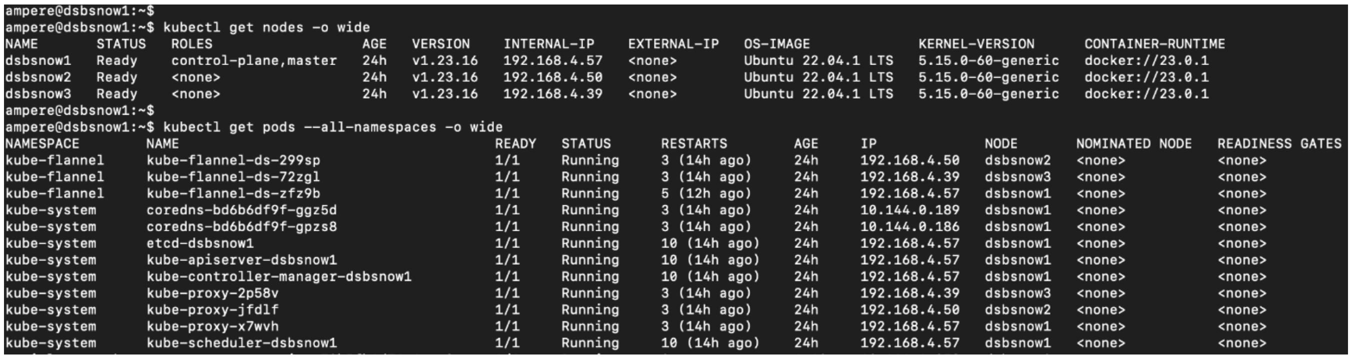
4.2 Benchmarking Steps

Step 1. Create a Kubernetes cluster: Start by creating a Kubernetes cluster on bare metal servers. Detailed steps for creating the cluster using the Kubeadm tool are documented in the *Transition and Tuning Guide* available here.

Step 2. Set up the client system: The workload for the social network application can be run from a separate system using a client/server topology. We used a bare metal server with the same configuration as the server under test (SUT) for our client. The client system and the Kubernetes cluster are configured to use the same internal network.

Start by installing Ubuntu 22.04 on the client system. Install kubectl and copy the KubeConfig file in order for the client system to access the Kubernetes cluster. Verify the node status is set to Ready and all kube-system and flannel pods are running (see *Figure 9*).

Figure 9: Verifying Node Status, kube-systems, and Flannel Pods



```
ampere@dsbsnow1:~$
ampere@dsbsnow1:~$ kubectl get nodes -o wide
NAME       STATUS   ROLES                  AGE   VERSION    INTERNAL-IP    EXTERNAL-IP   OS-IMAGE             KERNEL-VERSION      CONTAINER-RUNTIME
dsbsnow1   Ready    control-plane,master   24h   v1.23.16   192.168.4.57   <none>        Ubuntu 22.04.1 LTS   5.15.0-60-generic   docker://23.0.1
dsbsnow2   Ready    <none>                 24h   v1.23.16   192.168.4.50   <none>        Ubuntu 22.04.1 LTS   5.15.0-60-generic   docker://23.0.1
dsbsnow3   Ready    <none>                 24h   v1.23.16   192.168.4.39   <none>        Ubuntu 22.04.1 LTS   5.15.0-60-generic   docker://23.0.1
ampere@dsbsnow1:~$
ampere@dsbsnow1:~$ kubectl get pods --all-namespaces -o wide
NAMESPACE      NAME                                       READY   STATUS      RESTARTS        AGE   IP              NODE       NOMINATED NODE   READINESS GATES
kube-flannel   kube-flannel-ds-299sp                      1/1     Running     3 (14h ago)     24h   192.168.4.50    dsbsnow2   <none>           <none>
kube-flannel   kube-flannel-ds-72zgl                      1/1     Running     3 (14h ago)     24h   192.168.4.39    dsbsnow3   <none>           <none>
kube-flannel   kube-flannel-ds-zfz9b                      1/1     Running     5 (12h ago)     24h   192.168.4.57    dsbsnow1   <none>           <none>
kube-system    coredns-bd6b6df9f-ggz5d                    1/1     Running     3 (14h ago)     24h   10.144.0.189    dsbsnow1   <none>           <none>
kube-system    coredns-bd6b6df9f-gpzs8                    1/1     Running     3 (14h ago)     24h   10.144.0.186    dsbsnow1   <none>           <none>
kube-system    etcd-dsbsnow1                              1/1     Running     10 (14h ago)    24h   192.168.4.57    dsbsnow1   <none>           <none>
kube-system    kube-apiserver-dsbsnow1                    1/1     Running     10 (14h ago)    24h   192.168.4.57    dsbsnow1   <none>           <none>
kube-system    kube-controller-manager-dsbsnow1           1/1     Running     10 (14h ago)    24h   192.168.4.57    dsbsnow1   <none>           <none>
kube-system    kube-proxy-2p58v                           1/1     Running     3 (14h ago)     24h   192.168.4.39    dsbsnow3   <none>           <none>
kube-system    kube-proxy-jfdlf                           1/1     Running     3 (14h ago)     24h   192.168.4.50    dsbsnow2   <none>           <none>
kube-system    kube-proxy-x7wvh                           1/1     Running     3 (14h ago)     24h   192.168.4.57    dsbsnow1   <none>           <none>
kube-system    kube-scheduler-dsbsnow1                    1/1     Running     10 (14h ago)    24h   192.168.4.57    dsbsnow1   <none>           <none>
```

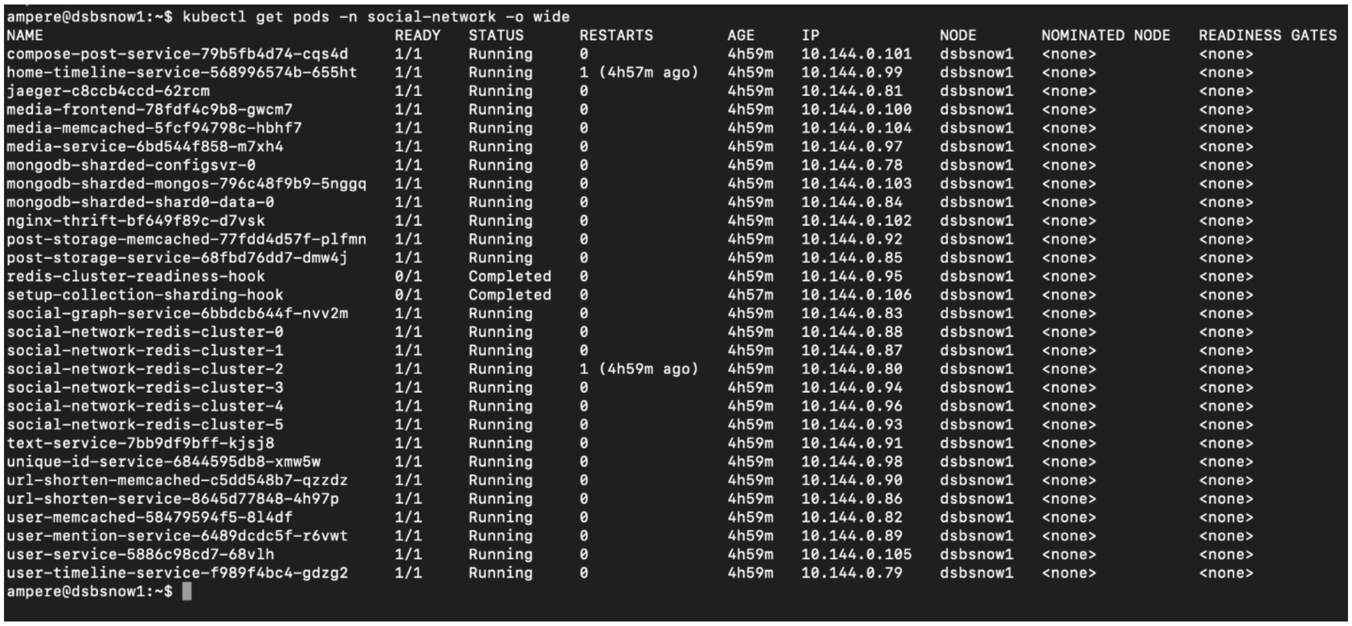
Step 3. Deploy the social network application on the Kubernetes cluster using helm: When running on a multi-node cluster, one replica of the frontend pod (NGINX) and one replica of each application service is deployed on every node by setting the global.replicas parameter in the helm chart. Database scaling for the backend caching layers (Redis, Memcached) and database (MongoDB) is implemented by enabling clustering.

```
cd DeathStarBench/socialNetwork/helm-chart

helm install social-network ./socialnetwork/ -n social-network --set \

global.mongodb.sharding.enabled=true,\

global.mongodb.standalone.enabled=false,\

mongodb-sharded.shards=1,\

mongodb-sharded.mongos.replicaCount=1,\

mongodb-sharded.shardsvr.dataNode.replicaCount=1,\

global.redis.cluster.enabled=true,\

global.redis.standalone.enabled=false,\

global.memcached.cluster.enabled=false,\

global.memcached.standalone.enabled=true,\

mcrouter.memcached.replicaCount=1,\

global.replicas=1 \

--timeout 5m0s
```

The above example shows the command used to deploy the social network application on a single node. Set the values for `global.replicas` and `replicaCount` for the shards, Mongos, and Redis cluster nodes based on the number of nodes in the Kubernetes cluster. More information about the commands for deploying the application on different clusters can be found in the *Appendix*.

Step 4. Once the installation is complete, verify that all the pods and services are in Running state (see *Figure 10*).

Figure 10: Verifying Pods and Services in Running State



```
ampere@dsbsnow1:~$ kubectl get pods -n social-network -o wide
NAME                                      READY   STATUS      RESTARTS         AGE     IP             NODE       NOMINATED NODE   READINESS GATES
compose-post-service-79b5fb4d74-cqs4d     1/1     Running     0                4h59m   10.144.0.101   dsbsnow1   <none>           <none>
home-timeline-service-568996574b-655ht    1/1     Running     1 (4h57m ago)    4h59m   10.144.0.99    dsbsnow1   <none>           <none>
jaeger-c8ccb4ccd-62rcm                    1/1     Running     0                4h59m   10.144.0.81    dsbsnow1   <none>           <none>
media-frontend-78fdf4c9b8-gwcm7           1/1     Running     0                4h59m   10.144.0.100   dsbsnow1   <none>           <none>
media-memcached-5fcf94798c-hbhf7          1/1     Running     0                4h59m   10.144.0.104   dsbsnow1   <none>           <none>
media-service-6bd544f858-m7xh4            1/1     Running     0                4h59m   10.144.0.97    dsbsnow1   <none>           <none>
mongodb-sharded-configsvr-0               1/1     Running     0                4h59m   10.144.0.78    dsbsnow1   <none>           <none>
mongodb-sharded-mongos-796c48f9b9-5nggq   1/1     Running     0                4h59m   10.144.0.103   dsbsnow1   <none>           <none>
mongodb-sharded-shard0-data-0             1/1     Running     0                4h59m   10.144.0.84    dsbsnow1   <none>           <none>
nginx-thrift-bf649f89c-d7vsk              1/1     Running     0                4h59m   10.144.0.102   dsbsnow1   <none>           <none>
post-storage-memcached-77fdd4d57f-plfmn   1/1     Running     0                4h59m   10.144.0.92    dsbsnow1   <none>           <none>
post-storage-service-68fbd76dd7-dmw4j     1/1     Running     0                4h59m   10.144.0.85    dsbsnow1   <none>           <none>
redis-cluster-readiness-hook              0/1     Completed   0                4h59m   10.144.0.95    dsbsnow1   <none>           <none>
setup-collection-sharding-hook            0/1     Completed   0                4h57m   10.144.0.106   dsbsnow1   <none>           <none>
social-graph-service-6bbdcb644f-nvv2m     1/1     Running     0                4h59m   10.144.0.83    dsbsnow1   <none>           <none>
social-network-redis-cluster-0            1/1     Running     0                4h59m   10.144.0.88    dsbsnow1   <none>           <none>
social-network-redis-cluster-1            1/1     Running     0                4h59m   10.144.0.87    dsbsnow1   <none>           <none>
social-network-redis-cluster-2            1/1     Running     1 (4h59m ago)    4h59m   10.144.0.80    dsbsnow1   <none>           <none>
social-network-redis-cluster-3            1/1     Running     0                4h59m   10.144.0.94    dsbsnow1   <none>           <none>
social-network-redis-cluster-4            1/1     Running     0                4h59m   10.144.0.96    dsbsnow1   <none>           <none>
social-network-redis-cluster-5            1/1     Running     0                4h59m   10.144.0.93    dsbsnow1   <none>           <none>
text-service-7bb9df9bff-kjsj8             1/1     Running     0                4h59m   10.144.0.91    dsbsnow1   <none>           <none>
unique-id-service-6844595db8-xmw5w        1/1     Running     0                4h59m   10.144.0.98    dsbsnow1   <none>           <none>
url-shorten-memcached-c5dd548b7-qzzdz     1/1     Running     0                4h59m   10.144.0.90    dsbsnow1   <none>           <none>
url-shorten-service-8645d77848-4h97p      1/1     Running     0                4h59m   10.144.0.86    dsbsnow1   <none>           <none>
user-memcached-58479594f5-814df           1/1     Running     0                4h59m   10.144.0.82    dsbsnow1   <none>           <none>
user-mention-service-6489dcdc5f-r6vwt     1/1     Running     0                4h59m   10.144.0.89    dsbsnow1   <none>           <none>
user-service-5886c98cd7-68vlh             1/1     Running     0                4h59m   10.144.0.105   dsbsnow1   <none>           <none>
user-timeline-service-f989f4bc4-gdzg2     1/1     Running     0                4h59m   10.144.0.79    dsbsnow1   <none>           <none>
ampere@dsbsnow1:~$ █
```

Step 5. This step will initialize the dataset and build the social-graph for the social network application.

```
cd DeathStarBench/socialNetwork

python3 ./scripts/init_social_graph.py --graph=socfb-Reed98 --ip=<NodeIp> --
port=<NodePort>
```

Step 6. Run the mixed workload load generator script with initial constant throughput (R value) of 5000 requests per second (RPS) for a duration of 5 minutes.

```
cd DeathStarBench/socialNetwork/wrk2

./wrk -t 10 -c 1000 -d 5m -L -s ./scripts/social-network/mixed-workload.lua
http://<node-ip>:<node-port> -R 5000
```

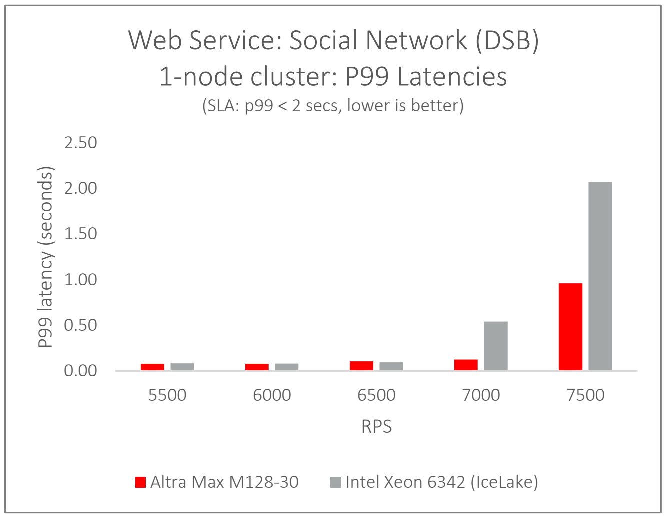
Record the P99 latency and actual throughput at the end of the run. Repeat the test with the same parameters five times. Now gradually increase the R value by 500 and run the mixed workload load generator again. Repeat this process for R values from 5000 RPS to 18000 RPS as you continue to scale number of nodes in the cluster.

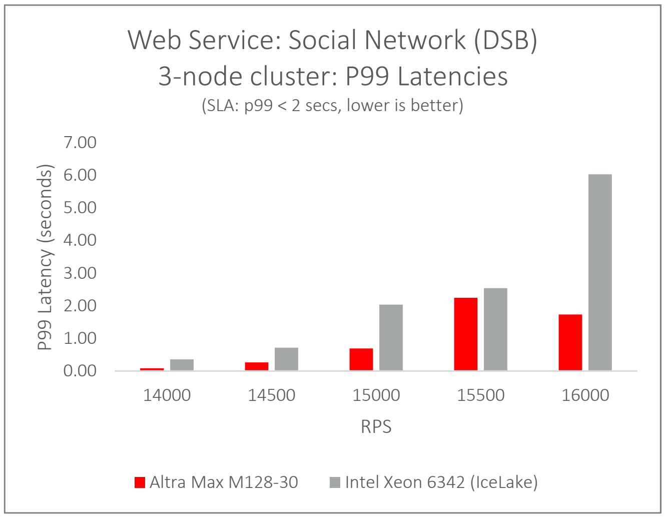
We use an SLA (service level agreement) of P99 latency < 2.00 seconds to measure the peak performance for each test. This is essential for any web service vendor that provides client-facing systems as minimizing latency to provide a high level of service is critical. We calculate P99 latencies by repeating each test five times and use the GEOMEAN of the five measured P99 latencies.

4.3 Benchmarking Results

Figure 11 shows the results of running WRK2 mixed-workload script that simulates users connecting to the web service. We measured the P99 latency, throughput (RPS), power usage and other metrics in a clustered deployment and used the values to estimate the performance as the cluster is scaled out to the rack level.

Figure 11: Web Services for Social Network (DSB) – 1-Node Cluster vs. 3-Node Cluster with P99 Latencies





As mentioned earlier, we measured the performance of the social network application under a service level agreement (SLA), where the SLA is set to P99 latency being less than 2.00 seconds. That means any results where P99 latency for user requests is greater than 2.00 seconds will not be taken into consideration when calculating the peak performance.

The chart above plots the measured P99 latencies with increasing load (RPS). When comparing latencies on the two clusters, the latencies on Ampere Altra Max are lower, and gradually start increasing as load on the web service increases. On the other hand, the latencies in the x86 cluster are higher to start with and continue to increase sharply as the load increases. At peak load measured under SLA, 99% of user requests on Ampere Altra Max received a response 77% faster than from the Ice Lake server. As the cluster is scaled to 3-nodes, the peak performance improved more than 2x on the Ampere Altra Max cluster. At this scale, the response times on the Ampere Altra Max cluster were 64% faster than the x86 cluster while handling the same number of user requests per second.

Figure 12: Web Services for Social Network (DSB) – Relative Performance at Rack Level

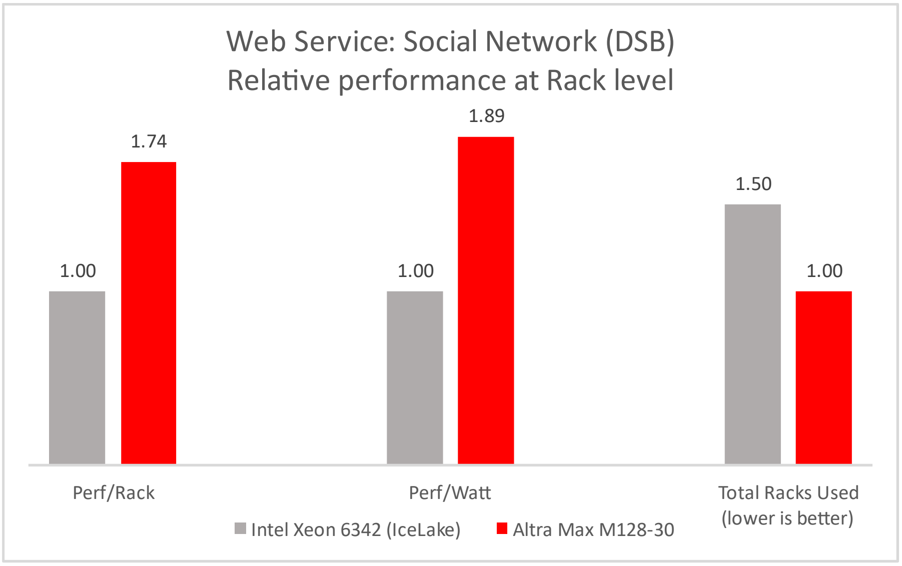
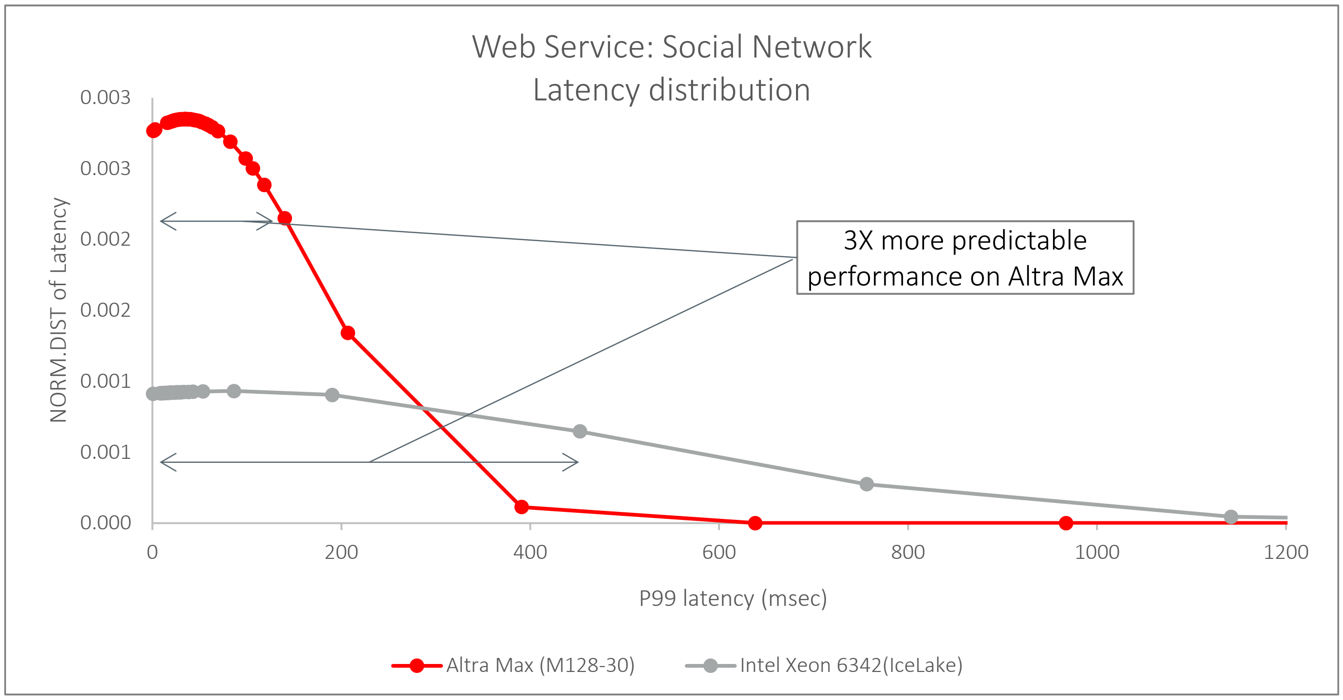


Figure 12 above compares the performance of the social network application in terms of throughput (or RPS) on Ampere Altra Max to the Intel Xeon 6342 cluster. The Ampere cluster was able to handle higher load (total RPS) while still maintaining an SLA of P99 latency under 2.00 seconds. We measured the power usage on both the servers at the system level at peak load and used this data to calculate the performance per watt (Perf/Watt). The lower power consumption on the Ampere Altra Max servers yields a 1.9x higher Perf/Watt compared to the x86 servers. And finally, combining the performance gains and the performance per watt advantage results in 1.7x higher performance on Ampere Altra Max at the rack level. That translates to a single rack of Ampere servers handling 70% more requests per second than a rack of Intel servers. Because of the high power utilization and lower Perf/Watt observed on legacy x86 servers, one will need 1.5x racks of x86 servers in order to handle the same load as a single rack of Ampere servers.

Figure 13: Web Service – Social Network Latency Distribution



Another important benchmarking result shows 3x better predictability for response times on Ampere servers. We compared the distribution of latencies for all user requests during a single test run using the mean and standard deviation measured by WRK2. The bell curve (normalized distribution) chart shown in *Figure 13* illustrates a much narrower range of latencies on the Altra Max cluster when compared to the Intel Ice Lake cluster. This correlates to more users on the Ampere cluster having a much faster and more uniform experience even under stressful load conditions.

4.4 Rack and Datacenter-Level Efficiency

Companies are increasing relying on data centers either on-premises or in the cloud to run their websites, e-commerce applications and to store various data. The amount of energy required to run these data centers continues to grow rapidly and to combat this, it is essential to produce more compute capacity in every rack.

The lower power consumption, superior performance, and energy efficiency of the Ampere Altra Max cores, results in fewer racks of servers being needed to handle the same workload. This delivers much more sustainable consumption by reducing the overall resource footprint.

Figure 14: Power Usage on 3-Node Intel Ice Lake vs. Altra Max M128-30 Cluster

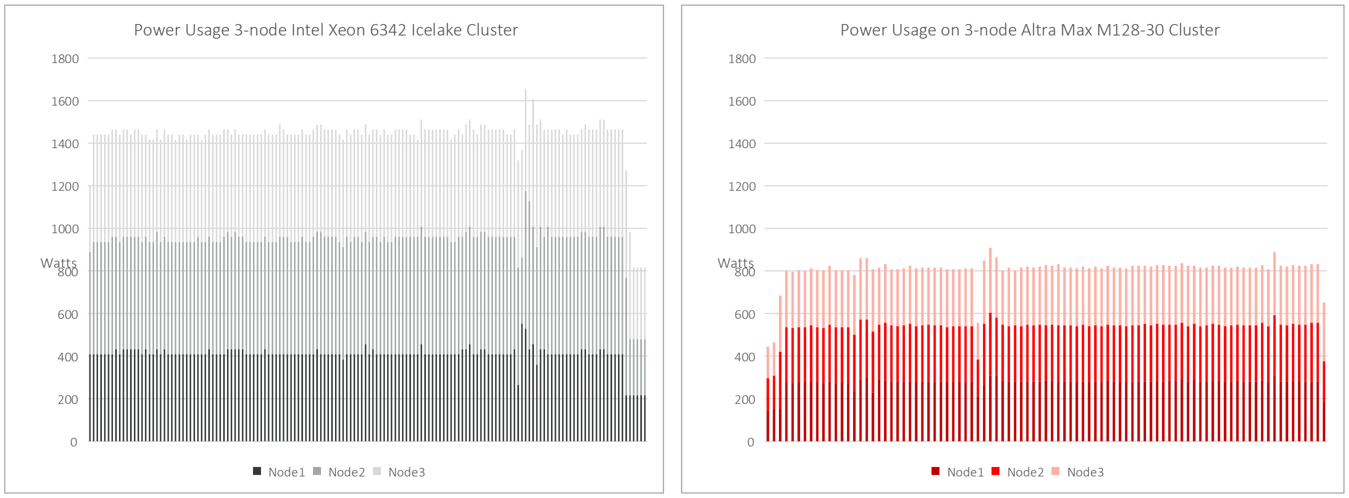
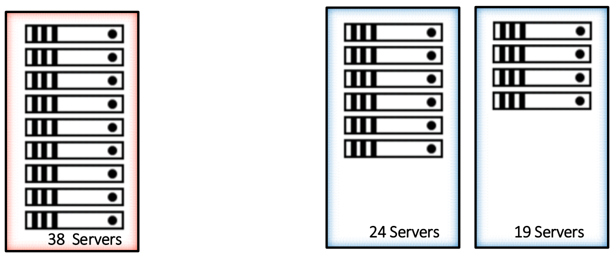


Figure 14 shows the power usage measured at the system level on a 3-node Ampere Altra Max cluster and a 3-node Intel Xeon 6342 cluster while running web services at peak load. It is evident that the power used by the Altra Max servers is almost half of the power used on the Intel servers while handling the same number of user requests, which eventually leads to more Ampere Altra Max servers that can be racked on a single rack without exceeding the rack power budget. One the other hand, the higher power usage on Intel servers leads to less servers per rack and a bigger datacenter footprint (see *Figure 15*).

Figure 15: Ampere M128-30 vs. Intel Ice Lake 6342

DeathStarBench/socialNetwork Service

Web Tier	App
Front End	NGINX
Caching Tier	Memcached
Key Value Store	Redis
Back End	MongoDB
Applications	C, C++, Python, Lua
Perf Target	
P99 latency < 2 seconds	



Ampere M12830
38 1S/1U Servers
1 Rack*
9.6 kW Total

Intel Ice Lake 6342
43 2S/1U Servers
1.5 Racks*
18.5 kW Total

* Based on 12kW constrained 42U Racks leaving space for network & PDU

We used the performance and power usage data measured on a 3-node cluster and extended that to a rack full of servers using the same method for the Ampere Altra Max and Intel Ice Lake clusters. Assuming a rack with a power budget of 12 kW and 42 units of capacity (leaving room for network and other equipment) we calculated the number of servers that can fit in a rack and computed the performance and power usage at the rack level using those values.

As the chart in *Figure 14* illustrates, to match the performance of a single rack of Ampere Altra Max servers running the DeathStarBench/socialNetwork application, you need 1.5x racks of x86 servers. The x86 servers also use up to 1.9x more power when running the social network application under the same load.

The rack and datacenter level efficiency shows that using Ampere Altra Cloud Native Processors in your datacenter is significantly more sustainable and can reduce the overall resource footprint of a modern web service deployment by over 50%. This kind of advantage is truly disruptive, making the Ampere Altra family the most sustainable processor for modern cloud infrastructure.

5. Conclusion

The reference architecture and the solution presented here showcases an example of a real-world web service running on a multi-node cluster with Ampere Altra Max processors. The process of porting and deploying the social network application is straightforward. The stack uses multiple cloud native technologies on the AArch64 architecture and many of the components are readily available as docker images from Docker hub and the rest can easily be rebuilt to support AArch64. The performance comparison on a scaled-out multi-node cluster shows higher and more predictable performance on the Altra Max systems compared to legacy x86 platforms. We also observed exceptional power and rack level efficiency while running a large-scale application like a web service.

6. Additional Resources

- NGINX tuning guide
- MongoDB tuning guide
- Memcached tuning guide
- DeathStarBench/socialNetwork porting and deployment guide (see this link)

7. Appendix

7.1 Configuration Parameters for Deploying the Social Network Application

Command used to deploy social network application on a 1-node cluster:

```
helm install social-network ./socialnetwork/ -n social-network --set \
global.mongodb.sharding.enabled=true,\
global.mongodb.standalone.enabled=false,\
mongodb-sharded.shards=1,\
mongodb-sharded.mongos.replicaCount=1,\
mongodb-sharded.shardsvr.dataNode.replicaCount=1,\
mongodb-sharded.common.mongodbDisableSystemLog=true,\
global.redis.cluster.enabled=true,\
global.redis.standalone.enabled=false,\
global.memcached.cluster.enabled=false,\
global.memcached.standalone.enabled=true,\
global.replicas=1 \
--timeout 5m0s
```

Command used to deploy social network application on a 2-node cluster:

```
helm install social-network ./socialnetwork/ -n social-network --set \
global.mongodb.sharding.enabled=true,\
global.mongodb.standalone.enabled=false,\
mongodb-sharded.shards=2,\
mongodb-sharded.mongos.replicaCount=2,\
mongodb-sharded.image.pullPolicy="IfNotPresent",\
mongodb-sharded.shardsvr.dataNode.replicaCount=1,\
mongodb-sharded.common.mongodbDisableSystemLog=true,\
global.redis.cluster.enabled=true,\
global.redis.standalone.enabled=false,\
global.memcached.cluster.enabled=false,\
global.memcached.standalone.enabled=true,\
global.replicas=2 \
--timeout 5m0s
```

Command used to deploy social network application on a 3-node cluster:

```
helm install social-network ./socialnetwork/ -n social-network --set \
global.mongodb.sharding.enabled=true,\
global.mongodb.standalone.enabled=false,\
mongodb-sharded.shards=3,\
mongodb-sharded.mongos.replicaCount=3,\
mongodb-sharded.image.pullPolicy="IfNotPresent",\
mongodb-sharded.shardsvr.dataNode.replicaCount=1,\
mongodb-sharded.common.mongodbDisableSystemLog=true,\
global.redis.cluster.enabled=true,\
global.redis.standalone.enabled=false,\
global.memcached.cluster.enabled=false,\
global.memcached.standalone.enabled=true,\
global.replicas=3 \
--timeout 5m0s
```

7.2 SLC on Ampere Altra and Altra Max

We enabled the option to report SLC as L3 for the tests that are documented here. Details for this setting are documented here: Altra Family reporting SLC as L3.

7.3 Footnotes

As part of web services performance benchmarking, we observed run-to-run variations in the measured latency and throughput due to the randomness built into the load generator. We have not configured requests/limits on the CPU or memory use by the individual microservices, which also leads to run-to-run variations. In order to minimize the effects of these variations, we ran each test 5 times and used the GEOMEAN of the measured latency and throughput for our final calculations.

8. Revision History

ISSUE	DATE	DESCRIPTION
1.00	May 10, 2023	Initial release.

May 10, 2023

Ampere Computing

4655 Great America Parkway, Santa Clara, CA 95054

Phone: (669) 770-3700

https://www.amperecomputing.com