

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

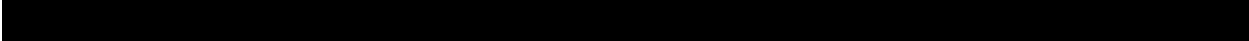
**Amended Infringement Claim Chart for
U.S. Patent No. 8,332,844 (“the ’844 Patent”)**

The Accused Systems and Services include, without limitation, Comerica’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 Comerica’s systems and services that have the same or substantially similar features as the specifically identified systems and services (“Comerica Count I Systems and Services”).¹ Upon information and belief, Comerica’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, Comerica uses Docker for various applications. In addition, Comerica directly infringes the ’844 Patent via testing the various systems and offerings. On information and belief, Comerica also indirectly infringes the ’844 Patent through inducement. On information and belief, Comerica also contributes to the direct infringement through its vendors and other third parties’ uses of Comerica’s Systems and Services that utilize Docker by providing components relating to Docker that have no substantial non-infringing uses. On information and belief, Comerica 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, Comerica servers, or servers that are run on behalf of Comerica, are involved and/or perform the method of the asserted claims of the ’844 Patent. On information and belief, Comerica 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.

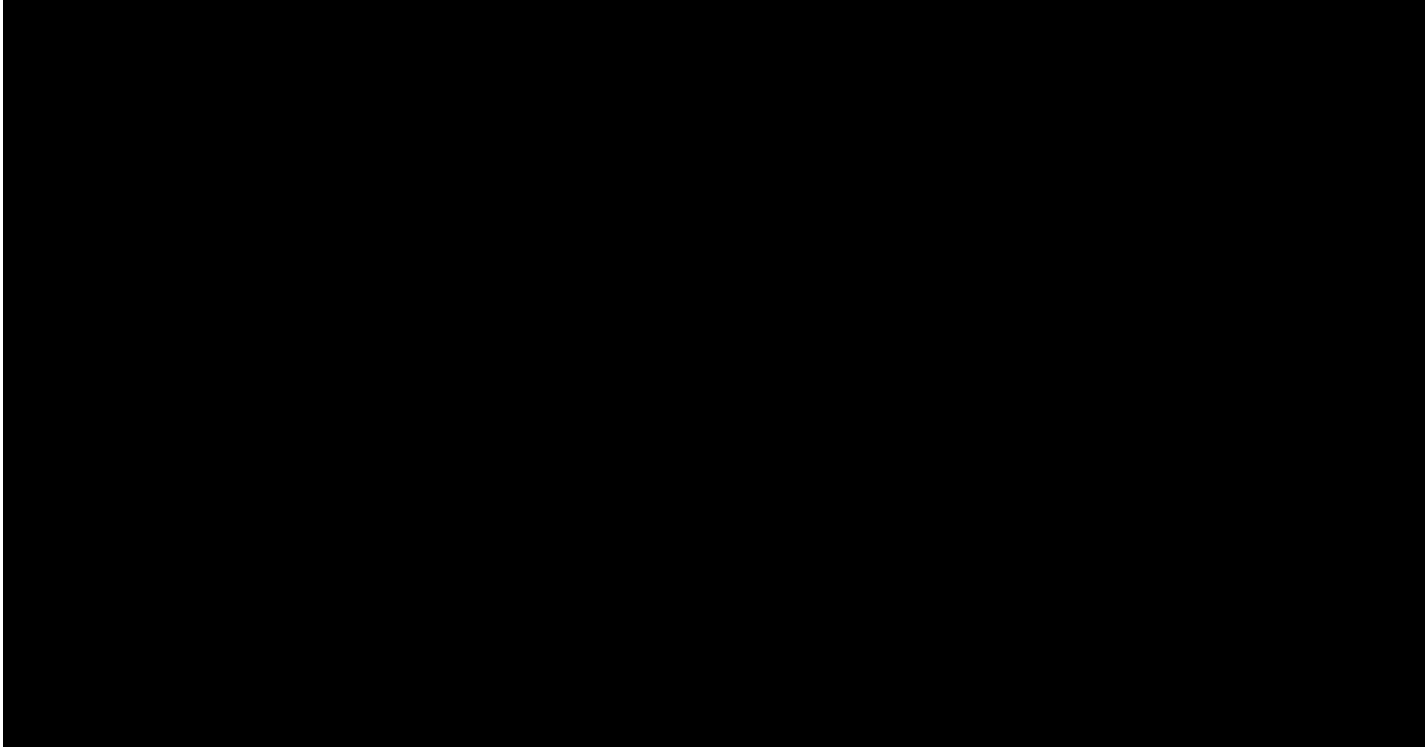
Pursuant to P.R. 3-1(b), IV identifies Docker as used in Comerica’s systems and offerings as the Accused Instrumentality. Information regarding how specifically this Docker is used for what product and offerings is non-public. IV intends to obtain discovery regarding how the Docker is used in Comerica 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 Comerica on June 26, 2024, seeking information relating to Comerica’s use and implementation of Docker, including use and implementation of Docker to support services that are offered to Comerica’s

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

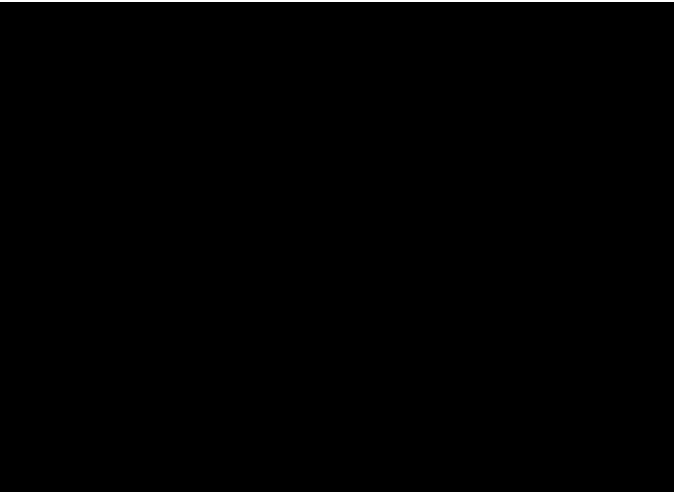
IV accuses Comerica private clouds that implement Docker and non-licensed public clouds that Comerica uses to support Docker for its systems and services. To the extent any of these licenses are relevant to Defendant’s activities, Plaintiffs will meet and confer with Defendants about the impact of such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

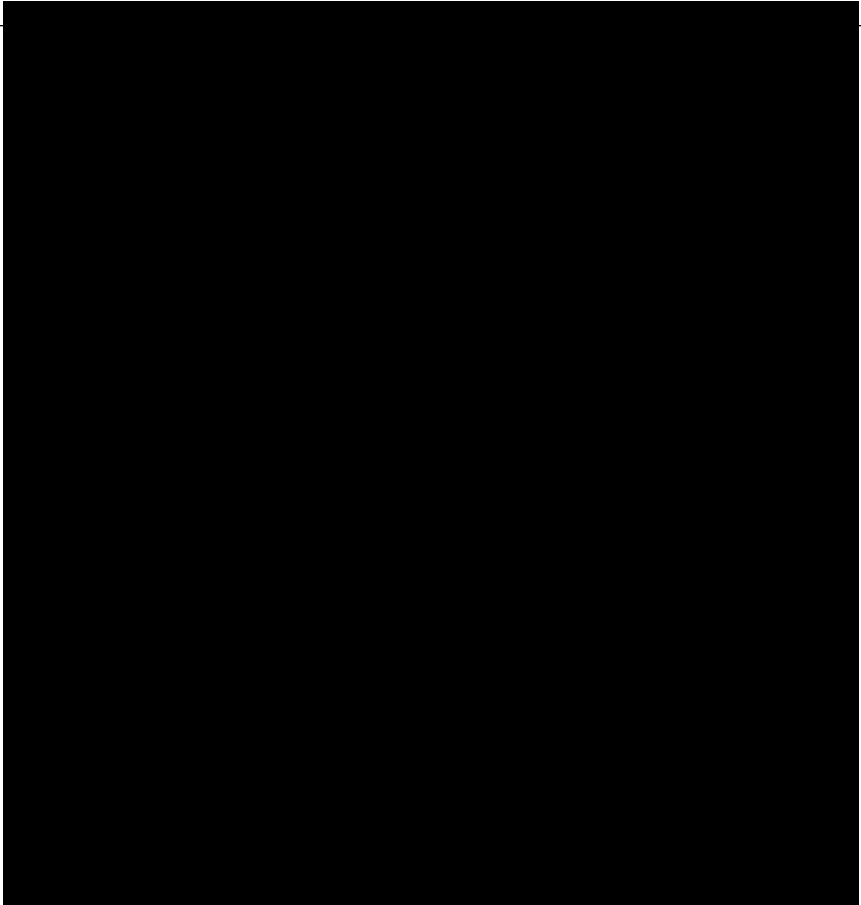


customers and/or employees. This information requested by IV includes for example technical documents that show the operation of Comerica's use and implementation of Docker, Comerica source code relating to Docker, whether third parties have information relating to Comerica's use and implementation of Docker, and other technical information. *See* Plaintiffs' Initial 30(b)(6) Notice to Defendant (Comerica) 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 Defendant (Comerica). IV has also served 30(b)(1) deposition notices for several technical Comerica employees that likely possess knowledge of Comerica's use and implementation of Docker. To date, Comerica has not offered any deposition dates for either potential corporate witnesses or witnesses having personal knowledge of Comerica'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 Comerica uses and implements Docker in its systems and services. Further, fact discovery remains open and Plaintiffs intend to serve additional discovery to better understand Comerica'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 Comerica 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	Comerica Count I Systems and Services
7[pre]. A method for providing data to a plurality of compute nodes, comprising:	Upon information and belief, Comerica’s systems perform “a method for providing data to a plurality of compute nodes.” Comerica deploys Docker in multiple applications. Docker provides docker images (e.g., providing data) to many containers (e.g., a plurality of compute nodes) that run an instance of the image. Upon information and belief, Comerica’s systems leverage Docker in the following applications (e.g., accused products). Below is a list of applications that leverage Docker in association with different software.  See COM000000131.

For example, upon information and belief, Comerica deploys the BigID Collibra integration app using Docker Compose. Requiring an OS, Docker Engine, and Docker Compose, a Docker image is first loaded from the installation kit and then the application runs on a Docker Container.





See COM000000132.

For example (on information and belief), Comerica’s Wire Fraud Detection technical architecture design within the Fraud Detection and Prevention application. This includes the implementation of Docker within the network and process flows. Docker EE and/or Docker Swarm clusters are utilized in the architecture and flow diagram from other clusters, operations, servers, and/or services.

[REDACTED]

[REDACTED]

See COM000000159.

[REDACTED]

See COM000000039.

[REDACTED]

	Upon information and belief, Comerica's system comprises the Selenium Grid Server which pools web browsers launched from Docker containers. [REDACTED] <i>See</i> COM000000233. For example (on information and belief), Comerica's use of Docker includes using Docker images, which includes providing all the file systems, binaries, libraries, and configuration (e.g., providing data), to relevant containers (e.g., plurality of compute nodes) that run an instance of that Docker image. The Docker platform, architecture, and other aspects are described and defined below within the Wire Fraud Detection technical architecture design.
--	--

[REDACTED]

[REDACTED]

See COM000000148, COM000000149.

See COM000000173.

As an example (on information and belief), Comerica's use of Docker includes providing docker images to relevant containers that run an instance of the image. As described and defined below, Docker uses a client-server architecture to facilitate building, running, and distributing Docker containers.

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

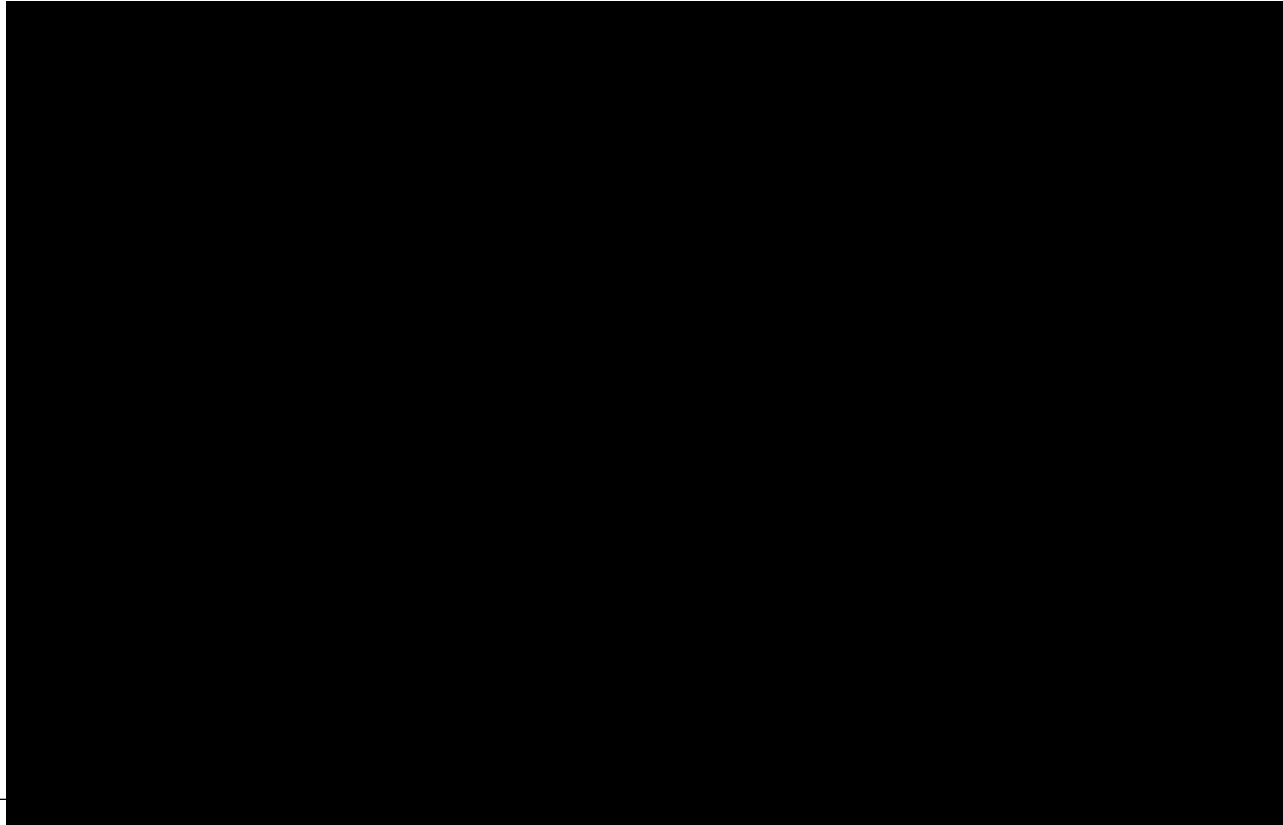
A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

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

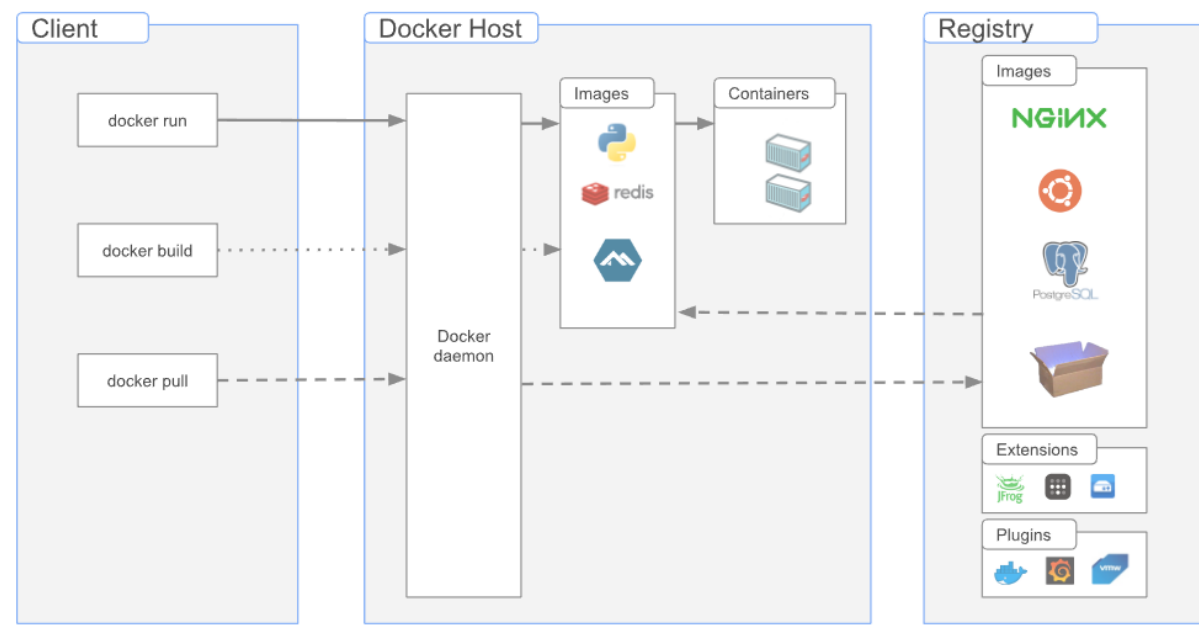
A running container uses an isolated filesystem. This isolated filesystem is provided by an image, and the image must contain everything needed to run an application - all dependencies, configurations, scripts, binaries, etc. The image also contains other configurations for the container, such as environment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

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

As an example, the Docker architecture involves a “Docker client” communicating with a server/host. At each host, there is a Docker daemon responsible for managing Docker containers, images from a registry, and any network connections.



See COM000000175.



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

As an example of Docker architecture, Docker SDK for Python includes functionalities that the docker command does within Python applications to run and manage containers. For example, the class `DockerClient` is a client for communicating with a Docker server. The `DockerClient` is responsible for making the API requests to the server, where the server is responsible for providing and managing images (e.g., providing data) for any active containers (e.g., a plurality of compute nodes).



```
class DockerClient:
    """
    A client for communicating with a Docker server.

    Example:

    >>> import docker
    >>> client = docker.DockerClient(base_url='unix:///var/run/docker.sock')

    Args:
    base_url (str): URL to the Docker server. For example,
        ``unix:///var/run/docker.sock`` or ``tcp://127.0.0.1:1234``.
    version (str): The version of the API to use. Set to ``auto`` to
        automatically detect the server's version. Default: ``1.35``
    timeout (int): Default timeout for API calls, in seconds.
    tls (bool or :py:class:`~docker.tls.TLSConfig`): Enable TLS. Pass
        ``True`` to enable it with default options, or pass a
        :py:class:`~docker.tls.TLSConfig` object to use custom
        configuration.
    user_agent (str): Set a custom user agent for requests to the server.
    credstore_env (dict): Override environment variables when calling the
        credential store process.
    use_ssh_client (bool): If set to ``True``, an ssh connection is made
        via shelling out to the ssh client. Ensure the ssh client is
        installed and configured on the host.
    max_pool_size (int): The maximum number of connections
        to save in the pool.
    """
```

See <https://github.com/docker/docker-py/blob/a3652028b1ead708bd9191efb286f909ba6c2a49/docker/client.py#L16> (last accessed August 3, 2024).

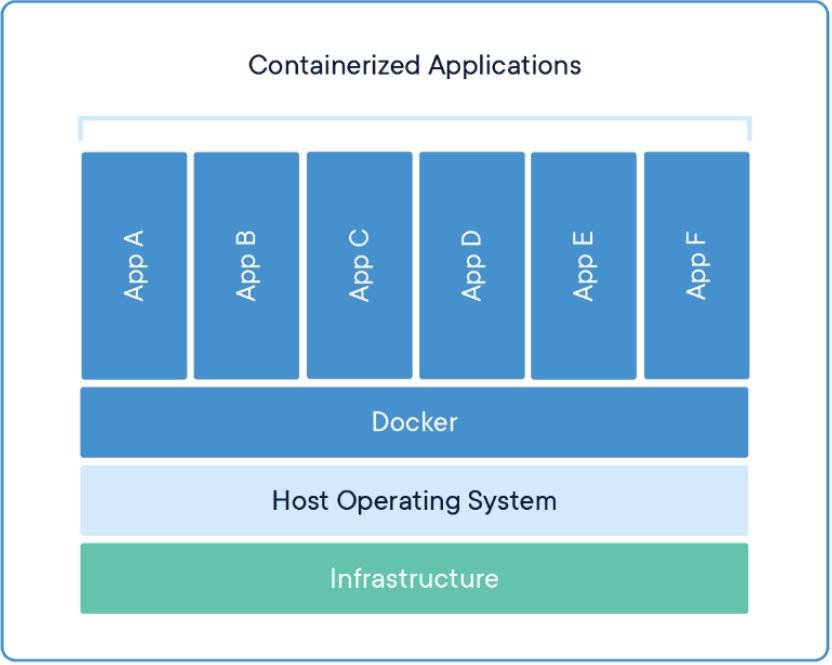
Furthermore, the implementation of a daemon on a server refers to the class CancellableStream. A DockerClient makes API requests to the server, and the daemon/class CancellableStream is a wrapper for these requests, and any other events or logs at the server.

```
class CancellableStream:
    """
    Stream wrapper for real-time events, logs, etc. from the server.

    Example:
    >>> events = client.events()
    >>> for event in events:
    ...     print(event)
    >>> # and cancel from another thread
    >>> events.close()
    """
```

See <https://github.com/docker/docker-py/blob/a3652028b1ead708bd9191efb286f909ba6c2a49/docker/types/daemon.py#L8> (last accessed August 3, 2024).

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).

See <https://www.linkedin.com/in/michaelmxu/> (last accessed on November 13, 2023) (noting the use of Docker at Comerica).

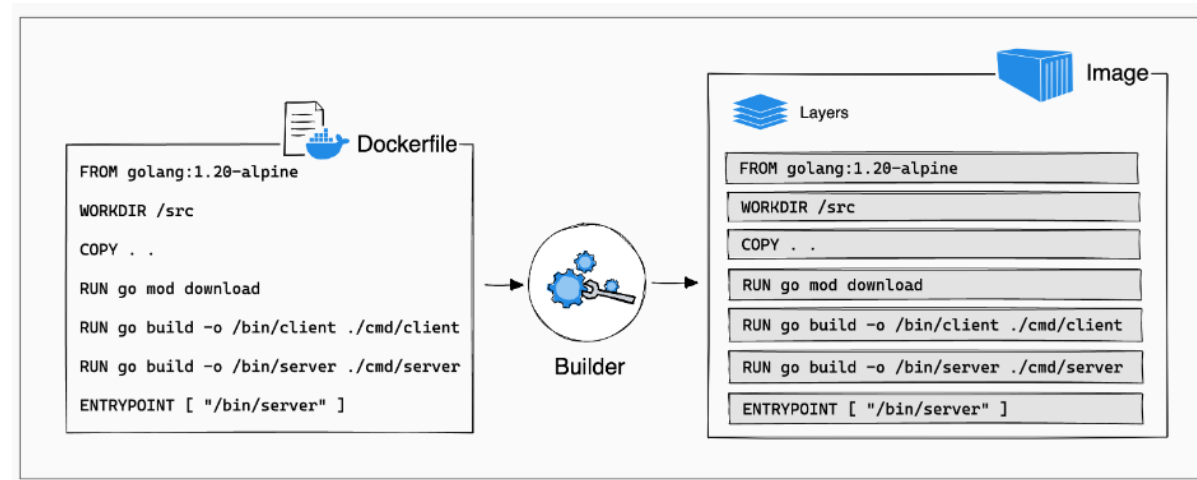
Source Code Evidence

[REDACTED]



	The folder [REDACTED] includes installing Docker packages, setting up and launching EC2 instances. Logical volumes are also added and formatted for Selenium as well as creating a mountpoint or mount. The folder [REDACTED] includes several Docker functionalities. • Line 1, logging into Docker • Line 43, building mapr-jruby Docker image • Line 47, pushing mapr-jruby Docker image to Artifactory As a further example, the following folders (e.g., the accused applications/services) also contain the configuration or implementation files for Docker. [REDACTED]
7[a] storing blocks of a root image of said compute nodes on a first storage unit;	Upon information and belief, Comerica's systems perform "storing blocks of a root image of said compute nodes on a first storage unit."

For example, Docker images are made of layers where each instruction of the Dockerfile is a layer (e.g., blocks of a root image). Images are stored locally on your computer or on an image registry, where Docker Hub is the default registry (e.g., storing on a first storage unit).

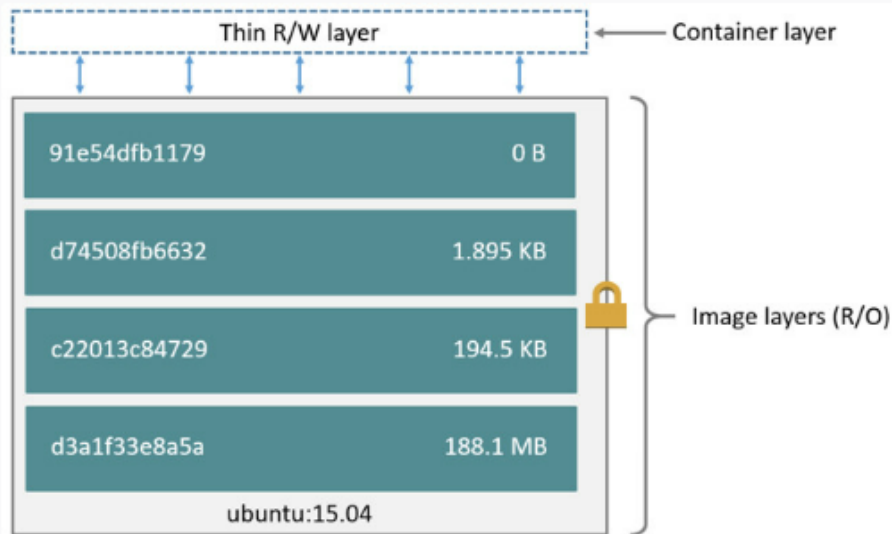


See <https://docs.docker.com/build/guide/layers/> (last accessed August 5, 2024).

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 9, 2023).

For example, Docker images are either stored locally on a computer or in Docker Hub, the configured default registry to look for images. A registry of these images can have several versions, for example, different release versions 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://docs.docker.com/registry/ (last accessed on November 9, 2023). A Docker <i>registry</i> 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 7pre, *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 said image and 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, nopruner=False): """ Remove this image. Args: force (bool): Force removal of the image nopruner (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, nopruner=nopruner,)</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. 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).

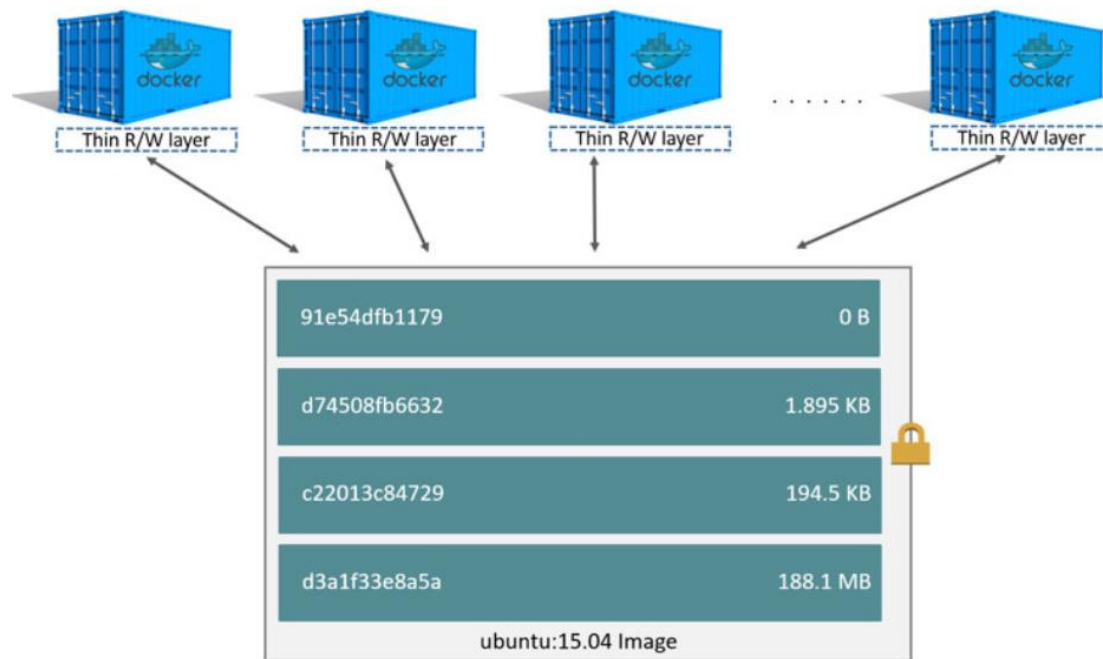


	<pre>class ImageCollection(Collection): model = Image def build(self, **kwargs): """ Build an image and return it. Similar to the ``docker build`` command. Either ``path`` or ``fileobj`` must be set. If you already have a tar file for the Docker build context (including a Dockerfile), pass a readable 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), depicted as blue containers, each have 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 is represented by the cyan blocks below the blue containers.

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



Id.

See COM000000176.

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 from. This class includes the functionalities for managing multiple containers and their images (e.g., root image of said compute nodes).

```
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).

```

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. <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.

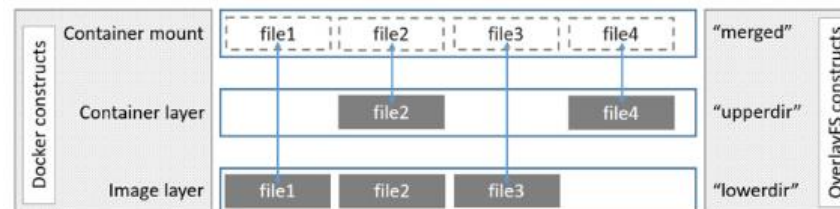
This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:18.04` image. The `LABEL` command only modifies the image's metadata, and does not produce a new layer. The `COPY` command adds some files from 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.

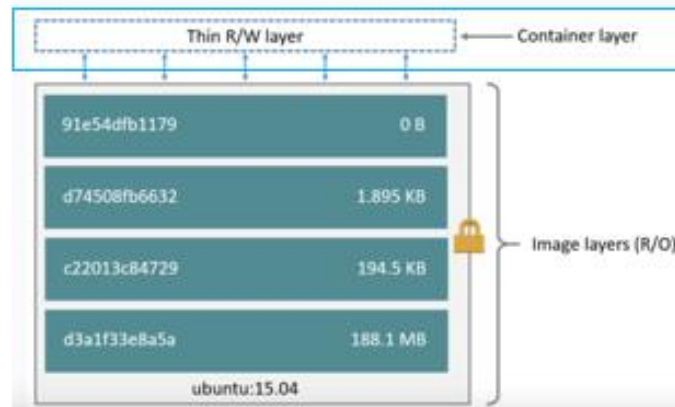
[REDACTED]

	[REDACTED]
7[b] storing leaf images for respective compute nodes on respective second storage units,	Upon information and belief, Comerica’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 on the local Docker storage area (e.g. respective second storage units). For example, (on information and belief) the development or production environments of Comerica’s applications and software would be hosted on physical machines or servers (e.g., second storage units), including with “OnPrem” locations. [REDACTED] See COM000000131.

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:

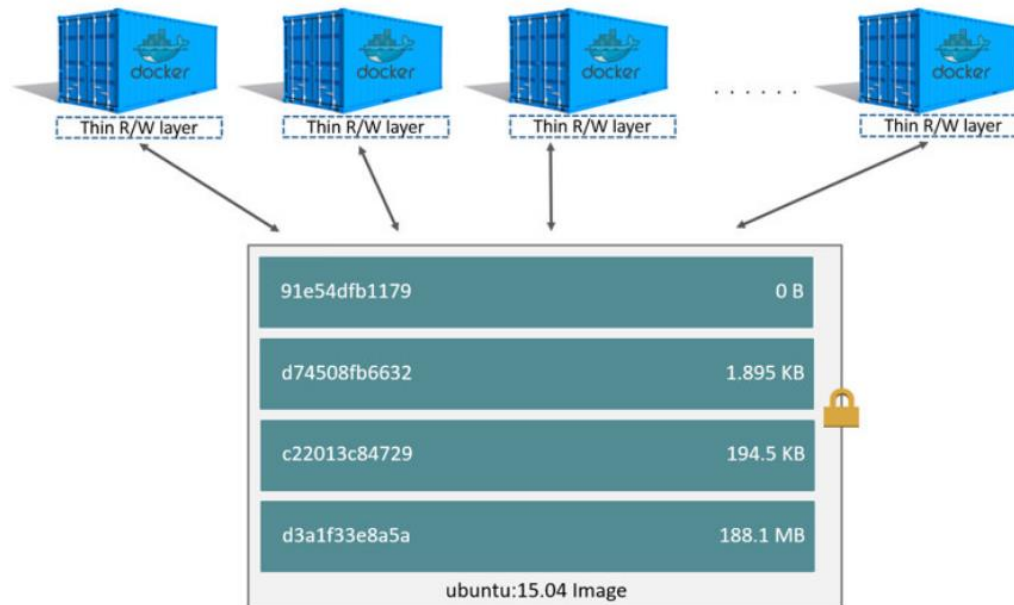
The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, 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 9, 2023).

In another example, Docker images on runnable containers are made up of layers. Container images (e.g., leaf images), represented by the blue containers below, can store any new data or modifications of the root image at the host container (e.g., storing on respective second storage units).

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



Id.

See COM000000176.

See <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. <i>Id.</i> 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> Another example, upon information and belief, any changes made to the data when a new container instance is created from an image, are performed in the writable container layer (e.g., leaf images for respective compute nodes on respective second 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.

Id.

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]
7[c] said leaf images including only additional data blocks not previously contained in said root image and	Upon information and belief, Comerica's systems also have "said 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 container 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

changes made by respective compute nodes to the blocks of the root image,	modifications are 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 November 9, 2023). 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 9, 2023). For example, any changes the container makes to the file systems are stored on the thin writable container layer (e.g., leaf images including only additional data blocks not previously contained in said
--	---



root image), while the base image is a read-only template (e.g., changes made by respective compute nodes to the blocks of the root image).

Copying makes containers efficient

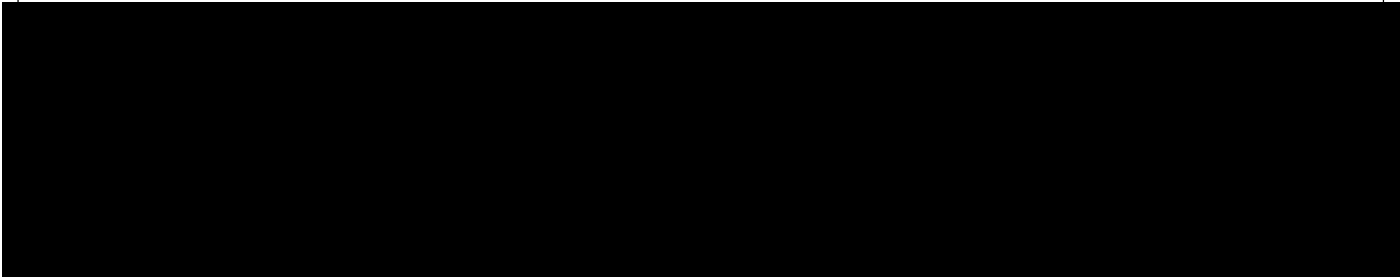
When you start a container, a thin writable container layer is added on top of the other layers. Any changes the container makes to the filesystem are stored here. Any files the container does not change do not get copied to this writable layer. This means that the writable layer is as small as possible.

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).



See COM000000176.

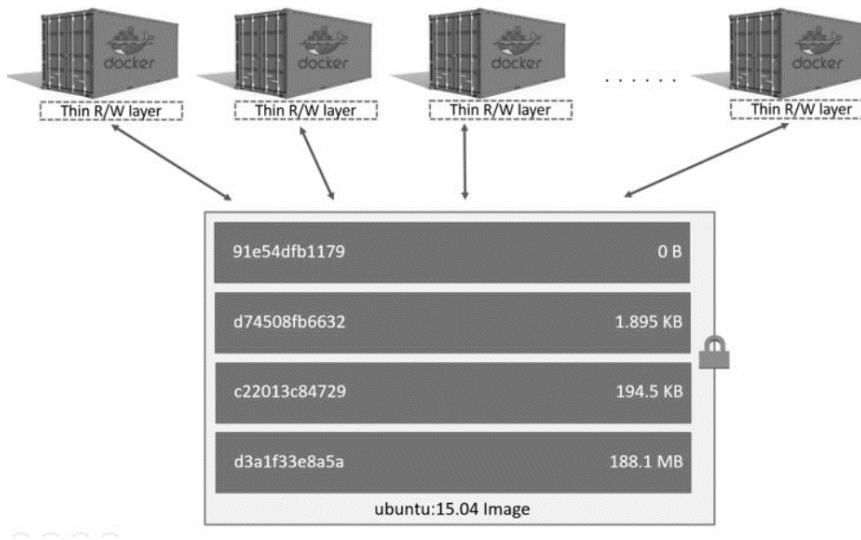
IMAGES

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).

Another example, upon information belief, a thin layer for each container instance is read and writable, whereas the Docker image (e.g., root image) is read-only.



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

Source Code Evidence

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

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

For example, Docker cache stores intermediate layers of the image, which don't change frequently. See below.

Understanding Docker Cache

Before we dive into the cleaning process, it's important to understand how Docker cache works. When you build a Docker image, Docker uses a build cache to speed up the build process. The build cache stores intermediate layers of the image, which are the layers that don't change frequently. This allows Docker to reuse these layers when building a new image, saving time and resources.

See <https://collabnix.com/how-to-clear-docker-cache/> (last accessed on November 9, 2023).

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). Another example shown below provides a way to write instructions or commands to utilize the build cache, where 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) do 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 when it is built (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>
--	--

See <https://github.com/docker/docker-py/blob/a3652028b1ead708bd9191efb286f909ba6c2a49/docker/models/images.py#L13> (last accessed August 3, 2024).
A docker cache is important because reusing the cache of a previous build on the host can save time. See below.

At each occurrence of a RUN command in the Dockerfile, Docker will create and commit a new layer to the image, which is just a set of tightly-coupled directories full of various file structure that comprise a Docker image. In a default install, these are located in /var/lib/docker.

During a new build, all of these file structures have to be created and written to disk — this is where Docker stores base images. Once created, the container (and subsequent new ones) will be stored in the folder in this same area.

What makes the cache important? If the objects on the file system that Docker is about to produce are unchanged between builds, reusing a cache of a previous build on the host is a great time-saver. It makes building a new container really, really fast. None of those file structures have to be created and written to disk this time — the reference to them is sufficient to locate and reuse the previously built structures.

See <https://thenewstack.io/understanding-the-docker-cache-for-faster-builds/> (last accessed on November 9, 2023).

Source Code Evidence

[REDACTED]

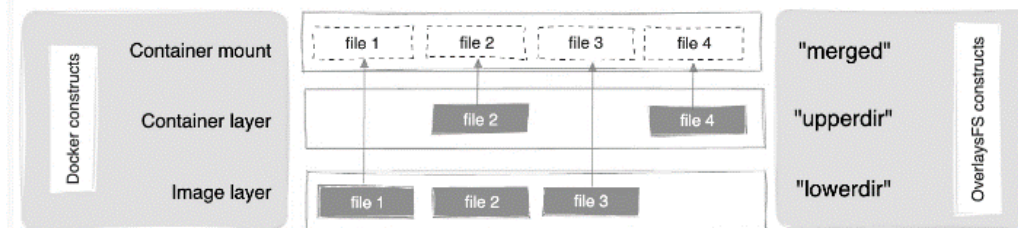
merging the blocks of said root image with the blocks of respective leaf images to create cohesive respective application environments.

See Claim 7, *supra*.

For example, upon information and belief, as shown below, Docker includes the overlay feature, the merge feature, and an add a new layer feature. (e.g., merging the blocks of said root image with the blocks of respective leaf images to create cohesive respective application environments).

Image and container layers on-disk

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).

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 app's components. Each component can lie on respective containers and combining them creates a functional application (e.g., create cohesive respective application environments).

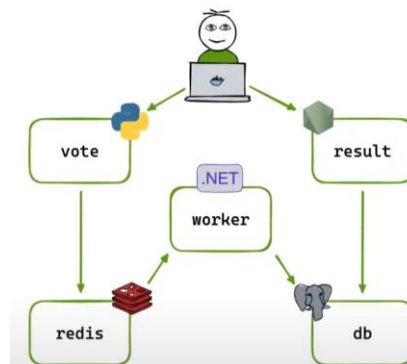
What is a container? Simply put, containers are isolated processes for each of your app's components. Each component - the frontend React app, the Python API engine, and the database - runs in its own isolated environment, completely isolated from everything else on your machine.

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 another example, the components “vote,” “redis,” “worker,” “db,” and “result” shown below all have their root images and/or configurations put together in the Docker compose file to 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]
--	-----------------------