

**Exhibit 1 to
Intellectual Ventures I LLC and Intellectual Ventures II LLC
Amended Infringement Contentions**

**Amended Infringement Claim Chart for
U.S. Patent No. 8,332,844 (“the ’844 Patent”)**

The Accused Systems and Services include, without limitation, Liberty Mutual’s systems that utilize Docker; all past, current, and future systems and services that operate in the same or substantially similar manner as the specifically identified systems and services; and all past, current and future Liberty Mutual’s systems and services that have the same or substantially similar features as the specifically identified systems and services (“Liberty Mutual Count I Systems and Services”).¹ Upon information and belief, Liberty Mutual’s systems and services directly infringe the asserted claims of the ’844 Patent through at least using Docker in its various systems and offerings. For example, as shown in the source code revisions below, Liberty Mutual uses Docker for various applications. In addition, Liberty Mutual directly infringes the ’844 Patent via testing the various systems and offerings. On information and belief, Liberty Mutual also indirectly infringes the ’844 Patent through inducement. On information and belief, Liberty Mutual also contributes to the direct infringement through its vendors and other third parties’ uses of Liberty Mutual’s Systems and Services that utilize Docker by providing components relating to Docker that have no substantial non-infringing uses. On information and belief, Liberty Mutual with knowledge at least at the time of filing the complaint, supplies its customers, employees, and/or vendors and other third parties with access to its systems and services through various client offerings and/or internal services that allows its customers, employees, and/or vendors and other third parties to use the claimed method. On information and belief, Liberty Mutual servers, or servers that are run on behalf of Liberty Mutual, are involved and/or perform the method of the asserted claims of the ’844 Patent. On information and belief, Liberty Mutual provides instructions on how to use its various offerings and the use of which results in infringement of the ’844 Patent claims through performance of the claimed methods.

Pursuant to P.R. 3-1(b), IV identifies Docker as used in Liberty Mutual’s systems and offerings as the Accused Instrumentality. Information regarding how specifically this Docker is used for what product and offerings is non-public. IV intends to obtain discovery regarding how the Docker is used in Liberty Mutual’s products and services and will update its infringement positions once this discovery is completed. For example, IV served a 30(b)(6) deposition notice on Liberty Mutual on June 26, 2024, seeking information relating to

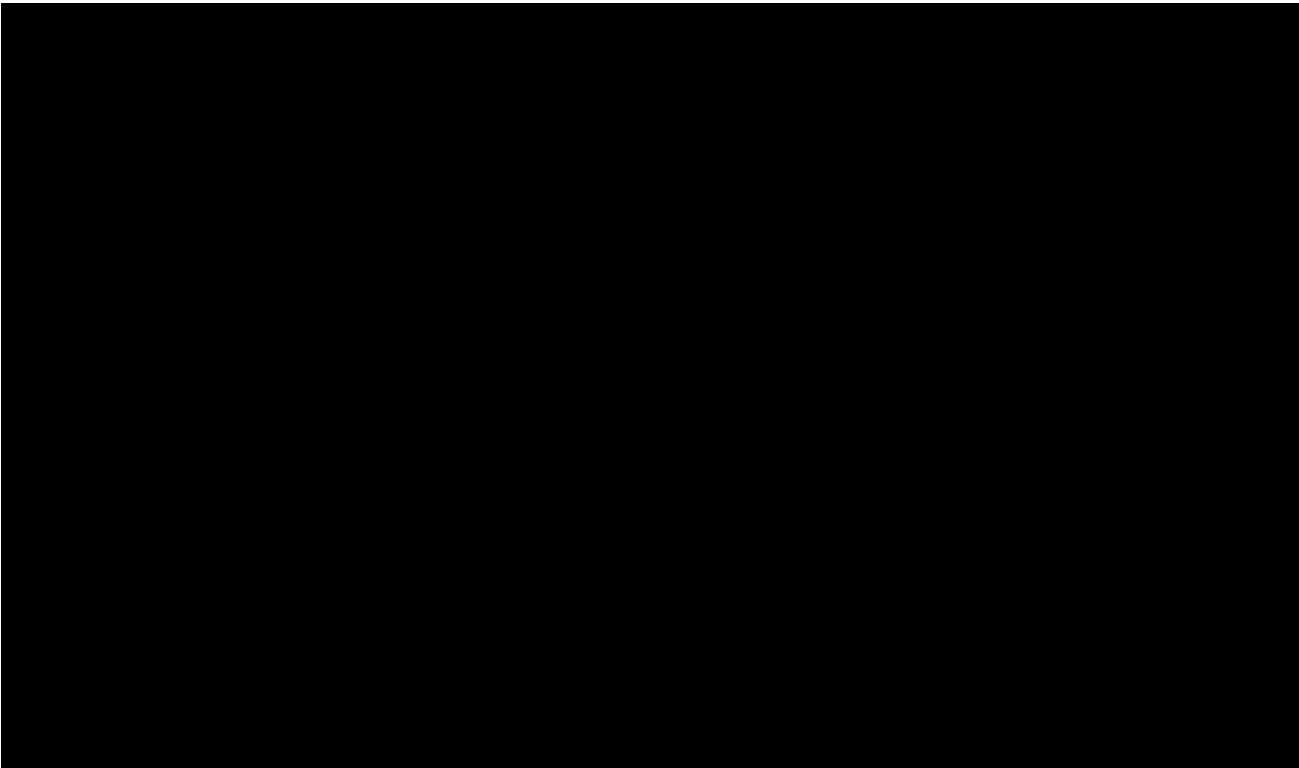
¹ For the avoidance of doubt, Plaintiffs do not accuse the public clouds of Defendants, to the extent those services are provided by a cloud provider with a license to Plaintiffs’ patents that covers Defendants’ activities. [REDACTED]

[REDACTED] IV accuses Liberty Mutual private clouds that implement Docker and non-licensed public clouds that Liberty Mutual uses to support Docker for its systems and services. To the extent any of these licenses are relevant to Defendants’ activities, Plaintiffs will meet and confer with Defendants about the impact of such license(s).



Liberty Mutual's use and implementation of Docker, including use and implementation of Docker to support services that are offered to Liberty Mutual's customers and/or employees. This information requested by IV includes for example technical documents that show the operation of Liberty Mutual's use and implementation of Docker, Liberty Mutual source code relating to Docker, whether third parties have information relating to Liberty Mutual's use and implementation of Docker, and other technical information. *See* Plaintiffs' Initial 30(b)(6) Notice to Defendants (Liberty Mutual) at 9-10. IV served an amended 30(b)(6) notice on August 16, 2024, requesting additional relevant information. *See generally* Plaintiffs' Amended 30(b)(6) Notice to Defendants (Liberty Mutual). IV has also served 30(b)(1) deposition notices for several technical Liberty Mutual employees that likely possess knowledge of Liberty Mutual's use and implementation of Docker. To date, Liberty Mutual has not offered any deposition dates for either potential corporate witnesses or witnesses having personal knowledge of Liberty Mutual's use and implementation of Docker. These amended contentions are based on information available to IV as of the date of service of these contentions, and are served without the benefit of even a single technical deposition to understand how Liberty Mutual uses and implements Docker in its systems and services. Further, fact discovery remains open and Plaintiffs intend to serve additional discovery to better understand Liberty Mutual's use and implementation of Docker. Accordingly, Plaintiffs reserve all rights to amend these contentions consistent with the local rules. Further, IV has filed a motion to compel seeking discovery including the identification of Liberty Mutual software applications that utilize Docker, and reserve the right to supplement these contentions once the Court has resolved this pending dispute.



U.S. Patent No. 8,332,844	
Claim 7	Liberty Mutual Count I Systems and Services
7[pre]. A method for providing data to a plurality of compute nodes, comprising:	Upon information and belief, Liberty Mutual’s systems perform “a method for providing data to a plurality of compute nodes.” Liberty Mutual uses Docker . Liberty Mutual uses docker images (e.g., providing data) for many containers (e.g., a plurality of compute nodes) that run an instance of the image. Upon information and belief, Liberty Mutual’s system comprises the service/application the Cost Explorer BI Architecture which includes the usage of Docker through their private network supported with cloudforge (bottom left in figure below) in their Cost Information Sources network.  See LM00004304.

[REDACTED]

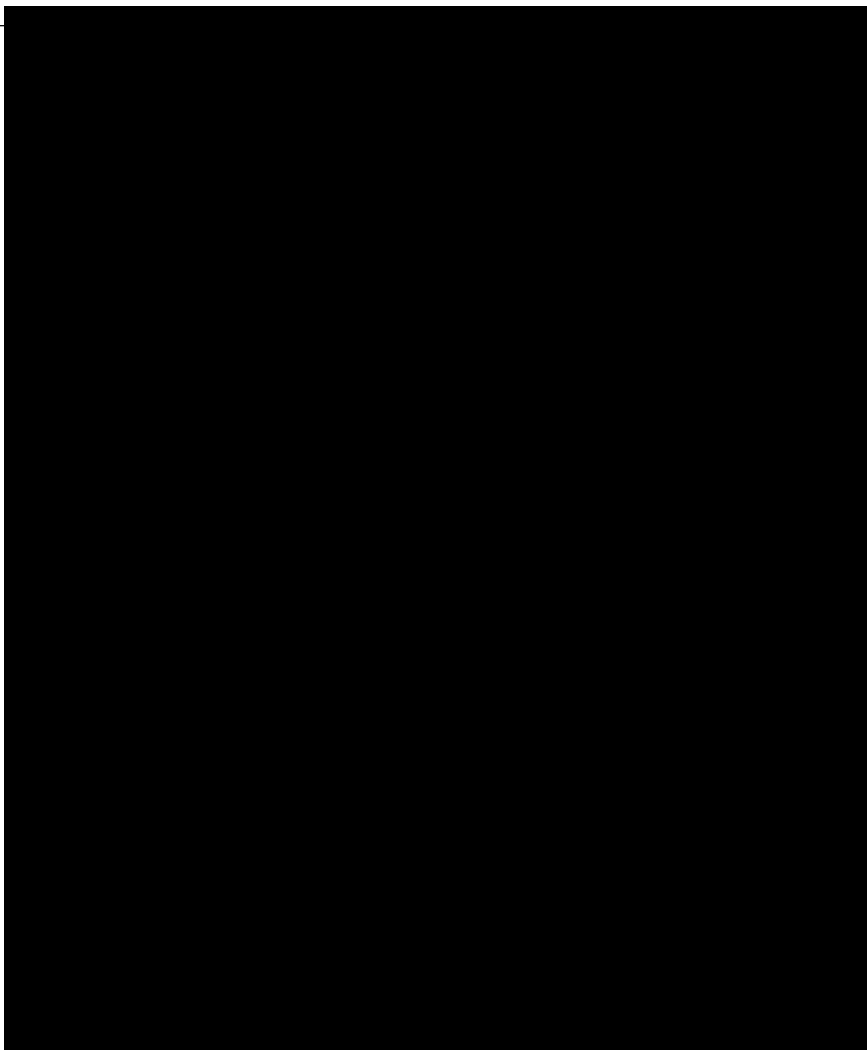
Upon information and belief, Liberty Mutual's systems support Docker Swarm. Docker Swarm is leveraged by Liberty Mutual systems to manage container workloads for Jenkins and Bamboo pipelines. [REDACTED]

[REDACTED]

[REDACTED]

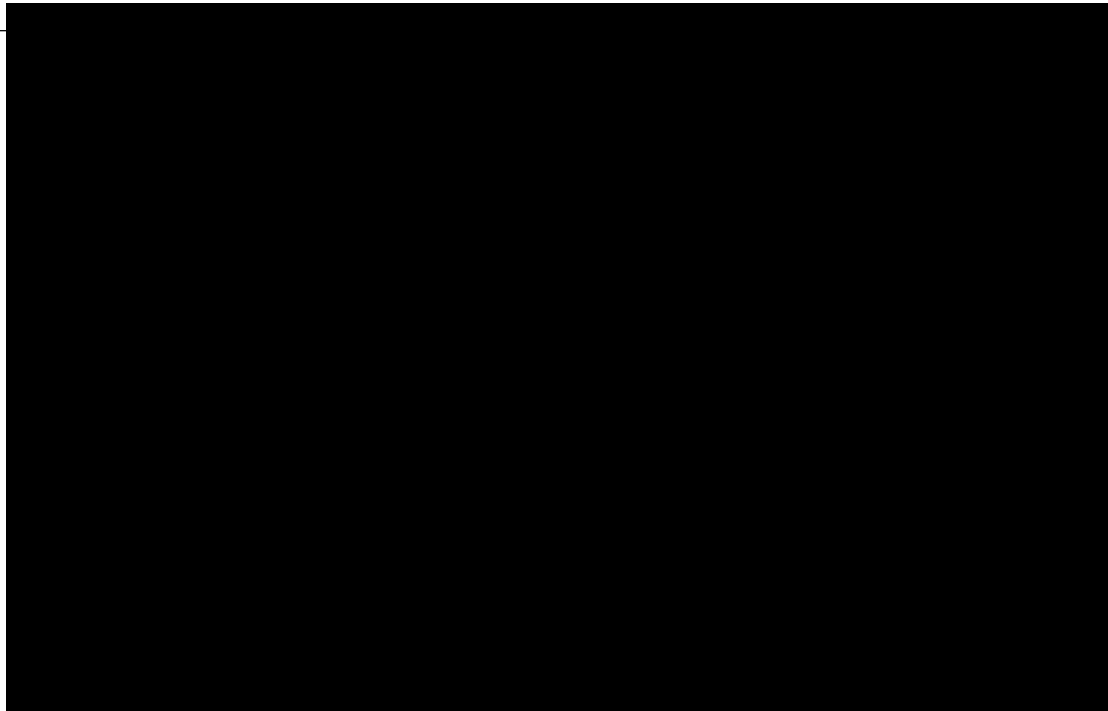
See LM00007344.

[REDACTED]



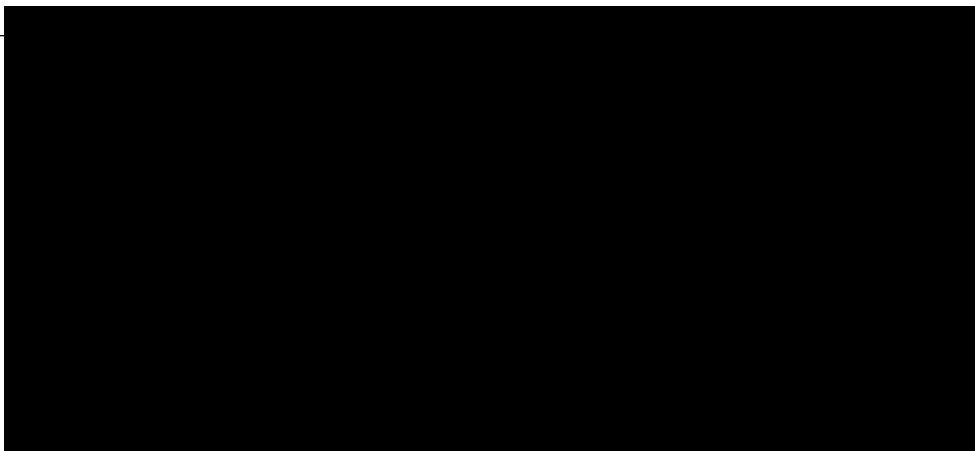
See LM00007544.

[REDACTED]



See LM00007595, LM00007596.

As a further example, upon information and belief, Bamboo/Jenkins caches Docker compose files processes into the SPI Frontend. When invoked, a backend Jenkins Job will map to a specified Docker Cluster using the Card Catalog, which involves non-licensed Liberty Mutual clouds.



See LM00008689.

For example, the usage of Docker includes providing Docker images, which have all the file systems, binaries, libraries, and configuration (e.g., providing data) for respective container instances (e.g., a plurality of compute nodes). Docker containerization provides scalability of applications regardless of the system or environment deployed.

The Docker platform

Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security lets you to run many containers simultaneously on a given host. Containers are lightweight and contain everything needed to run the application, so you don't need to rely on what's installed on the host. You can share containers while you work, and be sure that everyone you share with gets the same container that works in the same way.

Deployment and orchestration

Containerization provides an opportunity to move and scale applications to clouds and data centers. Containers effectively guarantee that those applications run the same way anywhere, allowing us to quickly and easily take advantage of all these environments. Additionally, as we scale our applications up, we need some tooling to help automate the maintenance of those applications, enable the replacement of failed containers automatically, and manage the roll-out of updates and reconfigurations of those containers during their lifecycle.

See <https://docs.docker.com/get-started/orchestration/> (last accessed on November 13, 2023).

As an example (on information and belief), Liberty Mutual's use of Docker includes providing Docker images. Images include all the file systems, binaries, libraries, and configurations (e.g., providing data), to respective containers (e.g., a plurality of compute nodes) that run an instance of that image.

A container image is a standardized package that includes all of the files, binaries, libraries, and configurations to run a container.

See <https://docs.docker.com/guides/docker-concepts/the-basics/what-is-an-image/> (last accessed August 5, 2024).

CONTAINERS

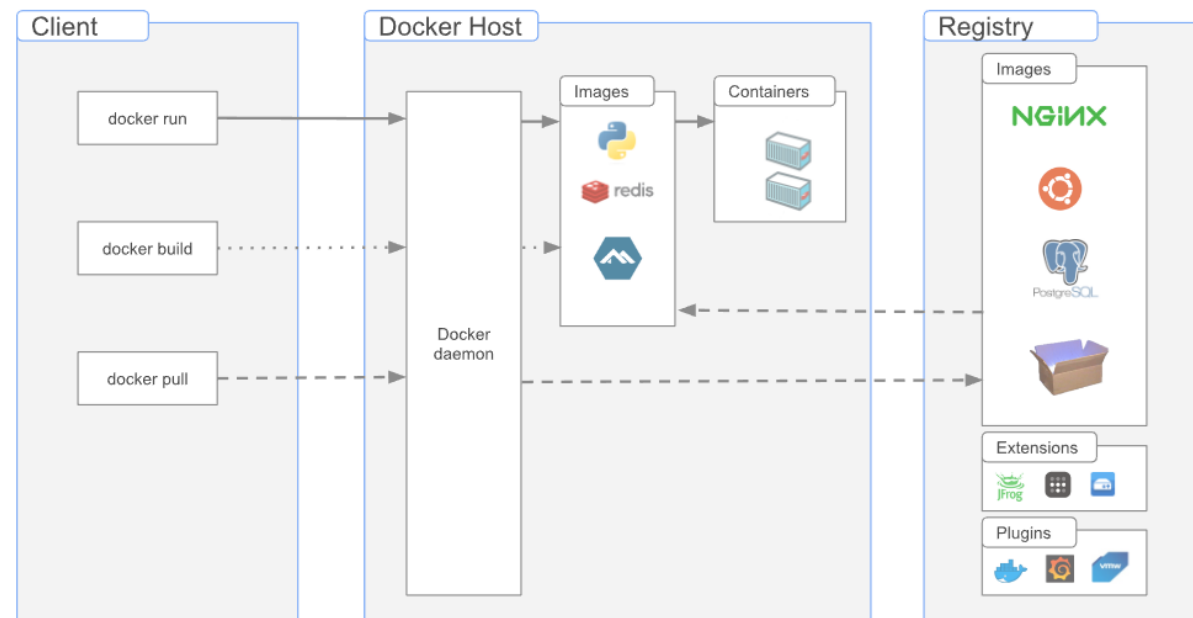
A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

See <https://docs.docker.com/engine/docker-overview/> (last accessed March 15, 2024).

A running container uses an isolated filesystem. This isolated filesystem is provided by an image, and the image must contain everything needed to run an application - all dependencies, configurations, scripts, binaries, etc. The image also contains other configurations for the container, such as environment variables, a default command to run, and other metadata.

See <https://docs.docker.com/guides/workshop/> (last accessed August 5, 2024).

As an example below, Docker's architecture involves a client communicating with a server/host. At each host, there is a Docker daemon responsible for managing Docker images (e.g., providing data) to containers (e.g., a plurality of compute nodes). Docker containerization provides scalability of applications regardless of the system or environment deployed.

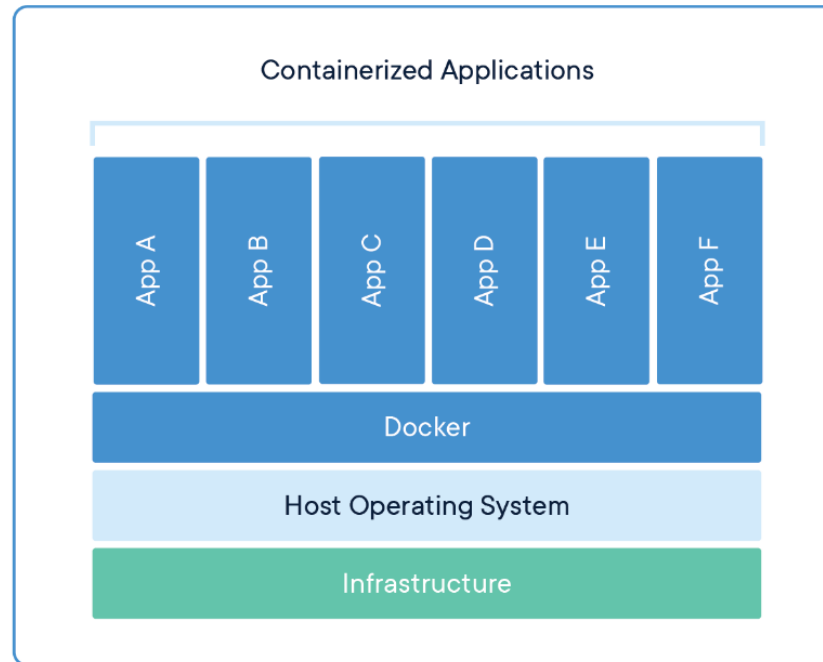


See <https://docs.docker.com/get-started/overview/> (last accessed March 15, 2024).

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.

Id.

As another example, Docker can facilitate the instances of applications through Docker containers by enabling a way to separate infrastructure and required installations. Containerized applications (e.g., a plurality of compute nodes) and their supporting components are easily distributable across different working environments by the Docker platform.

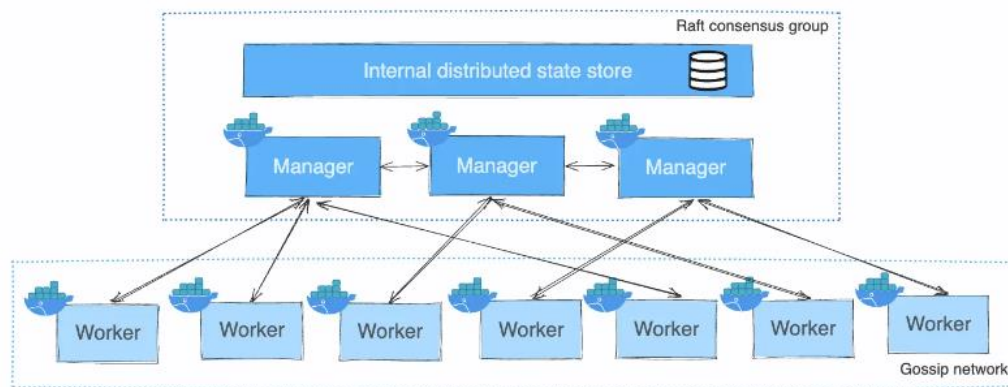


See <https://docs.docker.com/guides/docker-overview/> (last accessed August 5, 2024).

As an example (on information and belief), Liberty Mutual's systems use Docker Swarm, which consists of one or more Docker Engines that run on one or more physical or virtual machines (e.g., a plurality of compute nodes).

Swarm mode lets you create a cluster of one or more Docker Engines called a swarm. A swarm consists of one or more nodes: physical or virtual machines running Docker Engine.

There are two types of nodes: [managers](#) and [workers](#).



See <https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/> (last accessed August 6, 2024).

Liberty Mutual's systems include Docker, which provides docker images to containers.

See <https://www.itprotoday.com/containerization/liberty-mutual-s-container-deployment-from-docker-to-kubernetes> (last accessed August 5, 2024).

See <https://searchjobs.libertymutualgroup.com/careers/job/618493514108?microsite=libertymutual.com> (last accessed November 13, 2023).

See <https://searchjobs.libertymutualgroup.com/careers/job/618496258214?microsite=libertymutual.com> (last accessed on March 19, 2022).

See <https://searchjobs.libertymutualgroup.com/careers/job/618495353274?microsite=libertymutual.com> (last accessed on March 19, 2022).

Source Code Evidence

[illegible]

[REDACTED]

	[REDACTED]	
	[REDACTED]	
	[REDACTED]	
	[REDACTED]	
	[REDACTED]	
	[REDACTED]	
	[REDACTED]	
	[REDACTED]	
	[REDACTED]	
	[REDACTED]	
	[REDACTED]	
	[REDACTED]	
	[REDACTED]	
	[REDACTED]	
	[REDACTED]	
	[REDACTED]	
	[REDACTED]	
	[REDACTED]	
	[REDACTED]	
	[REDACTED]	

[REDACTED]

		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]

[REDACTED]

		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	

[REDACTED]

		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	

[REDACTED]

		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	

[REDACTED]

		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	

[REDACTED]

		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	

[REDACTED]

		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]

[illegible]

[REDACTED]

		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	

[REDACTED]

		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]

[REDACTED]

	[REDACTED] [REDACTED] [REDACTED] [REDACTED] [REDACTED]
--	---

[REDACTED]

		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	

[REDACTED]

		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	

[REDACTED]

		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	

[REDACTED]

		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	

[REDACTED]

		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	

[REDACTED]

		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	

[REDACTED]

		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	

[REDACTED]

		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	

[REDACTED]

	[REDACTED] [REDACTED] [REDACTED] [REDACTED] [REDACTED] [REDACTED] [REDACTED]
--	---

[REDACTED]

	[REDACTED]	
	[REDACTED]	[REDACTED]
	[REDACTED]	
	[REDACTED]	
	[REDACTED]	
	[REDACTED]	
	[REDACTED]	
	[REDACTED]	
	[REDACTED]	
	[REDACTED]	
	[REDACTED]	
	[REDACTED]	
	[REDACTED]	
	[REDACTED]	
	[REDACTED]	

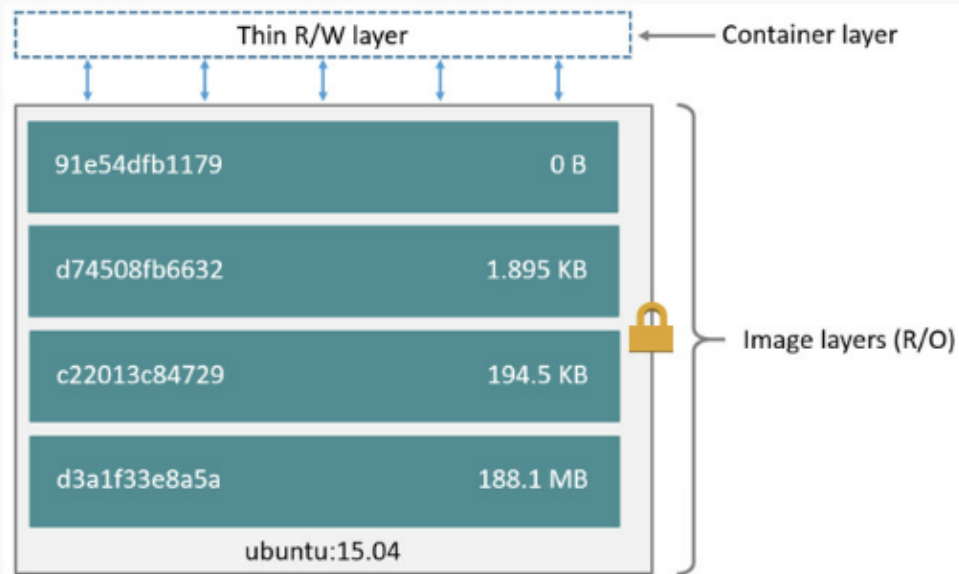
[REDACTED]

	[REDACTED] [REDACTED]
[7a] storing blocks of a root image of said compute nodes on a first storage unit;	Upon information and belief, Liberty Mutual’s system performs “storing blocks of a root image of said compute nodes on a first storage unit.” For example, Docker images are made up of layers where about each instruction of the DockerFile is a layer (e.g., blocks of a root image). Images can either be stored locally on your computer or in an image registry (e.g., first storage unit), where Docker Hub is the default configured registry (e.g., storing blocks...on a first storage unit). The diagram shows the workflow from a Dockerfile to a Docker Image. On the left, a box labeled 'Dockerfile' contains the following instructions: <pre>FROM golang:1.20-alpine WORKDIR /src COPY . . RUN go mod download RUN go build -o /bin/client ./cmd/client RUN go build -o /bin/server ./cmd/server ENTRYPOINT ["/bin/server"]</pre> An arrow points from the Dockerfile to a central circle labeled 'Builder' which contains a gear icon. Another arrow points from the Builder to a box on the right labeled 'Image'. This 'Image' box contains a stack of 'Layers', each corresponding to an instruction in the Dockerfile: - FROM golang:1.20-alpine - WORKDIR /src - COPY . . - RUN go mod download - RUN go build -o /bin/client ./cmd/client - RUN go build -o /bin/server ./cmd/server - ENTRYPOINT ["/bin/server"] See https://docs.docker.com/build/guide/layers/ (last accessed August 5, 2024).

Images and layers

A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:

The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image.



See <https://docs.docker.com/storage/storagedriver/> (last accessed on November 13, 2023).

For example, Docker images are either stored locally on a computer or in Docker Hub (e.g., first storage unit), the configured default registry to look for images. A registry of these images can have multiple versions, for example, one for each release version of a project in a repository.

Docker Registry

● Important

This page contains information about hosting your own registry using the [open source Docker Registry](#). For information about Docker Hub, which offers a hosted registry with additional features such as teams, organizations, web hooks, automated builds, etc, see [Docker Hub](#).

What it is

The Registry is a stateless, highly scalable server side application that stores and lets you distribute Docker images. The Registry is open-source, under the permissive [Apache license](#). You can find the source code on [GitHub](#).

Why use it

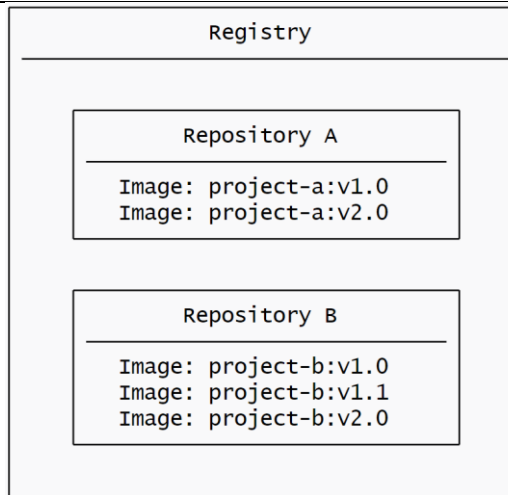
You should use the Registry if you want to:

- Tightly control where your images are being stored
- Fully own your images distribution pipeline
- Integrate image storage and distribution tightly into your in-house development workflow

See <https://distribution.github.io/distribution/> (last accessed on November 13, 2023).

A Docker *registry* stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry.

See <https://docs.docker.com/get-started/overview/> (last accessed March 15, 2024).



See <https://docs.docker.com/guides/docker-concepts/the-basics/what-is-a-registry/> (last accessed August 5, 2024).

See also claim limitation 7[pre], *supra*.

As a further example, images can be pulled, removed, and edited in a registry. The Docker SDK for Python implements these functionalities, but from within Python applications. The class `Image` represents an image on a server including the function `remove` to remove a said image. It also includes the function `save` to package the image up to, for example, be moved from one registry to another.

```
class Image(Model):
    """
    An image on the server.
    """
    def __repr__(self):
        tag_str = "", "".join(self.tags)
        return f"<{self.__class__.__name__}: '{tag_str}'>"
```



	<pre>def remove(self, force=False, noprune=False): """ Remove this image. Args: force (bool): Force removal of the image noprune (bool): Do not delete untagged parents Raises: :py:class:`docker.errors.APIError` If the server returns an error. """ return self.client.api.remove_image(self.id, force=force, noprune=noprune,)</pre>
--	--



	<pre>def save(self, chunk_size=DEFAULT_DATA_CHUNK_SIZE, named=False): """ Get a tarball of an image. Similar to the ``docker save`` command. Args: chunk_size (int): The generator will return up to that much data per iteration, but may return less. If ``None``, data will be streamed as it is received. Default: 2 MB named (str or bool): If ``False`` (default), the tarball will not retain repository and tag information for this image. If set to ``True``, the first tag in the :py:attr:`~tags` list will be used to identify the image. Alternatively, any element of the :py:attr:`~tags` list can be used as an argument to use that specific tag as the saved identifier. Returns: (generator): A stream of raw archive data. Raises: :py:class:`~docker.errors.APIError` If the server returns an error. Example: >>> image = cli.images.get("busybox:latest") >>> f = open('/tmp/busybox-latest.tar', 'wb') >>> for chunk in image.save(): >>> f.write(chunk) >>> f.close() """</pre>
--	---

See <https://github.com/docker/docker-py/blob/a3652028b1ead708bd9191efb286f909ba6c2a49/docker/models/images.py#L13> (last accessed August 3, 2024).

Furthermore, the class Collection is a base class for representing all objects of a particular type, where the class ImageCollection is built from. Objects of the class Image (e.g., root image) have a class ImageCollection, like a registry (e.g., first storage unit), to where they belong. Images in these collections can be built, loaded, or pulled by a container instance.

```
class Collection:
    """
    A base class for representing all objects of a particular type on the
    server.
    """
    #: The type of object this collection represents, set by subclasses
    model = None

    def __init__(self, client=None):
        #: The client pointing at the server that this collection of objects
        #: is on.
        self.client = client
```

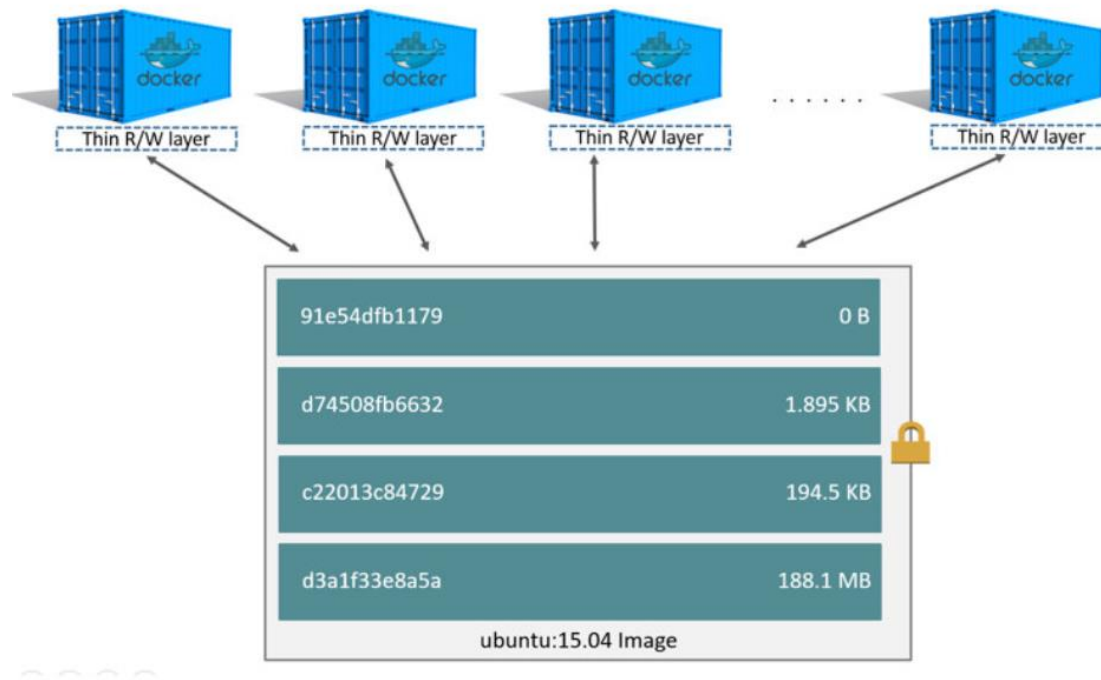
See <https://github.com/docker/docker-py/blob/a3652028b1ead708bd9191efb286f909ba6c2a49/docker/models/resource.py> (last accessed August 3, 2024).



	<pre>class ImageCollection(Collection): model = Image def build(self, **kwargs): """ Build an image and return it. Similar to the ``docker build`` command. Either ``path`` or ``fileobj`` must be set. If you already have a tar file for the Docker build context (including a Dockerfile), pass a readable file-like object to ``fileobj`` and also pass ``custom_context=True``. If the stream is also compressed, set ``encoding`` to the correct value (e.g. ``gzip``). If you want to get the raw output of the build, use the :py:meth:`~docker.api.build.BuildApiMixin.build` method in the low-level API. Args: path (str): Path to the directory containing the Dockerfile fileobj: A file object to use as the Dockerfile. (Or a file-like object) tag (str): A tag to add to the final image quiet (bool): Whether to return the status nocache (bool): Don't use the cache when set to ``True`` rm (bool): Remove intermediate containers. The ``docker build`` command now defaults to ``--rm=true``, but we have kept the old default of ``False`` to preserve backward compatibility timeout (int): HTTP timeout custom_context (bool): Optional if using ``fileobj`` encoding (str): The encoding for a stream. Set to ``gzip`` for compressing pull (bool): Downloads any updates to the FROM image in Dockerfiles forcerm (bool): Always remove intermediate containers, even after unsuccessful builds dockerfile (str): path within the build context to the Dockerfile buildargs (dict): A dictionary of build arguments container_limits (dict): A dictionary of limits applied to each container created by the build process. Valid keys: - memory (int): set memory limit for build - memswap (int): Total memory (memory + swap), -1 to disable swap - cpushares (int): CPU shares (relative weight) - cpusetcpus (str): CPUs in which to allow execution, e.g., ``0-3`` or ``0,1`` shmsize (int): Size of ``/dev/shm`` in bytes. The size must be greater than 0. If omitted the system uses 64MB labels (dict): A dictionary of labels to set on the image cache_from (list): A list of images used for build cache resolution target (str): Name of the build-stage to build in a multi-stage Dockerfile network_mode (str): networking mode for the run commands during build squash (bool): Squash the resulting images layers into a single layer. extra_hosts (dict): Extra hosts to add to /etc/hosts in building containers, as a mapping of hostname to IP address. platform (str): Platform in the format ``os[/arch[/variant]]``. isolation (str): Isolation technology used during build. Default: 'None'. use_config_proxy (bool): If ``True``, and if the docker client configuration file (``~/.docker/config.json`` by default) contains a proxy configuration, the corresponding environment variables will be set in the container being built. Returns: (tuple): The first item is the :py:class:`Image` object for the image that was built. The second item is a generator of the build logs as JSON-decoded objects.</pre>
--	---

See <https://github.com/docker/docker-py/blob/a3652028b1ead708bd9191efb286f909ba6c2a49/docker/models/images.py#L13> (last accessed August 3, 2024).

As a further example, the Docker containers (e.g., compute nodes) are depicted as the top blue containers. Each one has a thin R/W layer in the figure below. The Ubuntu image, containing multiple layers (e.g., blocks of the root image), is stored on the Docker Registry and represented by the cyan blocks below the blue containers.



See <https://docs.docker.com/storage/storagedriver> (last accessed March 15, 2024)



An *image* is a read-only template with instructions for creating a Docker container. Often, an image is *based on* another image, with some additional customization. For example, you may build an image which is based on the `ubuntu` image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.

See <https://docs.docker.com/get-started/overview/> (last accessed March 15, 2024).

As a further example, the class `Container` represents a container object based on an image, where local attributes are cached and have different configurable attributes.

```
class Container(Model):
    """ Local representation of a container object. Detailed configuration may
        be accessed through the :py:attr:`attrs` attribute. Note that local
        attributes are cached; users may call :py:meth:`reload` to
        query the Docker daemon for the current properties, causing
        :py:attr:`attrs` to be refreshed.
    """
```

See <https://github.com/docker/docker-py/blob/a3652028b1ead708bd9191efb286f909ba6c2a49/docker/models/containers.py> (last accessed August 3, 2024).

Furthermore, the class `Collection` is a base class for representing all objects of a particular type, where the class `ContainerCollection` is built. This class includes the functionalities for managing multiple containers and their images (e.g., root image of said compute nodes).



	<pre>class Collection: """ A base class for representing all objects of a particular type on the server. """ #: The type of object this collection represents, set by subclasses model = None def __init__(self, client=None): #: The client pointing at the server that this collection of objects #: is on. self.client = client</pre> See https://github.com/docker/docker-py/blob/a3652028b1ead708bd9191efb286f909ba6c2a49/docker/models/resource.py (last accessed August 3, 2024).
--	--



```
class ContainerCollection(Collection):
    model = Container

    def run(self, image, command=None, stdout=True, stderr=False,
            remove=False, **kwargs):
        """
        Run a container. By default, it will wait for the container to finish
        and return its logs, similar to ``docker run``.

        If the ``detach`` argument is ``True``, it will start the container
        and immediately return a :py:class:`Container` object, similar to
        ``docker run -d``.
```

Example:

Run a container and get its output:

```
>>> import docker
>>> client = docker.from_env()
>>> client.containers.run('alpine', 'echo hello world')
b'hello world\\n'
```

Run a container and detach:

```
>>> container = client.containers.run('bfirsh/reticulate-splines',
                                       detach=True)

>>> container.logs()
'Reticulating spline 1...\\nReticulating spline 2...\\n'
```

See <https://github.com/docker/docker-py/blob/a3652028b1ead708bd9191efb286f909ba6c2a49/docker/models/containers.py> (last accessed August 3, 2024).

On information and belief, a further example of the “...blocks of a root image...” can be seen in the description below. See <https://docs.docker.com/storage/storagedriver/> (last accessed March 15, 2024)

Images and layers

A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:

Id.

This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:18.04` image. The `LABEL` command only modifies the image's metadata, and does not produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which does not produce an image layer.

Id.

The major difference between a container and an image is the top writable layer. All writes to the container that add new or modify existing data are stored in this writable layer. When the container is deleted, the writable layer is also deleted. The underlying image remains unchanged.

Id.

See <https://docs.docker.com/storage/storagedriver/overlayfs-driver/> (last accessed March 15, 2024).



Id.

[REDACTED]

Source Code Evidence

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]

[REDACTED]

		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	

[REDACTED]

		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	

[REDACTED]

		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	

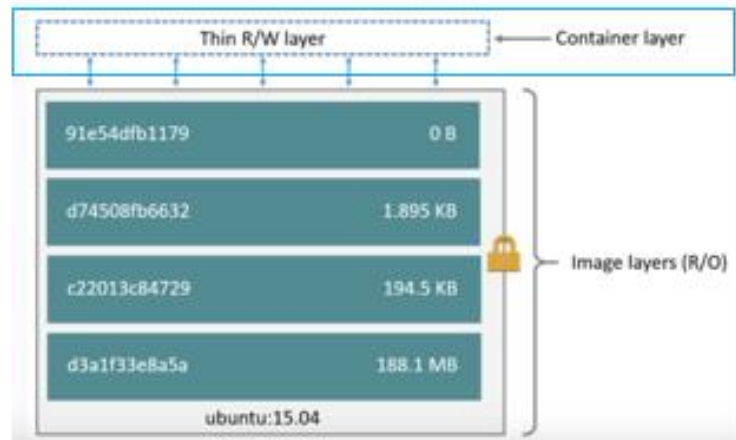
[REDACTED]

	[REDACTED]
[7b] storing leaf images for respective compute nodes on respective second storage units,	Upon information and belief, Liberty Mutual’s systems perform “storing leaf images for respective compute nodes on respective second storage units.” For example, when creating a new Docker container, any new files, modifications, or deletions are written to the thin R/W layer on top of the root image layers (e.g., leaf image for respective compute node) located in the local Docker storage area (e.g., respective second storage units). When you run an image and generate a container, you add a new writable layer, also called the container layer, on top of the underlying layers. All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this writable container layer. See https://docs.docker.com/develop/develop-images/dockerfile_best-practices/ (last accessed on March 15, 2024). When you use <code>docker pull</code> to pull down an image from a repository, or when you create a container from an image that does not yet exist locally, each layer is pulled down separately, and stored in Docker's local storage area, which is usually <code>/var/lib/docker/</code> on Linux hosts. You can see these layers being pulled in this example: See https://docs.docker.com/storage/storagedriver/ (last accessed on November 10, 2023). As a further example, Docker images are runnable containers made up of instructions, where each instruction is about a layer. Container images (e.g., leaf images), represented by the thin R/W and blue containers below, store any new data or modifications of the root image at the host of the container (e.g., storing on respective second storage units).

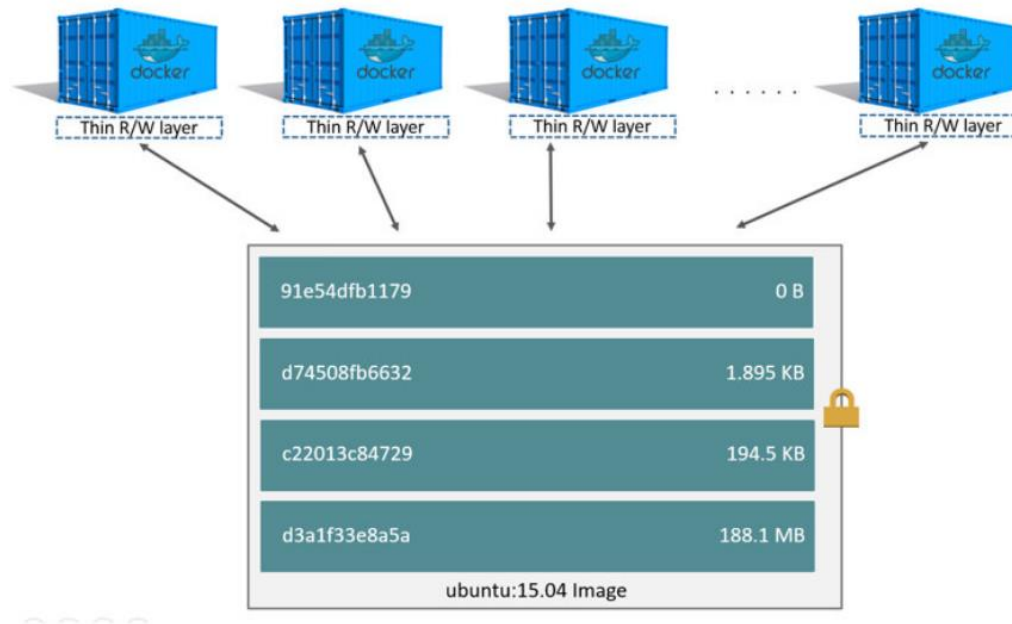
Images and layers

A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:

The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image.



See <https://docs.docker.com/storage/storagedriver/> (last accessed on November 10, 2023).



See <https://docs.docker.com/storage/storagedriver> (last accessed March 15, 2024).

See also description below of containers. <https://docs.docker.com/engine/docker-overview/> (last accessed March 15, 2024).

CONTAINERS

A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

Id.

	A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are not stored in persistent storage disappear. <i>Id.</i> As another example (on information and belief), any changes made by the new container instance are performed and written into the writable container layer (e.g., leaf images for respective compute nodes on respective storage units). When deleting the container contents, any respective changes will also be removed. When you run an image and generate a container, you add a new <i>writable layer</i> (the “container layer”) on top of the underlying layers. All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. See https://docs.docker.com/develop/develop-images/dockerfile_best-practices/ (last accessed March 15, 2024) Container and layers The major difference between a container and an image is the top writable layer. All writes to the container that add new or modify existing data are stored in this writable layer. When the container is deleted, the writable layer is also deleted. The underlying image remains unchanged. See https://docs.docker.com/storage/storagedriver/ (last accessed March 15, 2024) Docker uses storage drivers to manage the contents of the image layers and the writable container layer. Each storage driver handles the implementation differently, but all drivers use stackable image layers and the copy-on-write (CoW) strategy. <i>Id.</i>
--	--

[REDACTED]

When you use `docker pull` to pull down an image from a repository, or when you create a container from an image that does not yet exist locally, each layer is pulled down separately, and stored in Docker's local storage area, which is usually `/var/lib/docker/` on Linux hosts. You can see these layers being pulled in this example:

Id.

See also <https://docs.docker.com/storage/storagedriver/overlayfs-driver/> (last accessed March 15, 2024) (showing graphical representation)



Id.

Source Code Evidence

[REDACTED]

[REDACTED]

		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]

[REDACTED]

		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	

[REDACTED]

		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	

[REDACTED]

		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	

[REDACTED]

	[REDACTED]
[7c] said leaf images including only additional data blocks not previously contained in said root image and changes made by	Upon information and belief, Liberty Mutual's systems also have "leaf images including only additional data blocks not previously contained in said root image and changes made by respective compute nodes to the blocks of the root image." For example, when a Docker image is run, the container that is generated includes a writable layer on top of the root image (e.g., leaf images). Any new data such as new files (e.g., including only additional data blocks not previously contained in said root image) or modifications are



respective compute nodes to the blocks of the root image,	written onto this writable layer by the new container (e.g., changes made by respective compute nodes to the blocks of the root image). See, for example, the discussion above with respect to 7[a] and 7[b]. When you run an image and generate a container, you add a new writable layer, also called the container layer, on top of the underlying layers. All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this writable container layer. See https://docs.docker.com/develop/develop-images/dockerfile_best-practices/ (last accessed on March 15, 2024). Docker uses storage drivers to manage the contents of the image layers and the writable container layer. Each storage driver handles the implementation differently, but all drivers use stackable image layers and the copy-on-write (CoW) strategy. Copy-on-write is a strategy of sharing and copying files for maximum efficiency. If a file or directory exists in a lower layer within the image, and another layer (including the writable layer) needs read access to it, it just uses the existing file. The first time another layer needs to modify the file (when building the image or running the container), the file is copied into that layer and modified. This minimizes I/O and the size of each of the subsequent layers. See https://docs.docker.com/storage/storagedriver/ (last accessed on November 10, 2023). As a further example (on information and belief) this limitation is met using copying templates and other features described below. Any changes the containers make to the file systems, libraries, etc., are stored on the thin writable container layer (e.g., leaf images including only additional data blocks not previously
--	--

contained in said root image). On the other hand, the base image is read-only (e.g., changes made by respective compute nodes to the blocks of the root image).

Copying makes containers efficient

When you start a container, a thin writable container layer is added on top of the other layers. Any changes the container makes to the filesystem are stored here. Any files the container does not change do not get copied to this writable layer. This means that the writable layer is as small as possible.

Docker uses storage drivers to manage the contents of the image layers and the writable container layer. Each storage driver handles the implementation differently, but all drivers use stackable image layers and the copy-on-write (CoW) strategy.

When you use `docker pull` to pull down an image from a repository, or when you create a container from an image that does not yet exist locally, each layer is pulled down separately, and stored in Docker's local storage area, which is usually `/var/lib/docker/` on Linux hosts. You can see these layers being pulled in this example:

Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state.

See <https://docs.docker.com/storage/storagedriver/> (last accessed March 15, 2024).

IMAGES

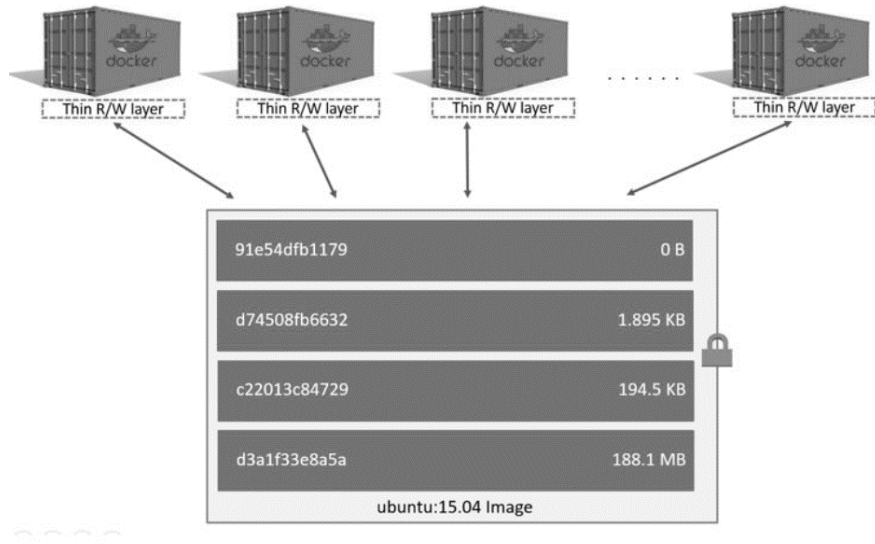
An *image* is a read-only template with instructions for creating a Docker container. Often, an image is *based on* another image, with some additional customization. For example, you may build an image which is based on the `ubuntu` image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.

You might create your own images or you might only use those created by others and published in a registry. To build your own image, you create a *Dockerfile* with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

See <https://docs.docker.com/engine/docker-overview/> (last accessed March 15, 2024).

As a further example, the depiction below shows the thin layer for each container instance to be read and writable, whereas the Docker image (e.g., root image) is read-only.

[REDACTED]



See <https://docs.docker.com/storage/storagedriver/> (last accessed March 15, 2024).

Source Code Evidence

[REDACTED]

[REDACTED]

	[REDACTED] [REDACTED] [REDACTED] [REDACTED]
--	--

[REDACTED]

		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	

[REDACTED]

		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	

[REDACTED]

		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]

[REDACTED]

	[REDACTED]
[7d] wherein said leaf images of respective	Upon information and belief, Liberty Mutual's systems include "wherein said leaf images of respective compute nodes do not include blocks of said root image that are unchanged by respective compute nodes."



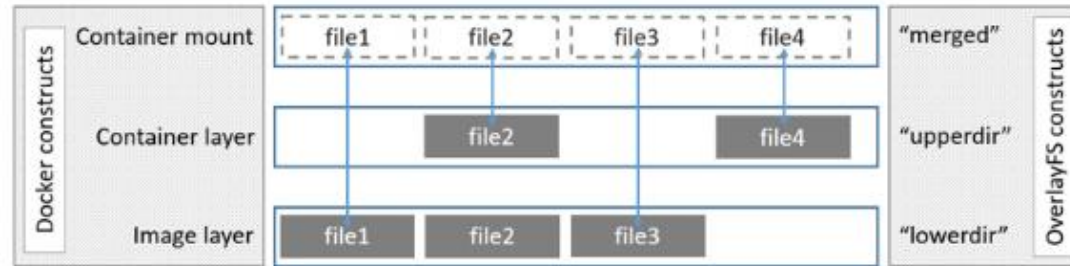
compute nodes do not include blocks of said root image that are unchanged by respective compute nodes; and	For example, when you create a new container, a new read-write layer is added on top of the root image (e.g., leaf images of respective compute nodes). All changes made to the container are written to this layer, where when the container is deleted, the writable layer is also deleted. The underlying image layers will remain unchanged (e.g., do not include blocks of said root image that are unchanged). See, for example, the discussion above with respect to 7[a], 7[b], and 7[c]. Copying makes containers efficient When you start a container, a thin writable container layer is added on top of the other layers. Any changes the container makes to the filesystem are stored here. Any files the container does not change do not get copied to this writable layer. This means that the writable layer is as small as possible. See https://docs.docker.com/storage/storagedriver/ (last accessed on November 10, 2023). As an example, this claim limitation is met as seen in the following descriptions: Container and layers The major difference between a container and an image is the top writable layer. All writes to the container that add new or modify existing data are stored in this writable layer. When the container is deleted, the writable layer is also deleted. The underlying image remains unchanged. See https://docs.docker.com/storage/storagedriver/ (last accessed March 15, 2024); An <i>image</i> is a read-only template with instructions for creating a Docker container. Often, an image is <i>based on</i> another image, with some additional customization. For example, you may build an image which is based on the <code>ubuntu</code> image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run. See https://docs.docker.com/get-started/overview/ (last accessed March 15, 2024);
---	---

Copying makes containers efficient

When you start a container, a thin writable container layer is added on top of the other layers. Any changes the container makes to the filesystem are stored here. Any files the container does not change do not get copied to this writable layer. This means that the writable layer is as small as possible.

See <https://docs.docker.com/storage/storagedriver/> (last accessed March 15, 2024).

See also diagram below showing “...said leaf images...” in the middle row and showing that “...compute nodes do not include blocks of said root image that are unchanged...” in the bottom row.



See <https://docs.docker.com/storage/storagedriver/overlayfs-driver/> (last accessed March 15, 2024).

As a further example, the depiction below shows “...respective compute nodes do not include blocks of said root image that are unchanged by respective compute nodes...”

[REDACTED]

		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]

[REDACTED]

		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	

[REDACTED]

		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	

[REDACTED]

		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	

[REDACTED]

	[REDACTED]
[7e] caching blocks of said root image that have been accessed by at least one of said compute nodes in a cache memory.	Upon information and belief, Liberty Mutual performs “caching blocks of said root image that have been accessed by at least one of said compute nodes in a cache memory.” For example, when a Docker image is being rebuilt, it will check all the instructions from the previous build (e.g., accessed by at least one of said compute nodes). The Docker image reuses layers from previous builds and uses the cached result (e.g., caching blocks of said root image that have been accessed) until it reaches a layer where it has

changed. For example, any changes to a RUN command, COPY, or ADD instruction can invalidate the cache result or that layer, resulting in a rebuild for all the instructions/layers that follow in the Dockerfile.

When you run the `docker build` command to create a new image, Docker executes each instruction in your Dockerfile, creating a layer for each command and in the order specified. For each instruction, Docker checks whether it can reuse the instruction from a previous build. If it finds that you've already executed a similar instruction before, Docker doesn't need to redo it. Instead, it'll use the cached result. This way, your build process becomes faster and more efficient, saving you valuable time and resources.

See <https://docs.docker.com/guides/docker-concepts/building-images/using-the-build-cache/> (last accessed August 5, 2024).

For example, Docker cache stores immediate layers of the image which will be reused in future builds for efficiency. A Docker build cache reuses caches of a previous build on the host to save time.

The `RUN` instruction will execute any commands to create a new layer on top of the current image. The added layer is used in the next step in the Dockerfile. `RUN` has two forms:

See <https://docs.docker.com/reference/dockerfile/#run> (last accessed August 5, 2024).



	Using the build cache effectively lets you achieve faster builds by reusing results from previous builds and skipping unnecessary work. In order to maximize cache usage and avoid resource-intensive and time-consuming rebuilds, it's important to understand how cache invalidation works. Here are a few examples of situations that can cause cache to be invalidated: • Any changes to the command of a <code>RUN</code> instruction invalidates that layer. Docker detects the change and invalidates the build cache if there's any modification to a <code>RUN</code> command in your Dockerfile. • Any changes to files copied into the image with the <code>COPY</code> or <code>ADD</code> instructions. Docker keeps an eye on any alterations to files within your project directory. Whether it's a change in content or properties like permissions, Docker considers these modifications as triggers to invalidate the cache. • Once one layer is invalidated, all following layers are also invalidated. If any previous layer, including the base image or intermediary layers, has been invalidated due to changes, Docker ensures that subsequent layers relying on it are also invalidated. This keeps the build process synchronized and prevents inconsistencies. See https://docs.docker.com/guides/docker-concepts/building-images/using-the-build-cache/ (last accessed August 5, 2024). As a further example, the depiction below provides an efficient way to adjust instructions or commands to utilize the build cache. Layers of the previous image (e.g., root image) that were accessed (e.g., accessed by at least one of said compute nodes in a cache memory) will not need rebuilding.
--	---



Consider the following example. A Dockerfile snippet that runs a JavaScript build from the source files in the current directory:

```
# syntax=docker/dockerfile:1
FROM node
WORKDIR /app
COPY . .           # Copy over all files in the current directory
RUN npm install    # Install dependencies
RUN npm build      # Run build
```

This Dockerfile is rather inefficient. Updating any file causes a reinstall of all dependencies every time you build the Docker image even if the dependencies didn't change since last time!

Instead, the `COPY` command can be split in two. First, copy over the package management files (in this case, `package.json` and `yarn.lock`). Then, install the dependencies. Finally, copy over the project source code, which is subject to frequent change.

```
# syntax=docker/dockerfile:1
FROM node
WORKDIR /app
COPY package.json yarn.lock .    # Copy package management files
RUN npm install                  # Install dependencies
COPY . .                         # Copy over project files
RUN npm build                    # Run build
```

By installing dependencies in earlier layers of the Dockerfile, there is no need to rebuild those layers when a project file has changed.

See <https://docs.docker.com/build/cache/> (last accessed August 6, 2024).

As a further example, the class ImageCollection implementation includes input argument fields that allow for the image to either not use the cache or to include where to cache from during the build (e.g., caching blocks of said root image).



	<pre>class ImageCollection(Collection): model = Image def build(self, **kwargs): """ Build an image and return it. Similar to the ``docker build`` command. Either ``path`` or ``fileobj`` must be set. If you already have a tar file for the Docker build context (including a Dockerfile), pass a readable file-like object to ``fileobj`` and also pass ``custom_context=True``. If the stream is also compressed, set ``encoding`` to the correct value (e.g. ``gzip``). If you want to get the raw output of the build, use the :py:meth:`~docker.api.build.BuildApiMixin.build` method in the low-level API. Args: path (str): Path to the directory containing the Dockerfile fileobj: A file object to use as the Dockerfile. (Or a file-like object) tag (str): A tag to add to the final image quiet (bool): Whether to return the status nocache (bool): Don't use the cache when set to ``True`` rm (bool): Remove intermediate containers. The ``docker build`` command now defaults to ``--rm=true``, but we have kept the old default of ``False`` to preserve backward compatibility timeout (int): HTTP timeout custom_context (bool): Optional if using ``fileobj`` encoding (str): The encoding for a stream. Set to ``gzip`` for compressing pull (bool): Downloads any updates to the FROM image in Dockerfiles forcerm (bool): Always remove intermediate containers, even after unsuccessful builds dockerfile (str): path within the build context to the Dockerfile buildargs (dict): A dictionary of build arguments container_limits (dict): A dictionary of limits applied to each container created by the build process. Valid keys: - memory (int): set memory limit for build - memswap (int): Total memory (memory + swap), -1 to disable swap - cpushares (int): CPU shares (relative weight) - cpusetcpus (str): CPUs in which to allow execution, e.g., ``0-3`` or ``0,1`` shmsize (int): Size of ``/dev/shm`` in bytes. The size must be greater than 0. If omitted the system uses 64MB labels (dict): A dictionary of labels to set on the image cache_from (list): A list of images used for build cache resolution target (str): Name of the build-stage to build in a multi-stage Dockerfile network_mode (str): networking mode for the run commands during build squash (bool): Squash the resulting images layers into a single layer. extra_hosts (dict): Extra hosts to add to /etc/hosts in building containers, as a mapping of hostname to IP address. platform (str): Platform in the format ``os[/arch[/variant]]``. isolation (str): Isolation technology used during build. Default: 'None'. use_config_proxy (bool): If ``True``, and if the docker client configuration file (``~/.docker/config.json`` by default) contains a proxy configuration, the corresponding environment variables will be set in the container being built. Returns: (tuple): The first item is the :py:class:`Image` object for the image that was built. The second item is a generator of the build logs as JSON-decoded objects.</pre>
--	---

[REDACTED]

See <https://github.com/docker/docker-py/blob/a3652028b1ead708bd9191efb286f909ba6c2a49/docker/models/images.py#L13> (last accessed August 3, 2024).

Source Code Evidence

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	

[REDACTED]

		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	

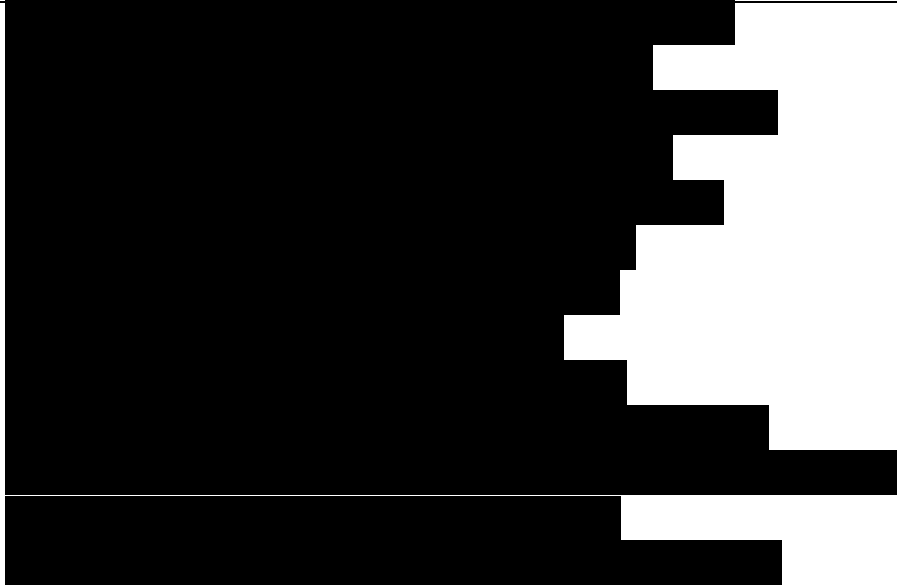
[REDACTED]

		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	
		[REDACTED]	

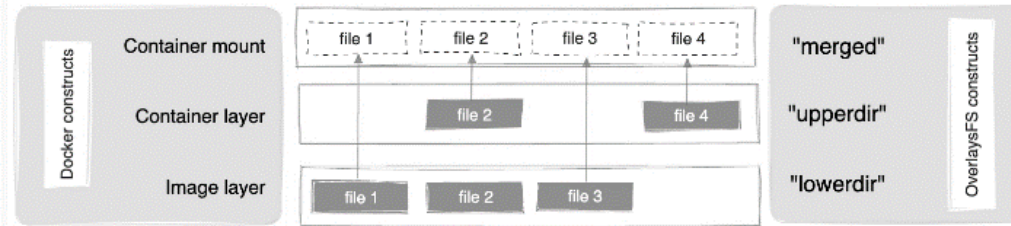
[REDACTED]

		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]
		[REDACTED]

[REDACTED]

	
Claim 11	Liberty Mutual Count I Systems and Services
The method as recited in claim 7 further comprising: merging the blocks of said root image with the blocks of respective leaf images to create cohesive respective application environments.	Upon information and belief, Liberty Mutual’s systems perform the “method as recited in claim 7 further comprising: merging the blocks of said root image with the blocks of respective leaf images to create cohesive respective application environments.” <i>See Claim 7, supra.</i> For example, on information and belief as shown below, Docker’s use of the <i>overlay</i> feature and/or <i>merge</i> feature and/or <i>add a new layer</i> feature facilitates a new container. It will combine the read-only image layer plus a new level layer for the said container that is launched. <u>Image and container layers on-disk</u>

The following diagram shows how a Docker image and a Docker container are layered. The image layer is the `lowerdir` and the container layer is the `upperdir`. If the image has multiple layers, multiple `lowerdir` directories are used. The unified view is exposed through a directory called `merged` which is effectively the containers mount point.



Where the image layer and the container layer contain the same files, the container layer (`upperdir`) takes precedence and obscures the existence of the same files in the image layer.

To create a container, the `overlay2` driver combines the directory representing the image's top layer plus a new directory for the container. The image's layers are the `lowerdirs` in the overlay and are read-only. The new directory for the container is the `upperdir` and is writable.

See <https://docs.docker.com/storage/storagedriver/overlayfs-driver/> (last accessed March 15, 2024).

The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer".

See <https://docs.docker.com/storage/storagedriver/#images-and-layers> (last accessed March 15, 2024).

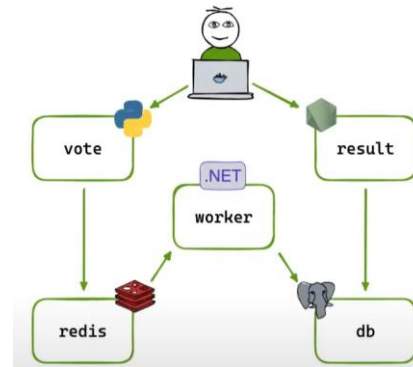
Note that explained above the "blocks..." can, for example, be a container.



	A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that aren't stored in persistent storage disappear. Containers are lightweight and contain everything needed to run the application, so you don't need to rely on what's installed on the host. See https://docs.docker.com/get-started/overview/ (last accessed March 15, 2024). As an example (on information and belief), several Docker containers (e.g., blocks of respective leaf images) can be isolated processes for each of an application's components. Each component can lie on respective containers and combining them will create a functional application (e.g., create cohesive respective application environments). What is a container? Simply put, containers are isolated processes for each of your app's components. Each component - the frontend React app, the Python API engine, and the database - runs in its own isolated environment, completely isolated from everything else on your machine. Here's what makes them awesome. Containers are: • Self-contained. Each container has everything it needs to function with no reliance on any pre-installed dependencies on the host machine. • Isolated. Since containers are run in isolation, they have minimal influence on the host and other containers, increasing the security of your applications. • Independent. Each container is independently managed. Deleting one container won't affect any others. • Portable. Containers can run anywhere! The container that runs on your development machine will work the same way in a data center or anywhere in the cloud!
--	---

See <https://docs.docker.com/guides/docker-concepts/the-basics/what-is-a-container/> (last accessed August 7, 2024).

As a further example, the components “vote”, “redis”, “worker”, “db”, and “result” below all have their own root images and/or configurations. After being put together in the Docker compose file, they will formulate a responsive application.



See <https://www.youtube.com/watch?v=W1kWqFkiu7k> (last accessed August 7, 2024). Timestamp: 2:36.

Source Code Evidence



[REDACTED]

	[REDACTED] [REDACTED]
--	--