🚨 AtomicJar is now part of Docker 🐋 ! Read the blog

Testcontainers®

Menu

Testcontainers Cloud

Log In

Free Trial

Docs for Testcontainers Cloud

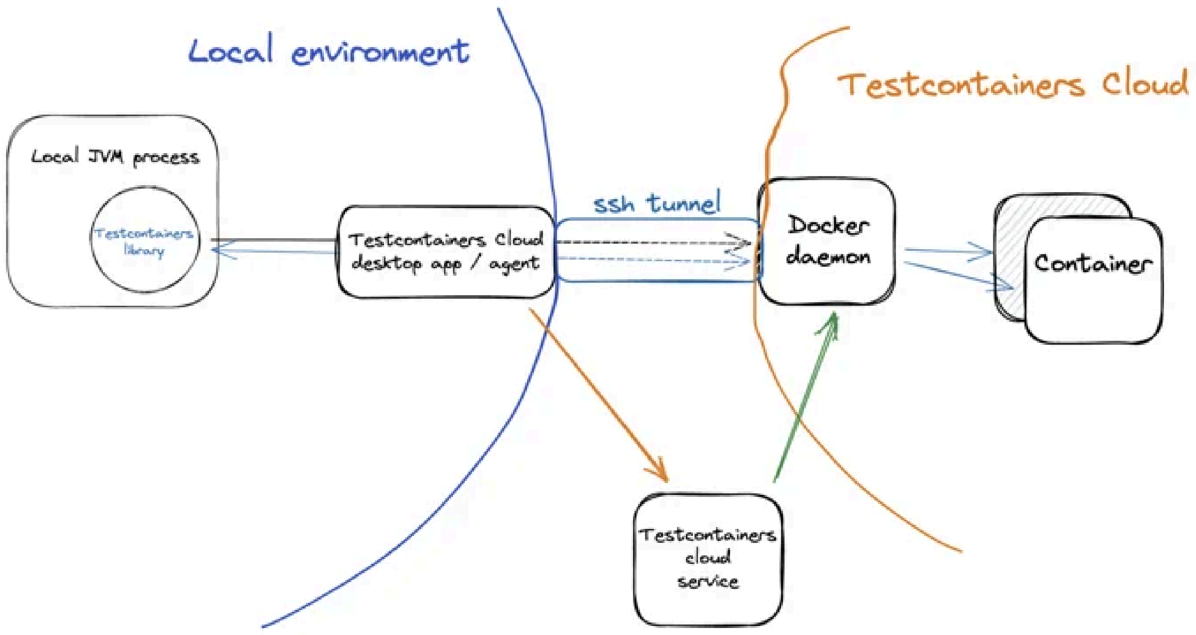
Overview

Running your Testcontainers tests with Testcontainers Cloud works out of the box. It gives you access to the same Testcontainers modules to test applications using popular technologies, without spinning heavy containers on your local machine.

How does Testcontainers Cloud work?

The Testcontainers Cloud agent opens an SSH tunnel and connects to the Docker daemon in the cloud environment

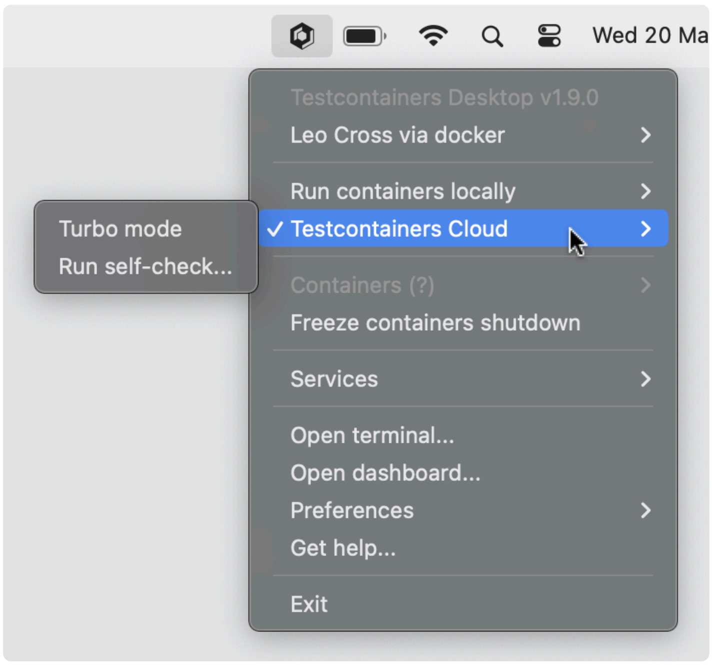
Testcontainers Cloud removes the need for running containers locally. When you trigger your tests in a local environment Testcontainers Cloud agent opens an SSH tunnel and connects to the Docker daemon in the cloud environment. It will pull a Docker image and start the container defined in your Testcontainers-based test, both in the connected cloud environment. Connection to the cloud environment will be active while your tests are running. When your test suite completes, Testcontainers Cloud will wait for a short period of time and then close the connection and remove allocated resources automatically.



Getting Started

Installing the client

1. Download and install [Testcontainers Desktop](https://testcontainers.com/desktop/)
2. Select `Run with Testcontainers Cloud`



You can now run your Testcontainers based tests with containers running on Testcontainers Cloud.

If you don't have existing tests that use Testcontainers, you can use one of the examples projects that contains a few tests that verify that Testcontainers Cloud is configured correctly on your machine.

Example Projects

.NET

SHELL

```
git clone https://github.com/AtomicJar/testcontainers-cloud-dotnet-example
cd testcontainers-cloud-dotnet-example
dotnet test --logger:"console;verbosity=detailed"
```

Go

SHELL

```
git clone https://github.com/AtomicJar/testcontainers-cloud-go-example
cd testcontainers-cloud-go-example
go mod download
go test -v -count=1
```

Java

SHELL

```
git clone https://github.com/AtomicJar/testcontainers-cloud-java-example
cd testcontainers-cloud-java-example
./mvnw test
```

Node.js

SHELL

```
git clone https://github.com/AtomicJar/testcontainers-cloud-nodejs-example
cd testcontainers-cloud-nodejs-example
```

```
npm install
npm test
```

Running existing tests

Running your existing Testcontainers based integration tests works out of the box with Testcontainers Cloud. To get a better feel for the experience of using Testcontainers Cloud, we recommend running your test two times in quick succession, so that you can get a feeling for working with a warmed-up environment.

Testcontainers Cloud also works when you run your tests from your IDE, so often it is useful to get started by running a few tests from the IDE to get an overview of how your project's tests run on Testcontainers Cloud. Use normal IDE functionality to run the tests and check the output logs for Testcontainers Cloud being the used Docker implementation.

If you want to switch back to local Docker just stop Testcontainers Cloud Client app and your Testcontainers based integration tests will run using local Docker.

How the correct container runtime is chosen

Sometimes a system has access to both a Docker daemon and Testcontainers Cloud. To resolve which Docker environment to use Testcontainers libraries do the following in order:

1. Read the `~/.testcontainers.properties` file (if it exists):
    - Find the Docker daemon location from its `tc.host` property
2. Obtain Docker daemon location from the environment variable `DOCKER_HOST`
3. Fall back to trying the default Docker locations for the current operating system

Testcontainers Cloud Agent, both Desktop and CI, configures its own location in the `~/.testcontainers.properties` file, which makes the tests automatically prefer it.

You can also configure individual projects not to consider the configuration in the `~/.testcontainers.properties` file.

Disable for a specific project

When started Testcontainers Cloud configures the local environment to use it for Testcontainers tests by default. Sometimes you might want to opt-out from using it in a specific project, for example because it uses older Testcontainers libraries not yet compatible with Testcontainers Cloud or code patterns that are not ideal for a cloud-based container execution.

You can configure a particular project not to use the global Testcontainers Cloud configuration by updating the dockerconfig.source property in the `testcontainers.properties` configuration file within the project (on your classpath).

This option is only available in Testcontainers for Java.

Add the `testcontainers.properties` configuration file to the classpath of your project with the following content:

```
dockerconfig.source=autoIgnoringUserProperties
```

You can keep the Testcontainers Cloud client app running and no more changes are needed, but this project will now use the usual automatic Testcontainers environment discovery mechanism to find a suitable way to communicate with Docker.

Parallelize your tests with Turbo mode

Testcontainers Cloud Turbo mode allows you to run tests in parallel so that each test process receives its own cloud environment making tests parallelization scalable. Parallelizing tests is one of the ways to speed up the execution of your build.

Depending on the composition of a test suite, and compute resources available parallel tests might not improve performance due to bottlenecking on local compute resources. Testcontainers Cloud Turbo mode enables one cloud environment per process running Testcontainers tests.

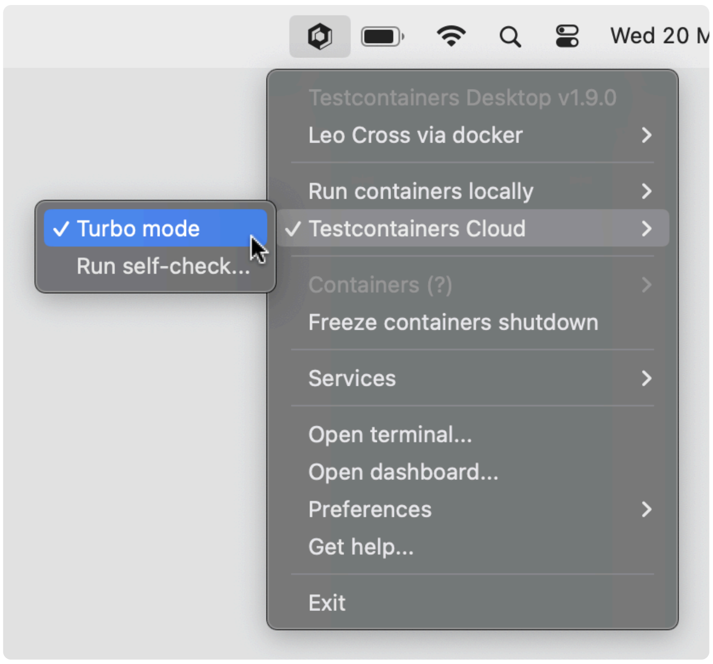
This means that increasing the number of processes simultaneously running tests doesn't increase the load on local compute resources linearly and the scalability of the cloud environment helps to run you tests faster.

Turbo mode will benefit users who struggle with massive test suites which run longer than is acceptable. Here's how you can enable Testcontainers Cloud Turbo mode and configure your tests to run in parallel.

Note that Turbo mode is currently restricted for free accounts.

Start using Turbo Mode with Testcontainers Cloud on Desktop

In the Testcontainers Desktop application you can select the checkbox Turbo mode to enable the mode locally.



Turn on Turbo mode in CI:

When starting the agent in CI specify the `--max-concurrency=N` flag to enable a maximum of N concurrent Testcontainers Cloud environments available to processes using this agent.

The default value for `--max-concurrency` is 4.

You can also configure the concurrency option via an environment variable `TC_CLOUD_CONCURRENCY`, for example setting:

SHELL

```shell
export TC_CLOUD_CONCURRENCY=2
```

Now you're ready to run tests in parallel using scalability of Testcontainers Cloud. If you're new to running tests in parallel here's a quick instruction on how to try it with common build tools.

How to turn on Turbo mode with a Java project using Gradle

To run tests in parallel with Gradle add maxParallelForks to the test task in your gradle.build file:

```
test.maxParallelForks = 4
```

This will instruct Gradle to use up to 4 forked processes to run tests and if you configured Turbo mode, containers created in each fork will not overload the same Docker environment.

If you would like to try Turbo mode on a Java project, consider using the sample project by following the instructions in its GitHub repo.

How to turn on Turbo mode with a Java project using Maven

If you're using Maven as your build tool, you're probably using Surefire plugin for test execution. Surefire plugin supports parallel test execution and you can read the docs on how to enable it. In general to run tests in parallel with Maven you can add the parallel and forkCount properties to the configuration of Maven Surefire or Failsafe plugins in your pom.xml. Note that you need a modern version of the Surefire plugin, here's a sample configuration for it:

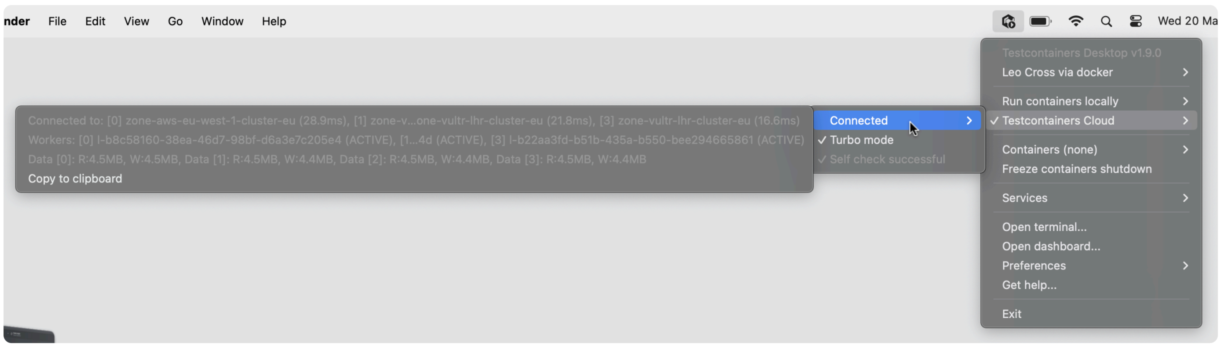
JAVA

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>3.0.0-M7</version>
  <configuration>
    <parallel>classes</parallel>
    <forkCount>4</forkCount>
  </configuration>
</plugin>
```

If you would like to try Turbo mode on a Java project, consider using the sample project by following the instructions in its GitHub repo.

How to verify that Turbo mode is enabled?

When you enabled Turbo mode for Testcontainers Cloud and parallel tests in your build tool, you should see multiple lease allocations in the connection tab of the Testcontainers Cloud application once you run your tests.



```
Connected to: [0] zone-aws-eu-west-1-cluster-eu (28.9ms), [1] zone-v...one-vultr-lhr-cluster-eu (21.8ms), [3] zone-vultr-lhr-cluster-eu (16.6ms)
Workers: [0] l-b8c58160-38ea-46d7-98bf-d6a3e7c205e4 (ACTIVE), [1...4d (ACTIVE), [3] l-b22aa3fd-b51b-435a-b550-bee294665861 (ACTIVE)
Data [0]: R:4.5MB, W:4.5MB, Data [1]: R:4.5MB, W:4.4MB, Data [2]: R:4.5MB, W:4.4MB, Data [3]: R:4.5MB, W:4.4MB
```

How does Testcontainers Cloud decide which containers to run in the same cloud environment?

We use sessions to distinguish containers, so the same process will connecting to the same cloud environment for example if a test needs to start several containers.

But a parallel process runs will receive a new cloud environment if the maximum concurrency configuration for the currently running Testcontainers Cloud agent allows creating more cloud environments.

So if you have configured Turbo mode, but only one test process, then all the containers used in the tests will run in the same cloud environment, even if you are running tests in parallel within one process.

If you have multiple test processes (for example forked JVMs), then with Turbo Mode each JVM will, if possible, receive to a dedicated cloud environment.

Memory limit

Each Testcontainers Cloud session gets 8G of RAM. You can see this information in the logs when the connection is established:

SHELL

```
    Connected to docker:
      Server Version: 70+testcontainerscloud
      API Version: 1.41
      Operating System: Ubuntu 22.04 LTS
      Total Memory: 7396 MB
```

Currently, it’s not possible to extend or limit memory usage for a single Testcontainers Cloud session.

Check tests are running in the cloud

Explore the Connection section in the Testcontainers Desktop application.

If you installed the Testcontainers Cloud application on your local machine you should see the icon in the tray menu. To check that your tests are running with the Testcontainers Cloud application just click on the Testcontainers Cloud application icon, you'll see the connection state: Running or Passive. If you click on the Connection sub-menu you'll see the detailed connection information.

Connected to: zone/region you are connected to (connection latency in ms)

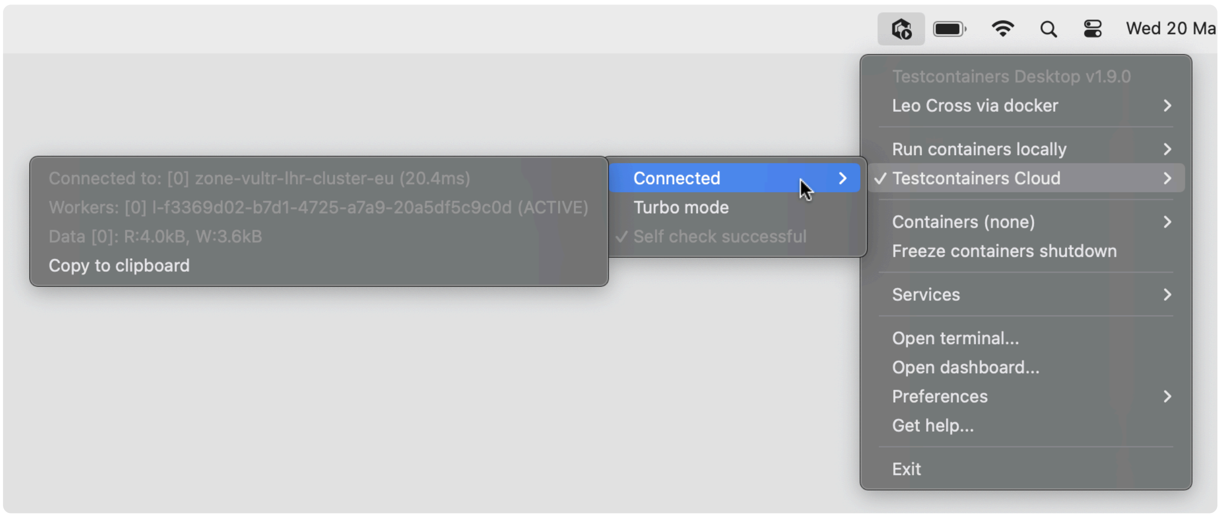
Leases: id of the opened connection (STATE).

`ACTIVE` Leases mean that your connection is active and you have Testconatiners Cloud resources allocated for your tests.

`PASSIVE` Leases mean that your connection is closed and no more Testconatiners Cloud resources are allocated.

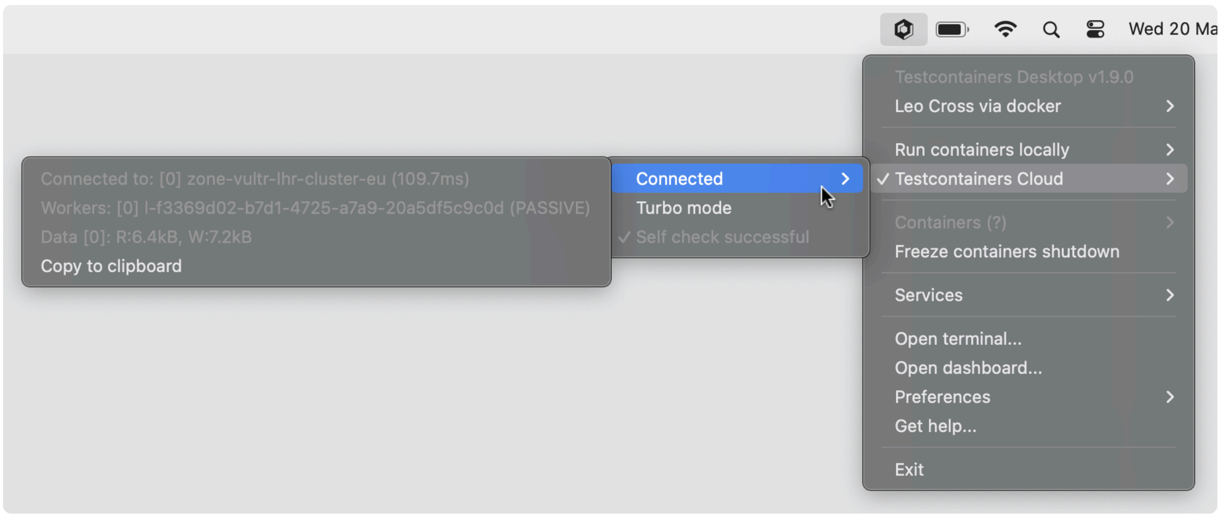
Data: R: reading kB, W: writing kB

Running State



```
Connected to: [0] zone-vultr-lhr-cluster-eu (20.4ms)
Workers: [0] l-f3369d02-b7d1-4725-a7a9-20a5df5c9c0d (ACTIVE)
Data [0]: R:4.0kB, W:3.6kB
```

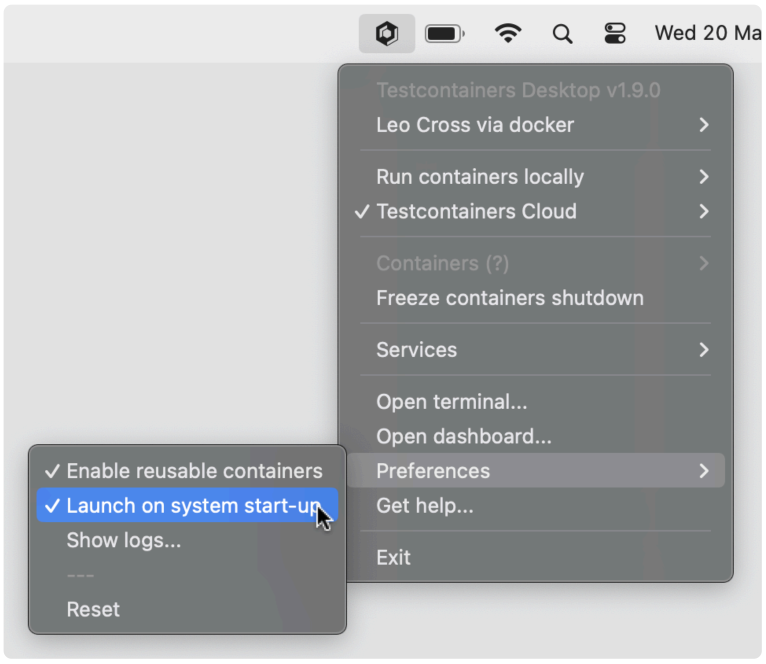
Passive State



Connected to: [0] zone-vultr-lhr-cluster-eu (109.7ms)
Workers: [0] l-f3369d02-b7d1-4725-a7a9-20a5df5c9c0d (PASSIVE)
Data [0]: R:6.4kB, W:7.2kB

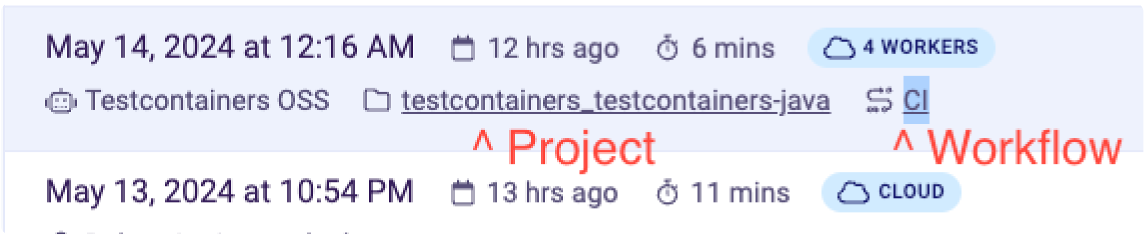
Automatically start the client on system restart

To bring more convenience into user experience we added an Autostart checkbox in the menu of our desktop application. If you want to start Testcontainers Cloud automatically after you turn on or restart your computer, just enable this checkbox. It works the same for all supported operating systems.



Tag test sessions by project

When you run your tests with Testcontainers Cloud, you can specify the project or workflow name and URL. This information will be displayed in the Testcontainers Cloud dashboard, making it easier to identify the tests and workflows that are running.



Please note, that url is optional and could be omitted. If you use GitHub Actions, the workflow name and URL will be automatically detected and displayed in the Testcontainers Cloud dashboard.

Via environment variables on CI

Set the

- `TCC_PROJECT_KEY` / `TCC_PROJECT_URL` environment variables. Alternatively, you could set the `tcc.project.key` and `tcc.project.url` properties (e.g. in your `~/.testcontainers.properties` file).
- For workflow, you could use the `TCC_WORKFLOW_KEY` and `TCC_WORKFLOW_URL` environment variables. Alternatively, you could set the `tcc.workflow.key` and `tcc.workflow.url` properties.

Via container labels (in your testcontainers tests)

- `cloud.testcontainers.tcc.project.key` / `cloud.testcontainers.tcc.project.url` - for project
- `cloud.testcontainers.tcc.workflow.key` / `cloud.testcontainers.tcc.workflow.url` - for workflow

Optimizing usage

As your team increases its adoption of Testcontainers Cloud, there might be a point where you wish to optimize your usage. This article describes some useful techniques.

Fine-tune turbo mode

Turbo mode allows you to run tests in parallel by requesting multiple workers. In general, this is a net-positive as your CI pipelines run faster, and overall usage stays relatively constant. For example, in an ideal case, a pipeline that previously took 15 minutes running on a single worker might instead take 5 minutes when being parallelized across 3 workers. However it's also possible to request more workers than is useful, increasing total usage without achieving significant speed improvements. When that happens, you can reduce the number of requested workers (e.g. `--max-concurrency` flag) and confirm that the duration stays constant, effectively reclaiming some unproductive usage.

Terminate workers eagerly when your tests finish

By default, workers remain active for a certain period of inactivity before they shutdown to provide a smooth user experience, particularly on desktop. In CI pipelines it's often possible to know that a specific pipeline or job will no longer need access to its Testcontainers Cloud worker(s) once it completes. In such cases, it's possible to optimize usage by eagerly terminating worker(s) on agent shutdown. To do so, simply pass the `--terminate` flag as an argument when starting the agent. Alternatively, you could call the `terminate` command while the agent is still running.

Compatibility with Docker

Using the Docker CLI

It is currently possible to talk directly to the underlying Docker API using the Docker context:

SHELL

```shell
docker context use tcc
docker ps
```

If you want to see information about CPU/memory usage for TCC sessions you can then use

SHELL

```shell
docker stats
```

Or

```
DOCKER_CONTEXT=tcc docker stats
```

However, note that Testcontainers Cloud is not a generic “Docker-as-a-Service” and this should be considered an implementation detail.

Mounting the local filesystem

Mounting of files from the local filesystem into containers is not implemented. Consider using the copyFileToContainer and copyFileFromContainer methods instead.

For example, if you can copy files to the container when definining it:

JAVA

```java
GenericContainer myContainer = new GenericContainer(ALPINE_IMAGE)
    .withCopyFileToContainer(
        MountableFile.forClasspathResource("/mappable-resource/"),
        directoryInContainer
    )
```

You can also copy files from the container like this:

JAVA

```java
myContainer.copyFileFromContainer(directoryInContainer + fileName,
destinationOnHost);
```

You can find more examples of working with files in the library documentation.

Network Configuration

Using internal Docker registries

For users who need to pull images from a ‘private’ registry that is not accessible from the public internet. Docker Hub, Amazon ECR, GCR, etc users *do not* need to use this feature, as these registries are exposed to the public internet.

Testcontainers Desktop usage:

Set per-user configuration in the `~/.testcontainers.properties` file:

```
cloud.private_registry.proxy.url =
https://private.registry.example.com:8999
```

It’s possible to configure more than one registry:

```
cloud.private_registry.proxy.url =
https://private.registry.example.com:8999
cloud.private_registry.proxy.url.second =
https://private2.registry.example.com:8999
cloud.private_registry.proxy.url.test =
https://test.registry.example.com:8999
```

> The part after .url. has to be unique, however, used only for convenience, so could be anything

Enable images such as `private.registry.example.com:8999/prefix/name` to be pulled:

```
cloud.private_registry.proxy.allowed_image_name_globs = **
```

Or more precisely:

```
cloud.private_registry.proxy.allowed_image_name_globs =
prefix/*,prefix/name
```

define this way comma-separated list of globs for images allow-listed for pulls (`**` means all). We would recommend that you keep the allowed list as small as possible.

It’s possible to ignore certificate-related issues with the:

```
cloud.private_registry.proxy.insecure_skip_verify = true
```

This is not recommended, as allows MitM attacks, however, could be used for testing purposes in case of connection issues.

Updates will be loaded on startup so you need to **Restart the Testcontainers Cloud application**

Agent CLI usage:

Add flags to the CLI invocation (the flag can be specified for each registry you want to enable):

```
--private-registry-url=https://private.registry.example.com:8999 --private-registry-url=https://private2.registry.example.com:8999
```

Enable images such as `private.registry.example.com:8999/prefix/name` or `private.registry.example.com:8999/name` to be pulled.

```
--private-registry-allowed-image-name-globs=**
```

Or more precisely:

```
--private-registry-allowed-image-name-globs=prefix/*,prefix/name
```

define this way comma-separated list of globs for images allow-listed for pulls (`**` means all). We would recommend that you keep the allowed list as small as possible.

Current limitations:

- Image pull is supported, but the push will be prevented.
- Proxying must be configured on a per-machine basis, but we expect this to be configurable organisation-wide later
- Credentials/tokens for all public/private docker registries are visible to the Testcontainers library and Testcontainers Cloud (data is proxied but not stored).
- Images pulled from private registries are cached within users' Testcontainers Cloud VM, which is deleted automatically after being idle for approximately 30 minutes.

As security measures, the agent will only allow proxying to a single configured registry host, restricts HTTP verbs to HEAD/GET, and only will enable requests which match an allowlist of paths (the path allowlist is based on the images which the agent is configured to allow).

At present these settings are configured on a per-installation basis, but we expect these to become centrally configurable at a later date.

How to use behind a Proxy

You can use environment variables or Testcontainers Settings to set up your Testcontainers Cloud client to point to your proxy. If you are running Testcontainers Cloud client in a network which uses an HTTP Proxy, you have three ways of setting up the client to use it.

1. Using environment variables

If you are launching the client from a shell which has any of the following environment variables set up to point to your proxy, the client will automatically pick these up and you're already good to go:

```
`http_proxy` or `HTTP_PROXY`
`https_proxy` or `HTTPS_PROXY`
`no_proxy` or `NO_PROXY`
```

2. Using the Testcontainers Cloud properties file

If you do not have the option of setting shell environment variables, you can also provide the proxy settings by adding any or all of the following optional lines:

```
http_proxy=host:port
https_proxy=host:port
no_proxy=host1, host2
```

to the `cloud.properties` file at one of the following locations:

MacOS and Linux

If `$XDG_CONFIG_HOME` is set:

```
$XDG_CONFIG_HOME/testcontainers/cloud.properties
```

Otherwise:

```
$HOME/.config/testcontainers/cloud.properties
```

Windows

If `$XDG_CONFIG_HOME` is set:

```
%xdg_config_home%\testcontainers\cloud.properties
```

Otherwise:

```
%userprofile%\.config\testcontainers\cloud.properties
```

3. Using the Testcontainers Settings

Alternatively, you can also provide the proxy settings in the `$HOME/.testcontainers.properties` file. The keys are the same as for Testcontainers Cloud properties file above.

Troubleshooting

Accessing logs

Testcontainers Desktop

The application writes the log into the file in the system-dependent location. The easiest way to access that location is via the menu `Preferences > Show logs...`

To locate the destination manually, these locations should be used:

- On macOS: `$HOME/Library/Logs/AtomicJar/testcontainers.cloud.desktop/testcontainers-cloud-desktop.log`
- On linux: `${XDG_CACHE_HOME:-$HOME}/.cache/AtomicJar/testcontainers.cloud.desktop/testcontainers-cloud-desktop.log`
- On windows: `%LocalAppData%/AtomicJar/testcontainers.cloud.desktop/testcontainers-cloud-desktop.log`

The log file is rotated once it gets bigger than 5Mb.

Testcontainers Cloud Agent

The agent writes logs to the stderr once executed. To store logs in a file, the agent could be launched this way:

SHELL

```
./tcc-agent > "./tcc/agent.log" 2>&1 &
```

This will redirect both stdout and stderr output to a specified location.

GitHub Action

Writing to a specific logfile supported out of the box in the official GitHub Action via `logfile` parameter. Later, the log file can be uploaded for download via actions/upload-artifact action.

Enable verbose logging

This can be useful for debugging issues with the help of additional debug log level enabled in the agent logs

Testcontainers Desktop usage:

Add a property into a per-user configuration in the `~/.testcontainers.properties` file:

```
cloud.logs.verbose = true
```

You would have to **Exit**, and then relaunch the client to apply the configuration change.

Agent CLI usage:

Add the flag to the CLI invocation:

```
--verbose
```

the same result could be achieved with the help of the environmental variable

```
TC_CLOUD_LOGS_VERBOSE=true
```

Agent CLI will also respect the property value from the Testcontainers Desktop usage section.

Alternatively, on linux, you could try:

Adding the following line into `~/.profile` file:

```
export TC_CLOUD_LOGS_VERBOSE=true
```

On the MacOS:

It’s possible to run desktop client from the terminal with full access to the environment variables management for debugging purposes:

SHELL

```shell
open -W --stdout $(tty) --stderr $(tty)
/Applications/Testcontainers\ Desktop.app
```

this way all the environment variables could be defined inline:

SHELL

```shell
TC_CLOUD_LOGS_VERBOSE=true open -W --stdout $(tty) --stderr $(tty)
```

```
/Applications/Testcontainers\ Desktop.app
```

How to verify if it was enabled:

If everything is correct, you will see in the log a similar line very early in the log flow:

```
2022-11-07T18:47:56.964Z --- DBG Verbose output enabled
```

No Docker activity detected

Common Causes

You may sometimes encounter a warning message stating *"this worker was requested by the Testcontainers Cloud agent but seemingly didn't run any containers."* This warning indicates that the worker did not perform any expected work. This can happen when:

1. **Running Docker commands directly in the terminal.** Typically, Testcontainers Cloud users rely on the Testcontainers open-source libraries to control the lifecycle of their containers. Executing raw Docker commands directly against the TCC Docker context is unsupported. For example, if you run Docker commands (e.g. `docker ps`) directly on your machine while the Testcontainers Cloud agent is active but no worker is currently connected, you might accidentally start a new worker.
2. **Starting the Testcontainers Cloud agent in CI pipelines without Testcontainers-based tests.** When starting the Testcontainers Cloud agent in your CI/CD pipelines, it is important to ensure that your pipeline jobs actually include Testcontainers-based tests. If you start the agent in pipelines that don't run any Testcontainers-based tests, the worker will start and wait, without performing any useful activity.

Troubleshooting Steps

If you encounter the "no activity detected" warning, consider the following troubleshooting steps:

1. Review CI pipeline configuration: inspect your CI/CD pipeline configuration and verify that the appropriate stages are set up to execute Testcontainers-based tests. Ensure that the pipeline triggers the execution of these tests.

2. Avoid Docker command interference: ensure that you are not running any Docker commands directly on your desktop machine while the Testcontainers Cloud agent is active. These external commands can interfere with the agent's ability to control and manage containers.
3. Validate Testcontainers integration: double-check the integration of Testcontainers within your test codebase. Ensure that the necessary dependencies and configurations are in place. Confirm that the tests correctly invoke Testcontainers APIs to create and manage containers.

Conclusion

The "No activity detected" warning in Testcontainers Cloud indicates that a worker did not run any containers as expected. By avoiding external Docker commands on the desktop and ensuring that CI pipelines execute Testcontainers-based tests, you can minimize "empty" workers and leverage the full capabilities of Testcontainers Cloud.

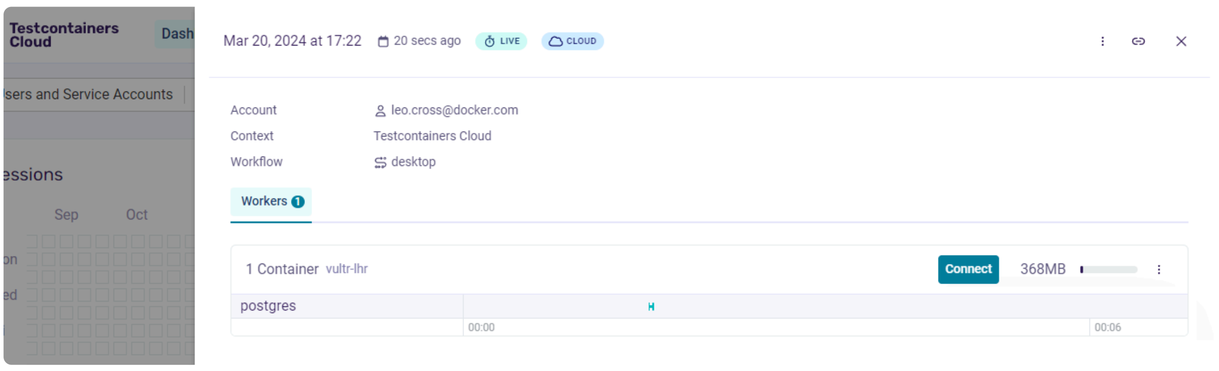
If the issue persists, consult the Testcontainers Cloud documentation or contact us to help further troubleshoot and resolve the problem.

How to connect to a cloud worker for troubleshooting

For advanced troubleshooting use cases, it can be useful to connect to a Testcontainers Cloud worker from your machine.

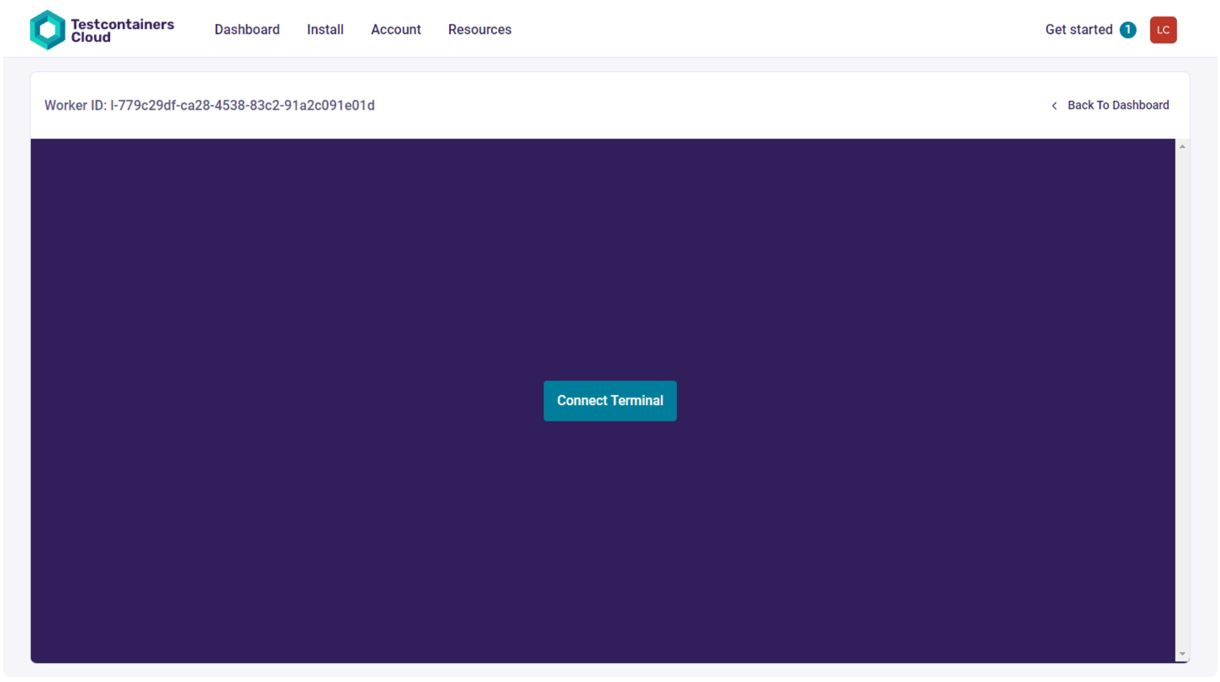
Connect with the web terminal

By navigating to dashboard, it's possible to see "live" activities for which cloud workers are available. By clicking a row and expanding the detail drawer and then clicking on the "Connect" button it is possible to open the web terminal.

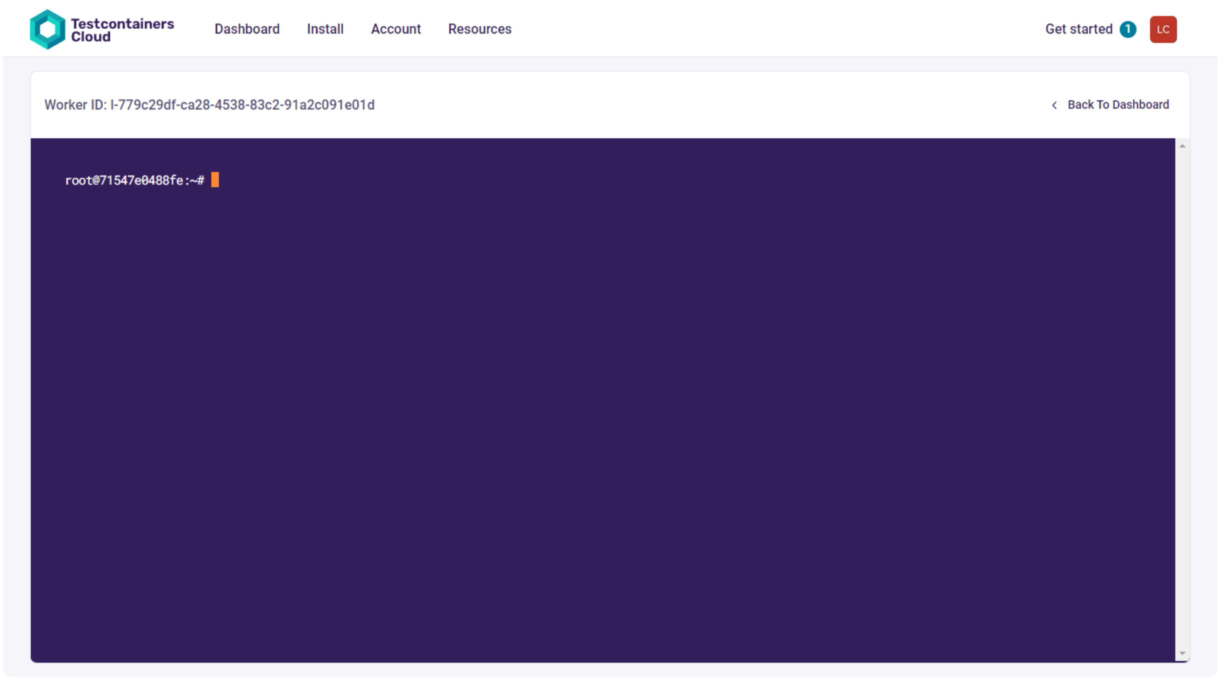

Mar 20, 2024 at 17:22 · 20 secs ago · LIVE · CLOUD

Account: leo.cross@docker.com
Context: Testcontainers Cloud
Workflow: desktop

One the web terminal is open clicking "Connect Terminal" will initiate a secure connection to the selected worker.



Worker ID: l-779c29df-ca28-4538-83c2-91a2c091e01d



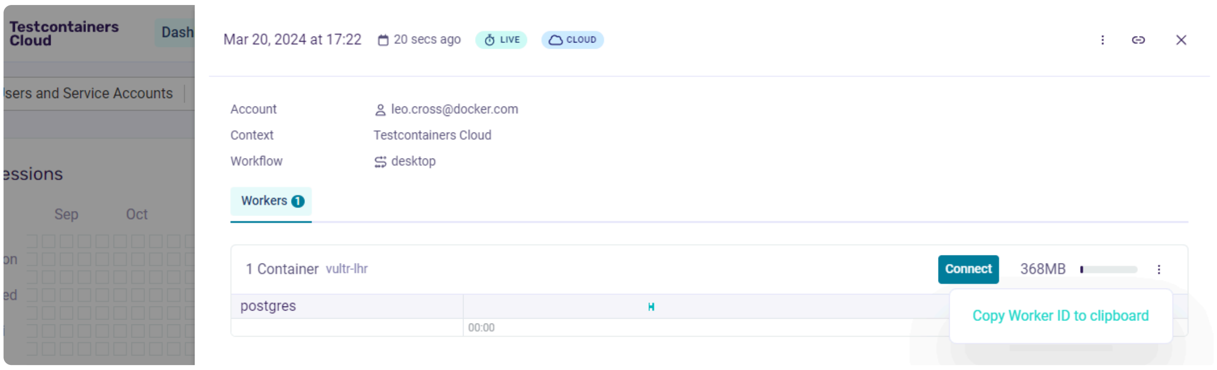
Worker ID: l-779c29df-ca28-4538-83c2-91a2c091e01d

```
root@71547e0488fe:~#
```

Connect with a local terminal

1. Obtain the “Worker ID”

By navigating to dashboard, it’s possible to see “live” activities for which cloud workers are available. By clicking a row and expanding the detail drawer and then clicking on the “kebab menu” (the 3 dots) on the right side of the desired worker it’s possible to copy the “Worker ID” to the clipboard.



2. Download the CI agent and make it executable

The connect feature is part of the Testcontainers Cloud binary. If you don't already have the agent, you can download it from the install page or directly at https://get.testcontainers.cloud/bash.

SHELL

```
curl -o tcc-agent -L
https://get.testcontainers.cloud/testcontainers-cloud-
agent_darwin_arm64
chmod +x tcc-agent
```

3. Connect to the cloud worker

In a terminal window, use the agen'ts `connect` method with the Worker ID as parameter:

SHELL

```
./tcc-agent connect <worker-id>
```

Assuming that you run Testcontainers Desktop and are signed-in, the Testcontainers Cloud agent should automatically reuse the authentication token and no further step is required. If that's not the case, see the next section.

4. (optional) Provide a Testcontainers Cloud authentication token

If you'd like to pass the token manually, you can either set the `TC_CLOUD_TOKEN` environment variable or provide it as a command line argument:

SHELL

```
./tcc-agent --token <token> connect <worker-id>
```

Slower tests

Generally, with fast internet, tests should be on par or faster than with the local Docker Desktop. If you notice the test speed downgrade - check the Connection submenu of the Testcontainers Desktop app, it will show the last captured latency. Ideally the connection latency should be below 20ms.

Also, with TCC, you can run tests in parallel, and, with the Turbo Mode enabled, you can have each test fork connected to its own VM, so you can literally run tests 3-4 times faster.

TCC for CI

Starting the agent in a CI job

To use Testcontainers Cloud in your CI environment please add a step to your CI job that starts the Testcontainers Cloud agent before you run your tests. Do not forget to set your Testcontainers Cloud token as well.

It is also recommended that you make use of the built-in `wait` command to block until a successful connection to Testcontainers Cloud has been established, so you can be confident that your tests will interact with Testcontainers Cloud in a ready state:

SHELL

```
# get the agent binary and execute it immediately
sh -c "$(curl -fsSL https://get.testcontainers.cloud/bash)"

# run your Testcontainers powered tests as usual
mvn verify
```

You also need to set your **Testcontainers Cloud token**. For many environments, the most convenient way is using an environment variable, `TC_CLOUD_TOKEN`. However, please keep the token as secret as possible.

Using TCC with Github Actions

To use Testcontainers Cloud with your GitHub Actions you just need to make sure `TC_CLOUD_TOKEN` is set to your corresponding token value, then download the agent and start it. You can use the following script to add necessary steps to your workflow :

YAML

```yaml
build:
  env:
    TC_CLOUD_TOKEN: $
  steps:
    - name: Prepare Testcontainers Cloud agent
      if: env.TC_CLOUD_TOKEN != ''
      uses: atomicjar/testcontainers-cloud-setup-action@main
    # ... existing steps go here (checkout, run tests, etc.)
```

If you are seeing jobs triggered by Dependabot fail make sure you have also set `TC_CLOUD_TOKEN` in Dependabot secrets.

Using TCC with Kubernetes

You can also use Testcontainers Cloud in Kubernetes based CI environments (such as Tekton or Jenkins X). Just download the agent, start it and make sure `TC_CLOUD_TOKEN` is set to your corresponding token value:

YAML

```yaml
command:
    - sh -c "$(curl -fsSL https://get.testcontainers.cloud/bash)"
    - mvn verify
envs:
    - name: TC_CLOUD_TOKEN
      valueFromSecretRef: tccToken
```

Using TCC with CircleCI

To use Testcontainers Cloud with CircleCI you need to set `TC_CLOUD_TOKEN` to your corresponding service account token. You can generate the token in the Testcontainers Cloud dashboard. Then add the token to your CircleCI workflow by setting the environment variable in your CircleCI Project Settings.

Next you need to configure your CircleCI workflow to install and use Testcontainers Cloud. You can add the testcontainers-cloud-orb setup as a pre-step to your CircleCI Job. And configure it to start the Testcontainers Cloud agent before running the tests.

You can use the following script to add the steps to your workflow, note the “tcc:” orb, and the “tcc/setup” config in the “pre-steps”:

YAML

```yaml
version: "2.1"
orbs:
  tcc: atomicjar/testcontainers-cloud-orb@0.1.0
workflows:
  workflow_name:
    jobs:
      - job_name:
          # ... existing steps go here (run tests, etc.)
          pre-steps:
            - tcc/setup
```

Using TCC with Cloud Build

To use Testcontainers Cloud with Google Cloud Build you need to set `TC_CLOUD_TOKEN` to your corresponding service account token. You can generate the token in the [Testcontainers Cloud dashboard](). You can store your service account token securely in [Google Cloud Secrets Manager](). Once the secret is created, grant access to the principal `YOUR_PROJECT_ID@cloudbuild.gserviceaccount.com`, and assign the `Secret Manager Secret Accessor` role to it.

Next you need to install and start Testcontainers Cloud agent by setting `TC_CLOUD_TOKEN` environment variable by looking up the value from `Google Cloud Secrets Manager`.

You can use the following script for your `cloudbuild.yaml` configuration:

YAML

```yaml
steps:
    - name: "docker-image:tag" # ex: maven:3-eclipse-temurin-19
      args:
          - "-c"
          - |
              curl -fsSL https://get.testcontainers.cloud/bash |
sh
              cp ~/.testcontainers.properties
/root/.testcontainers.properties
              # Your test command like "mvn test" or "npm test"
      dir: "${_APP_NAME}"
      entrypoint: bash
      secretEnv:
          - TC_CLOUD_TOKEN
availableSecrets:
    secretManager:
        - versionName:
projects/<PROJECT_ID>/secrets/TC_CLOUD_TOKEN/versions/latest
          env: TC_CLOUD_TOKEN
```

Using TCC with GitLab CI/CD

To use Testcontainers Cloud with GitLab you need to set `TC_CLOUD_TOKEN` to your corresponding service account token. You can generate the token in the Testcontainers Cloud dashboard. Then add the token to your GitLab job by setting environment variable in your GitLab Project Settings.

Next you need to configure your GitLab pipeline job to install and use Testcontainers Cloud. You can use the Testcontainers Cloud agent installation script and start it before running your tests.

You can use the following configuration for your `.gitlab-ci.yml`:

YAML

```yaml
image: "docker-image:tag" # ex: maven:3-eclipse-temurin-19

job_name:
    stage: test
    script:
        - curl -fsSL https://get.testcontainers.cloud/bash | sh
        -  # ... existing steps go here (run tests, etc.)
```

Using TCC with Azure Pipelines

To use Testcontainers Cloud with Azure Pipelines you need to set TC_CLOUD_TOKEN to your corresponding service account token. You can generate the token in the Testcontainers Cloud dashboard. Then add the token to your pipeline by setting an variable in your pipeline settings and using it as an environment variable in for the task.

Next you need to configure your Azure Pipelines pipeline job to install and use Testcontainers Cloud. You can use the Testcontainers Cloud agent installation script and start it before running your tests.

You can use the following configuration step for your `azure-pipelines.yml` before the pipeline executes the build tool command to run the tests.

YAML

```yaml
steps:
- task: Bash@3
  inputs:
    targetType: 'inline'
    script: 'sh -c "$(curl -fsSL
https://get.testcontainers.cloud/bash)"'
  env:
    TC_CLOUD_TOKEN: $(TC_CLOUD_TOKEN)
```

Using TCC with Jenkins

To use Testcontainers Cloud with Jenkins first you need generate the token in the Testcontainers Cloud dashboard. Then you can store the token as a Secret as follows:

- From **Dashboard** go to **Manage Jenkins -> Manage Credentials**
- Under **Stores scoped to Jenkins** click on **(globals)** domain and **Add Credentials**
- Provide the following values and click on **Create**
  - **Kind:** Secret text
  - **Secret:** `YOUR_TOKEN_VALUE`
  - **ID:** `tc-cloud-token-secret-id`

Next, you should configure your Jenkins Pipeline to look up the credentials and set the `TC_CLOUD_TOKEN` environment variable. You can use the Testcontainers Cloud agent installation script and start it before running your tests.

You can use the following configuration step for your `Jenkinsfile` before the pipeline executes the build tool command to run the tests.

```
pipeline {
    agent any

    environment {
        TC_CLOUD_TOKEN = credentials('tc-cloud-token-secret-id')
    }

    stages {
        stage('TCC SetUp') {
            steps {
                sh "curl -fsSL
https://get.testcontainers.cloud/bash | sh"
            }
        }
        stage('Test') {
            steps {
                // run your test command
            }
        }
    }
}
```

Using TCC with Tekton

To use Testcontainers Cloud with Tekton you need to set `TC_CLOUD_TOKEN` to your corresponding service account token. You can generate the token in the Testcontainers Cloud dashboard. Then add the token to your pipeline by creating a secret and using it as a variable in your pipeline task.

For example, you can define a secret in a separate secret.yaml file like in the following example:

YAML

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: tcc-token
stringData:
  token: YOUR_TOKEN_VALUE_GOES_HERE
```

Next you need to configure your Tekton pipeline to install and use Testcontainers Cloud agent prior running the tests. For that you can use the Testcontainers Cloud agent installation script.

Here's a sample configuration for a Tekton task task.build.yaml which bind the above secret to the `TC_CLOUD_TOKEN` environment variable, and runs the script that installs and starts Testcontainers Cloud agent, and then calls to the Maven test command.

YAML

```yaml
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: build-java-maven
spec:
  workspaces:
    - name: source
  steps:
    - name: maven-build
      image: amazoncorretto:17.0.5-al2
      env:
        - name: TC_CLOUD_TOKEN
          valueFrom:
            secretKeyRef:
              name: tcc-token
              key: token
      script: |
        set -e
        cd "$(workspaces.source.path)"
        sh -c "$(curl -fsSL
https://get.testcontainers.cloud/bash)"
        ./mvnw test
```

This is a flexible approach you can use to run Testcontainers Cloud in Tekton with the tests in other languages by swapping the `mvn` command for the corresponding build tool call.

Billing

How usage is measured and billed

Overview

Usage for Testcontainers Cloud is triggered by the Testcontainers Cloud Agent. In order to trigger usage, the Agent needs to be authenticated with the credentials of an authorized user within the customer's account. An authorized user can be either a Member (typically a person) or a Service Account (typically used by an automated system).

Usage is metered differently for Members and Service Accounts:

- For Service Accounts, see Testcontainers Cloud for CI below.
- For Members, see Testcontainers Cloud for Desktop below.

Testcontainers Cloud for CI

Usage is metered in `Worker Minute`.

When the Testcontainers Cloud Agent is deployed in a Continuous Integration (CI) environment, it is authenticated with the credentials for a Service Account. When the CI executes its workflow (sometimes called a "build" or "pipeline"), the Agent allocates one or several Worker(s) in order to deliver the Testcontainers Cloud service. For each Worker, usage is metered while the Worker is allocated. Allocation starts when the Worker is requested on-demand via the Agent by Service Account activity, and ends when the Worker is terminated after a period of inactivity.

The customer controls whether the Agent obtains one or several Workers from the Testcontainers Cloud Service by setting the `max-concurrency` flag. Setting this flag above 1 allows the customer to parallelize tests across multiple Workers thanks to Turbo mode.

Example 1

The Testcontainers Cloud Agent is deployed in the CI where it runs twice a day from Monday to Friday. Each execution requires a single Testcontainers Cloud Worker for 17 minutes.

Each "build" generates 17 minutes of usage on average. Over the course of 4 weeks, the CI is triggered 40 times, for a total usage of of 680 Worker Minutes.

Example 2

The Testcontainers Cloud Agent is deployed in the CI where it runs 10 times a day from Monday to Sunday. Each execution requires 4 Testcontainers Cloud Workers running in parallel. 3 of the Workers complete in 4 minutes while one completes in 8 minutes.

In total, each build generates 20 minutes of usage. Over the course of 30 days, the CI is triggered 300 times, for a total usage of 6,000 Worker Minutes.

Testcontainers Cloud for Desktop

Usage is metered in `Seats`.

Usage of the Testcontainers Cloud Agent by a Member of the Organization counts as a single Seat regardless of the number of Worker Minutes generated during a month (within the limits set by the Fair Use policy). This includes cases where the Member parallelizes tests across multiple Workers thanks to Turbo mode.

Example 3

An Organization includes 2 Members:

1. Alice is the Admin for the organization. She doesn't actively use the product and so Alice doesn't generate a Seat.
2. Barbara is a developer who uses the product several times every work day, with Turbo mode activated. Overall, Barbara triggers 2,500 Worker Minutes per month, but counts as a single Seat.

This organization has 2 Members generating a single Seat of usage for the month.

Frequently Asked Questions

When does the billing cycle start?

The billing cycle begins the first of the month UTC, regardless of when you sign up.

For the very first month of usage, Testcontainers Cloud for CI is billed based on actual Worker Minutes used after the upgrade date, and Testcontainers Cloud for Desktop is prorated based on the Organization upgrade date.

For the following months of usage, Testcontainers Cloud for CI is billed based on actual Worker Minutes used, and Testcontainers Cloud for Desktop is billed only for Members that joined the organization prior to the start of the month. This means that if a Member generates usage during the month they join, their Seat is waived as a courtesy until the start of the following month.

What is the Fair Use policy?

While a Seat grants "unlimited" usage of Testcontainers Cloud for Desktop, some practical limits exist to deter potential abuse of the service (e.g. crypto-mining). Under the Fair Use policy, a Seat grants a quota of 3,000 Worker Minutes per month. This quota is above the upper range that we observe in practice for the vast majority of users. Moreover, because quotas are summed for all the Seats used, even a few "power users" within an Organization typically don't exhaust the entire quota for their team.

What is a free plan?

With Testcontainers Desktop being free for solo developers, access to Testcontainers Cloud has some restrictions. Simply reach out to us to lift the restrictions and try all of Testcontainers Cloud!

What features are restricted in a free plan?

To make it easy to burst your tests to the cloud, we are offering a monthly quota of free for solo developers Testcontainers Cloud usage with the following restrictions:

- You can use Testcontainers Cloud on the desktop with a monthly limit of 300 minutes of cloud runtime.
- You can use Testcontainers Cloud on the desktop, but Turbo mode is disabled, which means that all tests execute on a single worker.
- You can use Testcontainers Cloud on the desktop, but only in solo.
- You can try Testcontainers Cloud with your CI but with a single Service Account running a single session at a time (i.e., no concurrent builds and no turbo mode).

Want to stay up to date?

Read the Newsletter

Links

- Community Champions
- Testcontainers for Java
- Testcontainers for Go
- Testcontainers for .NET
- Testcontainers for Node.js
- Testcontainers for Clojure
- Testcontainers for Elixir
- Testcontainers for Haskell
- Testcontainers for Python
- Testcontainers for Ruby
- Testcontainers for Rust

Join the community

We hope that you find Testcontainers reliable and intuitive to use. However sometimes things don't go the way we'd expect and we'd like to try and help out if we can.