Comparison of Containerized and Native Applications in the Store

Zac Freeman

Reviewers:
Paul Wiersma
Wesley Colin Wright

June 5, 2021

1 Introduction

This document aims to summarize the costs and benefits of deploying Envoy to the stores using Docker containers as a part of Retail Acceleration. The current utility and resource overhead of containerizing Envoy within the Data Sync prototype will be discussed and analyzed in Resource Consumption, Latency, and Deployment Process. The value of Docker to the store ecosystem will be discussed in Development Experience. Any code, data, or assumptions used in the analyses will be listed in Methodology, Data, and Scripts.

2 Expected Outcomes

The goals of this document are to inform Architecture's decision on:

1. containerizing Envoy for the Retail Acceleration Data Sync prototype and final product
2. containerizing further store applications, by providing an approximation of the resource overhead of containerizing further store applications

3 Resource Consumption

The foundation of the Docker container is the Linux namespace. A Linux namespace encapsulates a process to scope its access to system resources. A Docker container is able to make use of the host's kernel and additionally provide the dependencies needed by the containerized software in a partially isolated environment. This approach provides a lightweight solution to software virtualization.

The resource consumption of a containerized application is broken up into two groups. The first group, labelled "Docker Dependencies", consists of the per-system applications such as the docker daemon, the container daemon, etc., that support any number of containerized applications. The other group, labelled "Containerized Envoy", represents the application running inside a container, which gives an approximation of the per-application cost of containerizing an application in the store.

3.1 Storage Consumption

The storage consumption was recorded while the applications were idle, under the assumption that putting the applications under load would not impact their installed size.

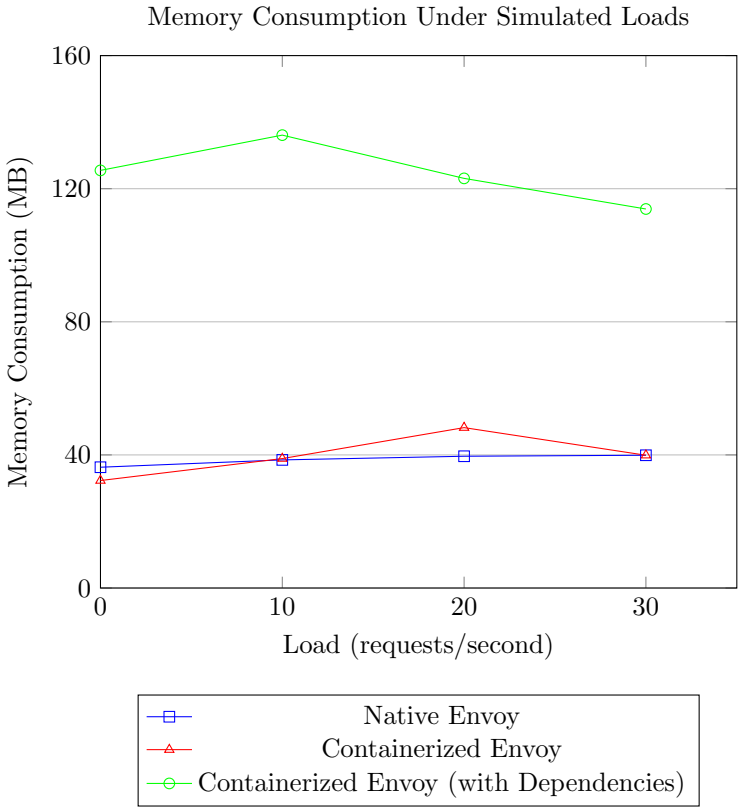
	Storage (MB)
Native Envoy	93.9
Containerized Envoy	84.6
Docker Dependencies	359.4
Increase (with Dependencies)	350.1
Increase (without Dependencies)	-9.3

Table 1: The storage consumed by the native Envoy application and the containerized Envoy application.

The decrease in storage costs between the native application and the containerized application is likely the result of the docker container using a version of the Envoy application being compiled differently, rather than some efficiency provided by Docker.

3.2 Memory Consumption

The following plot was constructed from data collected while the native and containerized Envoy applications were under simulated loads of 0, 10, 20, and 30 requests per second.

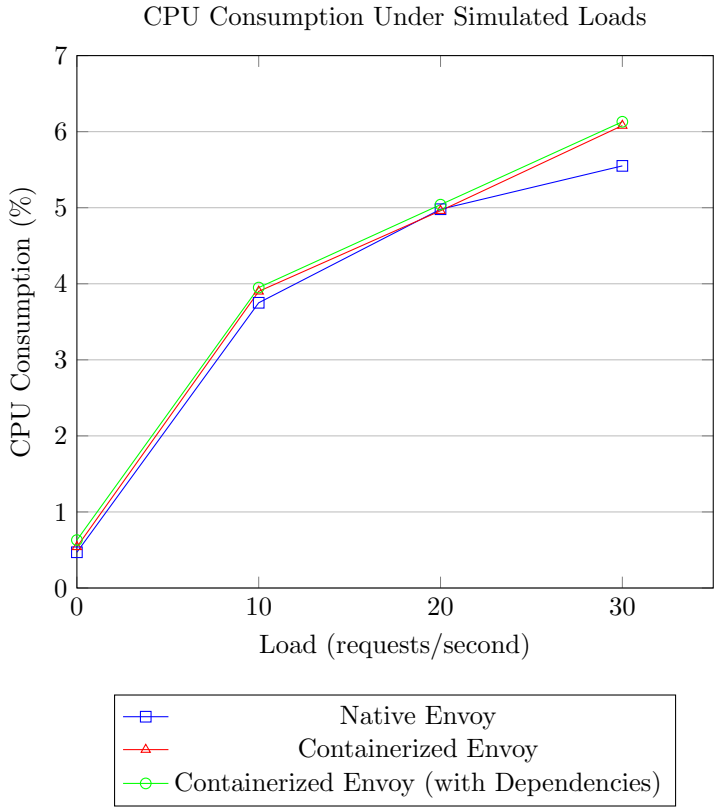


The Containerized Application (with and without Docker Dependencies) represents a fixed cost with respect to load on the containerized Envoy application.

The apparent decrease in Memory Consumption of the Containerized Application with respect to load is a result of variability in the memory consumption measurements.

3.3 CPU Consumption

The following plot was constructed from data collected while the native and containerized Envoy applications were under simulated loads of 0, 10, 20, and 30 requests per second.



The Containerized Application (with and without Docker Dependencies) represent a small increase in CPU Consumption over the Native Application.

The divergence at 30 requests/second is most likely a result of some variability in the method of measurement of CPU Consumption, but could warrant some more rigorous investigation.

3.4 Interpretation

Across the board, the containerized Envoy application requires more resources to run than the native Envoy application. Storage Consumption sees the greatest relative increase, followed by Memory Consumption, then CPU Consumption.

While the Storage Consumption of the Containerized Application (with Dependencies) represents a 4 fold increase over the native application, the increase in Storage Consumption is entirely due to the Docker Dependencies. Containerizing an application does cause increase in storage consumption. The more containers that are running on the store, the smaller the relative increase in Storage Consumption becomes.

A similar observation can be made for the Memory Consumption of the Docker Dependencies. The increase in memory consumption is entirely due to the Docker Dependencies. Containerizing an application does not cause an

increase in memory consumption. The more containers that are running on the store, the smaller the relative increase in Memory Consumption becomes.

The increase in CPU Consumption is regular, but small. The typical increase in CPU Consumption between the Native Application and the Containerized Application (with Docker Dependencies) is about 5 times smaller than the CPU Consumption of the Native Application when idle.

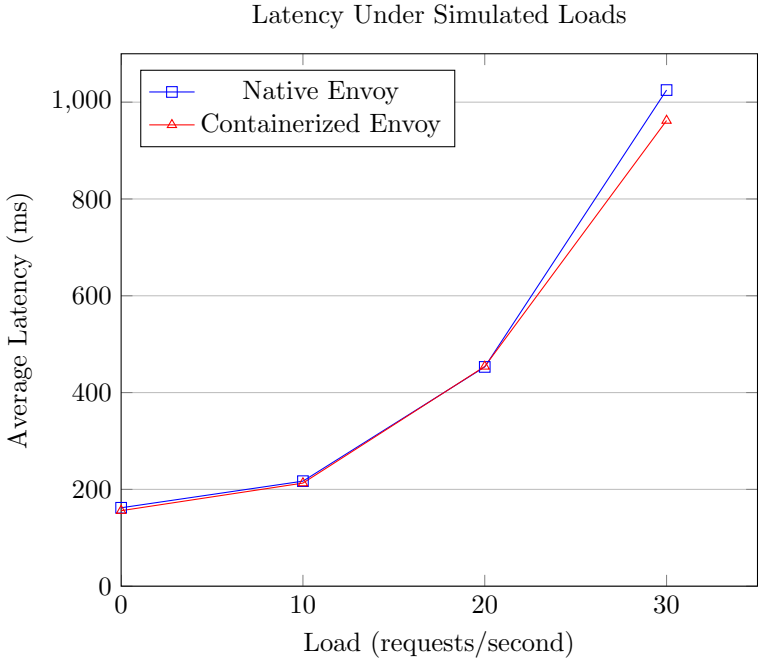
Furthermore, each of these increases in resource consumption do not scale with the load on the application. That is, containerizing an application represents a static resource cost with respect to load.

4 Latency

As mentioned in Resource Consumption, containers leverage the host's processes to minimize the cost of encapsulating an application. Reusing the host's network interface provides the additional benefit of reducing the latency introduced by the container.

4.1 Results

The following plot was constructed from data collected while the native and containerized Envoy applications were under simulated loads of 0, 10, 20, and 30 requests per second, in addition to the requests made in order to survey their average response times.



Under each simulated load, the difference between the average latencies of Native and Containerized Envoy was well within the standard deviation of each average latency. The negative increase in latency from the Containerized Application represents an insignificant difference in the calculated average latencies.

4.2 Interpretation

The containerized Envoy application does not incur a measurable increase in response times over the native Envoy application. This analysis of the relative performance in the store agrees with similar research from IBM's research report, *An Updated Performance Comparison of Virtual Machines and Linux Containers*.

While the trend in the difference in latencies is promising, the trend of the latencies themselves is concerning. The underlying cause of the superlinear increase in average latency should be investigated further to determine if components of the Data Sync prototype need to be reconsidered to meet the needs of the store.

5 Deployment Process

Currently, the Z-neXt project employs Docker to manage a containerized Envoy application. Envoy routes requests from in-store applications to either an above-store or in-store service, depending on the health of the above-store service.

5.1 Container Deployment

For the Data Sync prototype, deploying the Envoy container to a store involves installing the Docker daemon in the store, then downloading the official Envoy Docker image from Docker Hub. Installing the Docker daemon necessitates enabling the `ol7_addons-prod` package repository in the store. Once the image is downloaded to the store, it can be started by the Docker daemon.

This solution could be made production-ready by hosting the Docker image in the AutoZone Artifactory, and retrieving it in the store from the AutoZone Artifactory. Furthermore, the store deployment strategies would need to be extended to cover containers: preemptive container downloads, container lifecycle management, `yum`-esque dependency management, etc.

Lastly, some consideration would need to be given to how to manage detached containers. Currently, `systemd` is able to manage the Envoy container, but this hinges on the Envoy container being attached to a shell through which `systemd` can manage it.

5.2 Native Deployment

Deploying the native Envoy application to a store would require adding the `tetrate-getenvoy` package repository to the store. Once the package reposi-

tory is added, the Envoy application can be installed with the `getenvoy-envoy` package.

For the purposes of the Data Sync prototype, this solution necessitates micromanaging the added package repository. The containerized solution allows us more flexibility and velocity without sacrificing any opportunities for the prototype.

6 Development Experience

Docker provides a relatively approachable API for managing containers. Furthermore, Docker containers adhere to an industry-wide standard for containers, enabling easy transitions to other container management technologies.

6.1 Loose Coupling

Containers loosen the coupling between the store applications and the store system. This enables developing, running, and testing a store application on a local machine, in a production-like environment. This reduces the time between code change and application feedback, reduces disparities between development, testing, and production environments, and increases developer productivity.

Furthermore, containerizing an application smooths its path towards running in the "cloud", or any other hardware that AutoZone may consider for store systems.

6.2 Portability

Containers simplify the process of running store applications outside of a store by packaging applications with everything needed to run the application. Since an application's dependencies must be declared in the definition of its container, it becomes easier and more natural for a developer to state explicitly the dependencies of the applications they create. Encouraging the use of Docker for store applications would make each application more independent of a store's operating system and a store's configuration.

6.3 Onboarding Acceleration

Docker and containers provide a common language for environment configuration that can replace some of the domain-specific knowledge required to develop on the stores. This common language accelerates the onboarding process of new store developers by standardizing parts of the process and leveraging the developer's existing knowledge of Docker and containers.

6.4 Reusability

Containers, and their definitions, can be broken down into logical pieces called layers. Entire containers and individual layers can be shared between developers.

This enables solutions to environment problems to be made once, stored in version control, and reused indefinitely by future applications.

Furthermore, AutoZone developers can leverage purpose-built containers from 3rd parties to reduce the configuration overhead involved in preparing a system for a new application.

6.5 Reproducibility

Containers enable developers to manage the environment the same way they manage their applications. Containers enable more of the store system's state can be encoded into version control, which makes more of said state more consistent, reproducible, and auditable. Containers ensure that everyone and every environment are running the same code the same way.

7 Methodology

All scripts and commands were executed on `s8959.autozone.com`.

7.1 Storage Consumption

The size of each application was captured with `rpm -qi package_name`. The size of the docker image used by the containerized application was captured from `docker images`.

7.2 Memory Consumption

The memory consumed by each process was equated with the physical memory being used by each process. The physical memory being used by each process was captured with `cat /proc/process_id/status | grep VmRSS`.

At one point, there was an entry in the data for the "Docker Container". This was actually a shell being attached to the container. Starting the container without said shell necessitates running the `docker run` command with `-d` appended to it. Running the container in detached mode makes `systemd` unable to monitor it; process management for docker containers can be partially replaced with the `--restart` policy.

7.3 CPU Consumption

CPU consumption for each process was determined by taking the average of 100 values collected from `top` over a period of 200 seconds. `cpu-usage.sh` is used to collect these values and `stat-analysis.pl` is used to average the collected values.

While a standard deviation is produced for these collected values, it is not recorded as the average of the CPU usage is more akin to a single data point, than an average of data points. This is due to the extremely variable nature of

the CPU usage of a process necessitating a series of measurements over time to approximate the expected CPU usage at one point in time.

7.4 Latency

Latency was measured by taking the average latency of 1000 requests to `https://localhost:10443/v1/items/en-us` in 0.5 second intervals using `latency-test.sh` to collect the latency values, and `stat-analysis.pl` to calculate the average and standard deviation of the collected values. Envoy was configured to route these requests to a local service to reduce the effects of variable network quality on the latency measurements.

7.5 Load Testing

To simulate various network loads on the envoy applications, `load-test.sh` was run to make 10, 20, or 30 requests per second to `https://localhost:10443/v1/items/en-us`. Envoy was configured to route these requests to a local service to avoid stressing the above store application.

8 Data

All data was collected from `s8959.autozone.com`.

8.1 Storage Consumption

The following data was collection while the Envoy application was idle.

The storage consumption listed for the Docker Dependencies represents the storage consumed by the packages added to use Docker in the store. This includes `docker-engine`, `docker-cli`, `containerd`, and `container-selinux`.

	Storage (MB)
`docker-engine`	103.7
`docker-cli`	168.7
`containerd`	87.0
`container-selinux`	<0.1
Total	359.4

Table 2: A breakdown of the storage space consumed by each package that contributes to total storage consumption listed for the Docker Dependencies.

The storage consumption listed for the Containerized Application is just the storage occupied by the Docker image used by the Envoy container. The storage consumption listed for the Native Application is just the storage occupied by the installed Envoy package.

8.2 Memory and CPU Consumption

The following data was collected while the Envoy application was under a simulated load of 0, 10, 20, and 30 requests per second.

The Memory and CPU consumption of `dockerd`, `containerd`, and `containerd-shim` are grouped together into the "Dependencies" and the applications required to support Docker containers in the store.

	Memory (MB)	CPU (%)
Native Envoy	36.3	0.47
Containerized Envoy	32.3	0.54
`dockerd`	75.1	0.05
`containerd`	10.8	0.03
`containerd-shim`	7.3	0.01
Increase (with Dependencies)	89.2	0.16
Increase (without Dependencies)	-4.0	0.07

Table 3: A breakdown of the memory and CPU time consumed by the applications while idle.

	Memory (MB)	CPU (%)
Native Envoy	38.5	3.75
Containerized Envoy	38.9	3.90
`dockerd`	75.1	0.02
`containerd`	14.8	0.03
`containerd-shim`	7.3	<0.01
Increase (with Dependencies)	97.6	0.20
Increase (without Dependencies)	0.4	0.15

Table 4: A breakdown of the memory and CPU time consumed by the applications while under a load of 10 requests per second.

	Memory (MB)	CPU (%)
Native Envoy	39.6	4.98
Containerized Envoy	48.2	4.96
`dockerd`	54.3	0.06
`containerd`	13.1	0.02
`containerd-shim`	7.5	<0.01
Increase (with Dependencies)	83.5	0.06
Increase (without Dependencies)	8.6	-0.02

Table 5: A breakdown of the memory and CPU time consumed by the applications while under a load of 20 requests per second.

	Memory (MB)	CPU (%)
Native Envoy	39.9	5.55
Containerized Envoy	39.9	6.08
`dockerd`	54.5	0.03
`containerd`	13.0	0.02
`containerd-shim`	7.1	<0.01
Increase (with Dependencies)	74.6	0.58
Increase (without Dependencies)	0	0.53

Table 6: A breakdown of the memory and CPU time consumed by the applications while under a load of 30 requests per second.

The amount of memory consumed by the Envoy applications appear to grow with time when under load, and the memory consumed does not decrease when the simulated load is removed. This could be indicative of some kind of memory leak.

8.3 Latency

The following data was collected while the Envoy application was under a simulated load of 0, 10, 20, and 30 requests per second.

	Average Latency (ms)	Standard Deviation (ms)
Native Application	162	31
Containerized Application	156	39
Increase	-6	

Table 7: A comparison of the applications' response times while idle.

	Average Latency (ms)	Standard Deviation (ms)
Native Application	217	75
Containerized Application	213	72
Increase	-4	

Table 8: A comparison of the applications' response times while under a load of 10 requests per second.

	Average Latency (ms)	Standard Deviation (ms)
Native Application	453	264
Containerized Application	454	278
Increase	1	

Table 9: A comparison of the applications' response times while under a load of 20 requests per second.

	Average Latency (ms)	Standard Deviation (ms)
Native Application	1025	640
Containerized Application	962	564
Increase	-63	

Table 10: A comparison of the applications' response times while under a load of 30 requests per second.

It is interesting to note that standard deviations increase much more than the corresponding average latency when the applications are put under load, suggesting that the simulated load is causing a handful of requests to take much longer rather than causing all the requests to take slightly longer.

9 Scripts

The git repo containing these scripts (and the LaTeX for this document) can be found at https://gitlab.autozone.com/10904928/docker-analysis.

9.1 stat-analysis.pl

```
#!/usr/bin/perl

use strict;

my @values = <>;

my $total = 0;
my $count = 0;
while (my $value = <@values>) {
    $total += $value;
    $count++;
}

my $average = $total / $count;
my $variance = 0;
while (my $value = <@values>) {
    $variance += ($value - $average)**2 / $count;
}

print "Average = ", $average, "\n";
print "Variance = ", $variance, "\n";
print "Standard Deviation = ", sqrt($variance), "\n";
```

9.2 cpu-usage.sh

```
#!/bin/bash

if [[ $# -lt 2 ]]
then
    echo "missing arguments"
    exit
fi

pid="$1"
output="$2"
interval=1
num=100
if [[ $# -ge 3 ]]
then
    interval="$3"
fi

if [[ $# -eq 4 ]]
then
    num="$4"
fi

echo "recording CPU usage of process $pid $num times every 2*$interval seconds into $output.

rm "$output"

for (( count=0; count<num; count++ ))
do
    top -b -n 2 -d "$interval" -p "$pid" | tail -1 | awk '{print $9}' >> "$output"
done
```

9.3 load-test.sh

```
#!/bin/bash

if [[ $# -lt 1 ]]
then
    echo "missing url argument"
    exit
fi

url="$1"
interval=0.1
if [[ $# -eq 2 ]]
```

```
then
  interval="$2"
fi

echo "making requests to $url every $interval seconds..."

while true
do
    curl --insecure --silent --output "/dev/null" "$url" &
    sleep "$interval"
done
```

9.4 latency-test.sh

```
#!/bin/bash

if [[ $# -lt 2 ]]
then
    echo "missing arguments"
    exit
fi

url="$1"
output="$2"
interval=0.5
num=100
if [[ $# -ge 3 ]]
then
    interval="$3"
fi

if [[ $# -eq 4 ]]
then
    num="$4"
fi

echo "recording response times for $num requests to $url made every $interval into $output.

rm "$output"

for (( count=0; count<num; count++ ))
do
    curl --insecure --silent --output "/dev/null" --write-out "%{time_total}\n" "$url" >> "$
    sleep "$interval"
done
```