# Docker Container Principles and Patterns

Version: 1

Last update: 25.10.2014

# Change control

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Changes |
| 1 | 25.10.2014 | Oscar Renalias | Initial version |
|  |  |  |  |

# Table of contents

[Change control 2](#_Toc405131746)

[Table of contents 3](#_Toc405131747)

[Introduction 6](#_Toc405131748)

[Container principles 6](#_Toc405131749)

[Immutable Containers 6](#_Toc405131750)

[Designing container-friendly applications 6](#_Toc405131751)

[Container design patterns 6](#_Toc405131752)

[One Service per Container 6](#_Toc405131753)

[Service execution 6](#_Toc405131754)

[Container process monitoring 7](#_Toc405131755)

[Logging 7](#_Toc405131756)

[Container access 7](#_Toc405131757)

[Recommendations 7](#_Toc405131758)

[Thick Containers 7](#_Toc405131759)

[Service execution 8](#_Toc405131760)

[Container process monitoring 8](#_Toc405131761)

[Logging 8](#_Toc405131762)

[Container access 8](#_Toc405131763)

[Recommendations 8](#_Toc405131764)

[Image patterns 9](#_Toc405131765)

[The shared base container 9](#_Toc405131766)

[Recommendations 9](#_Toc405131767)

[Build delegation 9](#_Toc405131768)

[Recommendations 10](#_Toc405131769)

[Scratch image 10](#_Toc405131770)

[Recommendation 11](#_Toc405131771)

[Microcontainers 11](#_Toc405131772)

[Networking patterns 11](#_Toc405131773)

[Proxy containers 11](#_Toc405131774)

[Networked containers 11](#_Toc405131775)

[Addressable containers 11](#_Toc405131776)

[Persistence patterns 11](#_Toc405131777)

[Data volumes 11](#_Toc405131778)

[Host-based persistence 11](#_Toc405131779)

[Logging patterns 11](#_Toc405131780)

[Docker Logs 11](#_Toc405131781)

[Recommendation 12](#_Toc405131782)

[Host logging 12](#_Toc405131783)

[Recommendation 12](#_Toc405131784)

[Data container logging 12](#_Toc405131785)

[Recommendation 13](#_Toc405131786)

[Logging daemon 13](#_Toc405131787)

[Configuration patterns 13](#_Toc405131788)

[Container environment variables 13](#_Toc405131789)

[Distributed configuration 13](#_Toc405131790)

[Operating system patterns 13](#_Toc405131791)

[Container-oriented host 13](#_Toc405131792)

[Container operating system 13](#_Toc405131793)

[DevOps patterns 14](#_Toc405131794)

[Container as build output 14](#_Toc405131795)

[Using Jenkins 14](#_Toc405131796)

[Recommendation 14](#_Toc405131797)

[In-container builds 14](#_Toc405131798)

[Using Jenkins 15](#_Toc405131799)

[Recommendation 16](#_Toc405131800)

[Container-based development 16](#_Toc405131801)

[Environments as containers 16](#_Toc405131802)

[Toolbox images 17](#_Toc405131803)

[Infrastructure patterns 17](#_Toc405131804)

[Container Restart 17](#_Toc405131805)

[Infrastructure container 17](#_Toc405131806)

[Sidekick container 17](#_Toc405131807)

[Container Metrics 17](#_Toc405131808)

[Out of the box metrics support 17](#_Toc405131809)

[Container metrics generation 18](#_Toc405131810)

[Metrics aggregation 18](#_Toc405131811)

[Heapster 20](#_Toc405131812)

[Dashboards 20](#_Toc405131813)

[Service Discovery 20](#_Toc405131814)

[Recommendation 20](#_Toc405131815)

[Service discovery 21](#_Toc405131816)

[Service discovery with container linking 21](#_Toc405131817)

[Recommendation 22](#_Toc405131818)

[Service discovery with service registry 22](#_Toc405131819)

[Service registry 22](#_Toc405131820)

[Service discovery with etcd 23](#_Toc405131821)

[Recommendation 26](#_Toc405131822)

[DNS-based service discovery 26](#_Toc405131823)

[Recommendation 26](#_Toc405131824)

[Orchestration patterns 26](#_Toc405131825)

[Anti-patterns 27](#_Toc405131826)

[Container Remote Access 27](#_Toc405131827)

[References 27](#_Toc405131828)

# Introduction

In progress.

These patterns are strictly focused on Docker environments, and are complimentary to the more general application and infrastructure patterns described as part of ADA v4: <https://kxsites.accenture.com/groups/ADA/Pages/TechnologyV4/PrinciplesAndPatterns/Patterns/default.aspx?lftTitle=Architecture+Patterns>.

# Container principles

## Immutable Containers

Immutable containers embrace the principle of immutable infrastructure: containers are created once and their configuration and running software is never updated; should any changes be required the container is stopped and replaced with a new functionally equivalent one including the new configuration.

This principle does not preclude running an SSH daemon or using docker exec in a running container, but strongly encourages to use these container access patterns strictly to operate a container’s components (restart, troubleshoot, etc) and never to change the state of a running container beyond the changes the container introduces by itself.

## Designing container-friendly applications

TODO: Something from the 12-factor site, mainly things like “environment is the config”

# Container design patterns

## One Service per Container

The “one service per container” app enforces an approach where a container runs only a single application or service. No process manager is used within the container.

Main advantages of this pattern are

* Smaller, lightweight containers
* Stacks are easier to compose by combining and linking containers with different components
* Smaller attack surface in case of vulnerabilities

As only one service (process) is run in the container, this pattern requires careful consideration of its operational concerns such as logging, monitoring and restarting the container should the process inside crash.

This pattern is commonly combined with Immutable Containers.

### Service execution

The application or service runs as the entry point in the container, specified as the CMD or ENTRYPOINT directives in a Dockerfile.

Even though the application binary or script being set as the container’s entry point may spawn other processes or threads, it will always remain as the only executable component that is run within the container.

This pattern requires that the containerized application or service supports a mode that remains running in the foreground instead of forking itself and switch to a background running mode. For applications like Tomcat, this means starting the process with “catalina.sh start” instead of “catalina.sh run”; applications may provide different approaches to support this scenario, and alternatively custom scripts can be written.

### Container process monitoring

As no process monitor is used within the container, a crash or unexpected failure of the container’s entry point will automatically cause the container to be removed.

For critical services, an external component is necessary to monitor the container status, start a new instance of the image and ensure that all dependent (linked or networked) containers are reconfigured and/or restarted accordingly.

Depending on the complexity of the infrastructure, a more complex orchestrated approach may be necessary to support this scenario. Please refer to Orchestration patterns for more information.

For simpler scenarios, Docker 1.2 supports a command-line toggle to automatically restart a container if it crashes using –restart. See Container restart policies for more information.

### Logging

When implementing this pattern, and as only one application runs per container, there are three different approaches available for handling container logging:

* If the containerized application writes its logs to standard output, gather the data from “docker logs” and fed it into a log daemon or aggregator service. See Docker logs.
* If the containerized application writes its data to text files, two further alternatives are available: mount the application’s log folder from the host, and have the application write its log files to the host file system transparently, see Host logging. It is also possible to mount a data volume as the application’s log folder, see Data container logging.
* If the application uses a logging framework that has support for remote logging, it may be possible to redirect logs to a component such as rsyslog or Logstash for log aggregation. See Logging daemon.

### Container access

When the container only runs a single application it is no longer possible to set up an SSH daemon in the container and use that for local and remote access. Tools like nsinit and nsenter solve the problem by allowing local access to the container.

For Docker 1.3 or newer, docker exec is available out of the box and allows one to start a process within a container.

docker exec –it <container> /bin/bash

When available, docker exec should always be the preferred approach for interacting with running containers.

However, docker exec should only be used as a last resort and not be used to mutate containers at run time.

See Inaccessible Containers for more details.

### Recommendations

<TODO>

## Thick Containers

Thick containers stand opposite to “one service per containers”, as they may include more than one service within a single container, e.g. a full application stack in a single container.

Typically, as a result, think containers are also bigger in size and need a process supervisor to execute all the different processes that are part of the container.

### Service execution

The container’s process supervisor is defined as the container’s ENTRYPOINT.

### Container process monitoring

Since thick containers run more than one service, also require a process supervisor to keep the different processes within the container running, and restart them if they fail.

In addition to standard Linux service managers such as init.d, system or upstart, there are lighter-weight alternatives that should be considered for containers with simpler process supervision needs:

* supervisord – lightweight process monitor; it is written in Python and needs the Python runtime and additional libraries to run.
* runit – lightweight process monitor which provides more features than supervisord such as service dependencies. It is written in C and statically linked so its footprint is minimal (both memory and disk footprint)

### Logging

Just like with One Service per Container, there are several options available for logging:

* docker log is not a suitable option since multiple services logging to standard output would make the messages too difficult to read
* Use Host logging and have a mount several host folders for logging in the container; this requires a clear folder structure for scenarios with multiple containers with multiple services each:

docker run -v /var/log/container-<ID>/nginx:/var/log/nginx –v /var/log/container-<ID>/tomcat:/opt/tomcat/logs -d <container-name>

* Mount a data volume as the application’s log folder, see Data container logging. A single data volume could be used for all the container’s logs, but multiple data volumes with logs is also possible
* Use a Logging daemon and redirect logs to a component such as rsyslog or Logstash for log aggregation. This requires an additional service running in the container.

### Container access

Thick containers usually include an SSH server to access the container and operate/troubleshoot the services running inside.

As of Docker 1.3, docker exec should be the preferred way to execute commands within a running container, e.g. to check a service status or trigger a service restart.

Please note that docker exec is run from the container host, and if therefore needs access to the host. In environments where this is not an option (such as multi-tenant environments), bundling SSH in the container is still the preferred option.

### Recommendations

Containers that have an entire application stack are fully self-contained and do not require orchestration to make them work (either manual orchestration, or external automatic orchestration); this is useful for development and testing purposes as only a single container is ever needed.

For the majority scenarios, thinner One Service per Container should be always preferred.

# Image patterns

## The shared base container

The shared base container pattern creates a general container that is used as the foundation for more specific containers, and it is a great way to ensure that common configuration and packages are present across all containers.

Base container

FROM base

…

FROM base

…

FROM xxx

…

Base containers can be created either from scratch for specific scenarios where using containers with a very small footprint is needed, or using one of the official images available from the Docker Hub such as Ubuntu or CentOS.

When creating new containers, the Dockerfile FROM directive is used to inform Docker to use the base container.

See Container operating system for more information on images specifically tailored to be used as base containers.

### Recommendations

The base container pattern encourages the grouping of common image functionality and reusability, and should be commonly used when working with container hierarchies.

## Build delegation

While Dockerfiles provide capabilities to run any command during the creation of a container, it does not provide the flexibility that a dedicated “infrastructure as code” tool such as Chef, Puppet or Ansible may provide.

In this pattern the Dockerfile’s only role is to bootstrap the preferred automation tool and eventually delegate execution to it to complete the container build process. As long as the automation tool returns a non-zero code to report build failures, docker build will detect the failure and report the overall container build as failed.

There are several scenarios where delegating the build process to an automation tool is desirable:

* When an existing significant investment in infrastructure automation exists, or an extensive collection of automation resources (modules/manifests, recipes or playbooks) is available
* When the container build process is sufficiently complex, and the declarative approach of these tools is preferred to the imperative approach of specifying a sequence of commands in a Dockerfile
* When there is a need to share a large amount of common configuration across containers, without the need to create a complex hierarchical structure of base containers, as Dockerfiles do not currently have a feature to share or include common content
* When different container operating systems are used, or different versions of an operating system, as these tools will abstract all the differences away

For example, when using Puppet the container build delegation process works as follows:

FROM base

RUN rpm -ivh http://yum.puppetlabs.com/puppetlabs-release-el-6.noarch.rpm

RUN yum install -y puppet

ADD puppet /opt/puppet

RUN puppet apply --modulepath \

/opt/puppet/modules \

--verbose \

--detailed-exitcodes \

/opt/ puppet/manifests/docker-entrypoint.pp || [ $? -eq 2 ]

EXPOSE ...

CMD ...

Chef and equivalent tools have similar approaches.

### Recommendations

The main drawback of this pattern is that bootstrapping a configuration management tool inside the container for configuring it is a one-time task that requires the installation of additional components, libraries and their dependencies, and consequently increases the overall size of the container.

Even when removing the configuration management tool’s packages after the container build process is complete will not reduce the container size unless the container itself is “squashed” by merging multiple layers into one and throwing away unused content; Docker currently does not implement support for this (<https://github.com/docker/docker/issues/332>), and there are 3rd party solutions currently available but they involve creating a new container out of an existing multi-layered one: <https://github.com/docker/docker/issues/332>.

Usage of this pattern should therefore be carefully considered. Configuration via Dockerfiles should always be the preferred approach unless the container build process is a very complex process.

## Scratch image

Images that are based on 'scratch' are absolutely void of any content.

The scratch image has the smallest footprint of all Docker images as it does not bundle an entire operating system, and are intended for applications that are statically linked and do not need any support libraries or operating-system tools to work.

A scratch image can be easily creates as follows:

tar cv --files-from /dev/null | docker import - scratch

The scratch image may be rather useful in many scenarios, and it has been published to the Docker registry as such so that it can be used to create new images based off it:

FROM scratch

COPY my-binary /my-binary

CMD ["/my-binary"]

## Recommendation

Complying with the principle of always striving for the smallest possible image, the scratch image plays a key role as it is the smallest possible Docker image ever, and should always be used as the starting point for Microcontainers images.

## Microcontainers

TODO: Ilkka?

# Networking patterns

## Proxy containers

In progress.

## Networked containers

In progress.

## Addressable containers

In progress.

# Persistence patterns

## Data volumes

In progress.

## Host-based persistence

In progress.

# Logging patterns

## Docker Logs

Command docker logs will dump the container’s standard output and error streams to the console. Additionally, Docker also writes container logs in JSON format to **/var/lib/docker/containers/<container ID>/<container-ID>-json.log**.

When using One Service per Container, docker logs is one of the viable alternatives to visualize the containerized application’s logs.

Using Docker logs to capture application logs has one major drawback, however, as each log line is logged as its own line in the Docker logs. As an example, Java stack traces are logged as multiple lines in Docker logs instead of a single very long line and this makes Docker logs to be rather difficult to consume directly, and will require pre-processing.

Not all applications are capable of generating logs to standard output and standard error; it is possible to wrap some of these applications with launchers that redirect log output to standard output channels but that may not always be desirable.

### Recommendation

Reading and processing logs from docker logs is not a recommended approach as per the limitations described above, and this pattern should not be considered when implementing containerized services in Docker.

During container development it may however be a valid alternative since the containerized applications’ logs will be easily visible via the console.

## Host logging

Host logging uses a folder mounted from the host in the container as the location where application logs will be written.

For an application like nginx that writes its logs to /var/log/nginx, this command would write all the logs to a local host folder /var/containers/nginx/logs:

docker run --name nginx -d –v /var/containers/nginx/logs:/var/log/nginx -p 8080:80 nginx

If multiple containers are running in the same host (nginx containers, or another application), a simple naming convention for the local host log folders will ensure that each container writes its own logs to separate locations.

Should log aggregation capabilities be needed, a single log aggregation process running in the host (or itself as a container) and responsible for gathering and propagating the logs to a central storage.



### Recommendation

This pattern is one of the most straightforward approaches to handling container logs, and it is also the preferred approach and should therefore be most commonly implemented.

## Data container logging

Data container logging uses a separate data volume where all logs are written to, by mounting the data volume into the container that generates the logs:



Each one of the containers writes its log data to its own data volume.

If log aggregation is required, a separate log aggregation service (possibly in its own container) is needed to pull the data out of the log data volumes and deliver it to its next step in the aggregation flow. The log aggregator container must be aware of how many log data volumes are present so that it can process log data from all of them; this may be represent a challenge in scenarios where containers are started and stopped dynamically within the Docker host, as it requires orchestration logic to make the log aggregator aware of new and removed containers.

### Recommendation

<TODO>

## Logging daemon

In progress.

# Configuration patterns

## Container environment variables

In progress.

## Distributed configuration

In progress.

# Operating system patterns

## Container-oriented host

In progress.

## Container operating system

In progress.

# DevOps patterns

## Container as build output

Integrates the container build process within a Continuous Integration tool and ensures repeatability of the container build process.

The Continuous Integration platform should also responsible for versioning containers and publishing them to a private/public Docker registry for consumption.



### Using Jenkins

The Jenkins [Docker Build Step plugin](https://wiki.jenkins-ci.org/display/JENKINS/Docker+build+step+plugin) works out of the box and provides support for executing a variety of operations with Docker containers as part of a job, including building and tagging a container out of Dockerfile. The plugin does not currently support pushing containers to a private registry, and therefore an “Execute command” steps is required to carry out the docker push operation.

### Recommendation

There should be no reason to not build containers using a Continuous Integration platform and leverage the automation and versioning capabilities provided.

## In-container builds

Continuous Integration builds run within their own containers.

This approach abstracts the build process away from the CI tool, and pushes the complexity of managing the build runtime tooling and configuration to the containers allowing to:

* run potentially conflicting versions of build time toolset and libraries, e.g. different containers with different versions of the JDK or Ruby runtime in use, as opposed to multiple installation of JDK or Ruby’s rvm within the same CI server
* keep the CI configuration as simple as possible
* greatly simplify the process for introducing new build time toolsets

When using this pattern, the role of the Continuous Integration platform here is to create container, ensure that the code to be built has been pushed into the container and then trigger the build process within the container.

The output of the build process may be another container, which can be pushed to an internal registry for storage.

Products like drone.io and TravisCI (with OpenVZ) implement this approach out of the box.

### Using Jenkins

This pattern can be implemented in two ways for Jenkins:

1. Using Docker-based build slaves; build slaves are dynamically provisioned for each build and destroyed once the build is completed
2. Use a build container that contains all tools to execute the build, and destroy after the build is completed

#### Dynamic build slaves

The first approach use the Jenkins Docker plugin to provision a Jenkins build slave; the build slave runs the build process and then completed, the build slave container is deleted. This provides each build with a completely pristine environment.



#### Build containers

Build containers are a special type of containers that are only used for build purposes. For Java environments, these containers will typically contain a version of the JDK and build tooling (Maven, Ant, Gradle)

When using build containers Jenkins will be used to handle updating code in the repository but instead of triggering a build tool Jenkins will instead provision a build container, and the entire build process will take place inside the container. Build artifacts that are to be stored may be written back to a host folder for Jenkins to push to an artifact repository, or have the build container push the artifacts itself.



This process requires that build containers are provisioned either via an Execute Shell task that uses docker, or via the Jenkins Docker Build Step plugin.

If using docker, the command would be as follows:

docker run -v $WORKSPACE:/build -it internal.registry/build-java

This assumes that the build container uses /build as the folder where the build process is run, and therefore Jenkins’ workspace folder is mounted there.

### Recommendation

Using a containerized build process makes the process a bit more complicated and is not recommended for scenarios where the build and development landscape is homogenous.

However, it should be used when:

* different projects or components require conflicting versions of development toolset and libraries
* there is a requirement to build and potentially test with different combinations of tools and runtimes, e.g. Java 6, 7 and 8
* to support wildly different build processes across different projects under the same CI instance

## Container-based development

All development work takes place in containers.

This allows to isolate all dependencies and installation of software packages to a container which can be removed and recreated if needed.

It also greatly supports the development of software that may need to be tested under different configurations of a runtime environment, e.g. multiple JVM, Ruby or Python versions.

## Environments as containers

Package a full application environment as a single or multiple containers and

A full application environment can be packaged within a single container to dramatically simplify test preparation as well as test execution.

## Toolbox images

Ephemeral containers that contain specific tools to carry out whichever purpose; packaging these tools, their dependencies and libraries within a single container allows one to distribute such tool without polluting the host with additional packages and libraries.

These containers typically mount the current folder into a folder in the container, and have an entry point that run a specific tool (optionally taking parameters from the host) and then delete themselves after completion.

These container could use a command as follows to run:

docker run -v $(pwd):/src --workdir=/src --user=<some-user> --rm <image-script> "$@"

There will commonly be a script that contains all these parameters, and that simplifies the process for running these containers.

This pattern was first documented here: <https://www.youtube.com/watch?v=M9hBsRUeRdg#t=1398>

# Infrastructure patterns

## Container Restart

As of Docker 1.2, docker run provides an additional flag named --restart that notifies the Docker daemon to restart a given container in case it stops working

## Infrastructure container

Service containers run in each Docker host and provide specific architecture services as part of the infrastructure.

Infrastructure containers are commonly used to support Service discovery, for example, advertising a new application container to that application’s load balancer and reverse proxy.

Other example of infrastructure containers are log aggregation or metrics containers.

Infrastructure containers require at least a basic form of orchestration to ensure that they run in each Docker host and that they are restarted upon failure. For example in CoreOS this can be achieved by running these as “global” units in the cluster with Fleet.

## Sidekick container

The sidekick containers (also known as “buddy” containers) are commonly used to support an application or service container that is unable to perform a specific task for itself.

Sidekick containers are comparable to infrastructure containers, but the main difference between them is that sidekick containers only support specific applications/containers and do not run in all hosts.

## Container Metrics

Container metrics support the analysis of historical container performance over a period of time, and can be used as the input for container monitoring.

### Out of the box metrics support

As any other container that runs in LXC, Docker containers expose runtime metrics under the cgroups pseudo-filesystem, which is usually mounted in the /sys/fs/cgroup folder.

Metrics are provided separately for each container and provide metrics in the following areas: memory, CPU and block IO. Docker container metrics can be found in:

* /sys/fs/cgroup/memory/docker/<container-id> (memory metrics)
* /sys/fs/cgroup/cpu/docker/<container-id>, /sys/fs/cgroup/cpuacct/docker/<container-id> (CPU metrics)
* /sys/fs/cgroup/devices/docker/<container-id>, /sys/fs/cgroup/blkio/docker/<container-id> (block device metrics).

The Docker official documentation provides more insight on what metrics are available for each one of these: <https://docs.docker.com/articles/runmetrics/>.

Network metrics are not exposed by cgroups but instead must either be gathered directly from iptables (if enabled) or from the container’s virtual network interface. According to the Docker reference documentation, the best approach to gather network metrics from a running container is to use the ip netns exec <namespace-name> <command> to access the container’s network namespace and retrieve its data; executing ip netns exec requires establishing a symbolic link from /var/run/netns/<namespace-name> to /proc/<pid>/ns/net so that ip netns can find the network namespace. This can be automated with a script when launching a container.

### Container metrics generation

#### cAdvisor

Google’s cAdvisor is a project that provides additional insight into container runtime performance, resource usage and the various performance characteristics of a running container. cAdvisor also provides a simple web-based UI to view live container, a REST API to extract cAdvisor data and integration with InfluxDB for long-term time data storage.

cAdvisor is run as a separate container on the Docker host. As it requires read-only access to the host’s Docker socket as well as the /sys filesystem, it can gather metrics from all the containers running in the host as well as the host itself.

The cAdvisor container is typically started as follows:

sudo docker run \

-v /var/run:/var/run:rw \

-v /sys:/sys:ro \

-v /var/lib/docker/:/var/lib/docker:ro \

-p 8080:8080 \

-d \

--name=cadvisor \

google/cadvisor:latest

#### In-container metrics generation

Generating metrics in the container can be implemented using one of the existing projects that generate metrics of a running sytem. Collectd is a popular alternative that with a large amount of plugins that extend its capabilities beyond basic monitoring and allow it to generate metrics for components such as nginx, HAproxy and PostgreSQL among others.

In-container metrics generation is more invasive than cAdvisor as it requires that each container runs the collectd daemon, and to do so a process supervisor is needed in the container so that both collectd and the container’s own process can run at the same time.

### Metrics aggregation

cAdvisor by default uses an in-memory backend so all historical data will be lost should cAdvisor be restarted, and it only aggregates data for the containers running in a single Docker host, and a therefore a separate component is needed to aggregate data from all cAdvisor containers within the cluster. Collectd exhibits the same characteristics.

Therefore, an external data store is required to provide long-term time series storage and aggregation across all members of a cluster.

One of the most popular open source projects in the area of time series databases is InfluxDB, which is a scalable database designed to efficiently store, handle and query time series. InfluxDB can be run on a single host or scaled out using shards. Scaling InfluxDB is outside of the scope of this document.

cAdvisor currently provides support for InfluxDB as its storage backend, and collectd can be run alongside a very thin proxy that converts its data formats into something that InfluxDB can store.

This is the overall architecture when combining cAdvisor with InfluxDB and a dashboard:



When using collectd, the architecture is slightly different as collectd runs inside of each container:



### Heapster

Heapster is another Google project, similar in purpose to cAdvisor but tailored for the Kubernetes environment as Heapster can automatically discover all the pods running in a Kubernetes minion.

For storage, Heapster also supports pushing metrics data to an InfluxDB server for long-term storage.

### Dashboards

cAdvisor provides its own dashboard but it only displays the information about the containers running in the host where the cAdvisor service is running, and not all hosts in the cluster.

There are several dashboards available that can pull data from InfluxDB and display it, but Grafana seems to be one of the most popular alternatives so far.

### Service Discovery

In more complex deployment scenarios, cAdvisor services running in Docker hosts should be decoupled from the specific location of their InfluxDB store. A distributed configuration store such as etcd, Consul or Zookeeper could be used to store the current address of the InfluxDB service, and boot the cAdvisor container with the correct address. Stores like etcd also provide notifications of changes, and that could be used to restart the cAdvisor container should the etcd address change.

Please see Service discovery patterns for more details.

### Recommendation

Container metrics provide insight into container runtime performance and can help discover and address container and application performance issues, and should be included in any Docker-enabled infrastructure.

cAdvisor with InfluxDB greatly simplify the process of extracting and aggregating data, should be the preferred approach to implement this capability in a Docker infrastructure for architectures where basic container and operating system metrics are required.

When comparing cAdvisor/Heapster vs. collectd, we find that collectd is more intrusive than cAdvisor as it requires that containers run a process supervisor that starts the collectd daemon as well as whichever process the container is designed to run; however, collectd provides an extensive list of plugins that can generate metrics for a large variety of 3rd party components and tools beyond basic operating system and container metrics, while cAdvisor remains purely focused on container monitoring.

As cAdvisor does not currently extract all the metrics that are available via the cgroups pseudo-filesystem, implementations of this pattern should carefully examine whether all the metrics needed are available and opt for collectd instead. Same applies if any of the more advanced collectd plugins is needed.

# Service discovery

Service discovery patterns support both service registration (a container advertising a specific service for others to consume) as well as service discovery where a container locates its backing services, or is automatically configured to locate its services.

Service discovery helps avoid tight coupling between application components, as application components discover each other’s specific location (hosts/containers) in a decoupled manner at run time as opposed to statically via configuration files.

Typically, service discovery requires three parties:

* A service provider, that provides a specific service to be used by other applications i.e. a load balancer
* A service consumer, that requires the advertised function to perform some (or all) of its functions
* A mechanism to communicate services between consumer and provider

Service discovery supports scenarios such as:

* Application locating database servers to persist their data
* Application instances advertising their presence to a load balancer or reverse proxy that automatically configures itself to route requests to the new instance

Service registration and discovery is a wider topic as it also includes application-supported (“in-application”) registration and discovery, e.g. [Netflix’s Eureka](https://github.com/Netflix/eureka), [AirBnB’s SmartStack](http://nerds.airbnb.com/smartstack-service-discovery-cloud/), but this pattern will focus on an external approach to discovery, where components outside of the application commonly running as containers themselves will support applications and containers to register themselves, and discover other containers with needed services.

## Service discovery with container linking

Container linking effectively makes two containers aware of each other using environment variables, and applications running in these containers are expected to be able to read host and port information from a pre-defined set of environment variables that follow a common naming convention.

Docker supports container linking since its very early versions, and is extensively documented in its [official documentation](https://docs.docker.com/userguide/dockerlinks/).

While container linking is a very simple and effective way to implement a primitive form of service discovery, it has some essential drawbacks:

* Only works when containers are in the same Docker host
* It is not dynamic; linked containers cannot reconfigure themselves if one of the linked containers is restarted, or changes its address or ports

### Recommendation

Given the limitations described above, service discovery with container links can only be used in extremely simplistic scenarios such as local development environments.

## Service discovery with service registry

When using a service registry, a separate component brokers the discovery and registration processes on behalf of providers and consumers.



A distributed data store takes the role of the service registry and coordinates information exchange between service providers and service consumers.

### Service registry

Typically these data stores are also partition-tolerant; from a CAP theorem point of view these usually favour Availability and Partition-tolerance over Consistency to ensure that dependent applications will get a response under most circumstances.

The most commonly used distributed stores for service discovery are:

* etcd
* Consul
* SkyDNS

#### Etcd

Etcd is a distributed key-value store that uses the Raft consensus algorithm to coordinate data replication. It is written in Go and compiled as a static binary, and its tiny footprint makes it ideal to run in Docker infrastructures.

Etcd provides a REST API to maintain keys and values, TTL values for keys and long polling-based notification of changes (clients can listen to updates to keys and reconfigure themselves, for example).

Etcd is part of the CoreOS platform but can also be used as a standalone component, e.g. Kubernetes uses it to maintain cluster information.

#### Consul

Consul is a lightweight orchestration and service discovery service that provides a distributed key-value store, a REST API to manipulate data and DNS server. Consul clusters use a gossip replication protocol and are datacenter-aware, so that a single Consul cluster can be split across multiple datacentres while maintaining replication and keeping discovery to the local datacentre.

Consul’s main differentiator is its DNS server, as it allows service registration using DNS. See DNS-based service discovery for more details.

#### SkyDNS

SkyDNS is a DNS-based service discovery and distributed store that is built on top of etcd. In comparison to Consul’s DNS support, SkyDNS uses SRV records to store service data.

### Service discovery with etcd

Etcd is a reliable and scalable data store, and its small footprint and resource needs make it a very suitable alternative to act as the service registry in a Docker-based infrastructure.

As etcd is only a distributed key-value data store, additional intelligence is required to support each container:

* Help service provider containers expose their services in etcd
* Help service consumer containers query etcd as part of their boot process with the correct configuration, and optionally reconfigure themselves using etcd’s notification support

This pattern does not provide instructions for setting up, clustering, securing and tuning etcd, please refer to the project’s [official documentation](https://coreos.com/docs/cluster-management/setup/cluster-discovery/).

#### Registration and discovery flow

With etcd, the process is as follows:



#### Architecture

The architecture of service discovery and registration is as follows:



The following components are part of the architecture:

* Etcd as the data store for discovery information
* Registrator as an Infrastructure container to support container registration
* Confd to support service consumers dynamically configure themselves based on etcd data

#### Etcd

Etcd acts as the data store for discovery information.

The etcd instance/cluster has to be in a location that is known to all hosts so that they are able to locate it to be able to publish/extract the needed service information.

#### Registrator

[progrium/registrator](https://github.com/progrium/registrator) is a container that runs as an Infrastructure container and provides automatic registration of containers in etcd (Consul and SkyDNS are also supported) optionally with additional container metadata.

The biggest advantage of registrator is that it works for all containers that export a port, and is totally independent from the applications being run in the container. It does so by listening to container events (creation and destruction) via the local Docker socket, capturing the relevant events and sending the corresponding updates to a known etcd endpoint.

Running registration is straightforward:

docker run -d \

-v /var/run/docker.sock:/tmp/docker.sock \

-h $HOSTNAME progrium/registrator etcd://<etcd-server/services

At this point, registrator will automatically register and unregister containers automatically from etcd. If two containers share the same service id or are created from the same image, registrator will identify them as belonging to the same service, which provides support for load balanced containers.

These two containers would be part of the same service:

docker run -d \

--name whoami-1 \

-e SERVICE\_ID=whoami-1 \

-e SERVICE\_NAME=whoami \

-p 8000:8000 \

jwilder/whoami

docker run -d \

--name whoami-2 \

-e SERVICE\_ID=whoami-2 \

-e SERVICE\_NAME=whoami \

-p 8001:8000 \

jwilder/whoami

Environment variables SERVICE\_ID and SERVICE\_NAME are automatically read by registrator and can be used to provide additional service metadata that overrides Docker’s --name.

When querying etcd for keys under the /service/<service-name> namespace (http://<etcd-server>:4001/v2/keys/services/whoami), the following should be visible:

{

"action": "get",

"node": {

"key": "/services/whoami",

"dir": true,

"nodes": [

{

"key": "/services/whoami/whoami-1",

"value": "<host>:8000",

"modifiedIndex": 6,

"createdIndex": 6

},

{

"key": "/services/whoami/whoami-2",

"value": "<host>:8001",

"modifiedIndex": 7,

"createdIndex": 7

}

],

"modifiedIndex": 4,

"createdIndex": 4

}

}

#### Confd

[confd](https://github.com/kelseyhightower/confd) is a project that provides a component that reads values from etcd and creates files based on templates in response to values read or changes from etcd; this is used to dynamically create configuration files with service data such as database URLs when a container boots.

Confd uses configuration files that link etcd URLs to Golang templates, which are processed when confd runs and/or detects a change in the etcd data.

The following is a configuration file that could be used to generate an nginx load balancer configuration file for the whoami service previously run as a Docker container:

[template]

src = "whoami-nginx.conf.tmpl"

dest = "/tmp/whoami-nginx/conf.d/whoami.conf"

keys = [

"/services/whoami"

]

The following template file could be used to generate nginx’s configuration:

upstream whoami {

{{range getvs "/services/whoami/whoami-\*"}}

server {{.}};

{{end}}

}

server {

listen 80;

location / {

proxy\_pass http://whoami;

}

}

A “one time” run of confd can be now triggered:

confd -onetime -backend etcd -node <etcd-host>:4001

And it will generate the following contents:

upstream whoami {

server <host>:8000;

server <host>:8001;

}

server {

listen 80;

location / {

proxy\_pass http://whoami;

}

}

Confd can be triggered as part of the startup script that runs nginx in the nginx load balancer containe, or it can be run in the Docker host that runs the nginx container, have its output be written to a local file and then mount the configuration folder as the nginx container’s appropriate folder in order to make the load balancer container independent from the configuration file that it runs.

### Recommendation

Service registries address all the limitations of linked containers when running in distributed Docker architectures and should be strongly considered as soon as infrastructures are intended to run containers across multiple hosts.

For the role of the service registry, etcd and Consul have been designed as distributed and available data stores, and make them much more suitable choices for the role of service registry.

However, service registries require additional infrastructure and sidekick containers to support an end-to-end discovery process that is independent from the containerized applications, thereby increasing the complexity of the overall architecture.

Given the importance of the role of these infrastructure and sidekick containers, additional components such as systemd or CoreOS’ fleet are required to ensure that these containers are managed when new nodes are added to the cluster, and restarted if they crash or stop working.

Additional components such as registrator and confd simplify the task of dynamically exposin application backend data and configuring applications to use the data exposed in etcd, but their role is not strictly necessary as we expect that applications will evolve towards being able to handle etc-based configuration on their own.

## DNS-based service discovery

<TODO>

### Recommendation

<TODO>

# Orchestration patterns

In progress.

# Anti-patterns

## Container Remote Access

In progress.

# References

Encrypting Docker containers in a host: https://launchbylunch.com/posts/2014/Jan/13/encrypting-docker-on-digitalocean/