Troubleshooting

- Run diagnostics
- Sandbox is not responding
- Restart the sandbox daemon
- Resetting sandboxes
- Sandbox doesn't contain my project files
- Kiro, Copilot, or Droid shorthand fails
- Agent can't install packages or reach an API
- Kit fails to install: source not in allowlist
- Kit install or startup command fails
- SSH and other non-HTTP connections fail
- Can't reach a service running on the host
- Docker authentication failure
- Agent authentication failure
- MCP server streams stall
- API calls fail with a certificate error
  - Certificate errors during Docker builds
- Sandbox runs out of disk space
- Filesystem operations are slow in large repositories
- Clone mode reports "not in a Git repository" on WSL
- SSH agent socket is missing
- Sandbox commits aren't signed
- Daemon fails to start after downgrading
- Removing all state
- Enable automatic diagnostics uploads
- Report an issue

The following diagnostics and recovery steps apply to local sandboxes. Use `sbx --cloud diagnose` to check cloud connectivity and account access. For cloud files, expiration, and network access, see

Cloud sandboxes. Local daemon restarts and `sbx reset` do not repair cloud sandbox state.

Run diagnostics

Before digging into a specific issue, run `sbx diagnose` to check for common problems with your installation, such as a missing CLI binary, daemon reachability problems, a CLI/daemon version mismatch, missing storage directories, or broken authentication.

```
$ sbx diagnose
```

The command prints a summary of checks that passed, warned, or failed, along with suggested fixes. Use `--json` or `--format yaml` for machine-readable output, or `--format github-issue` to generate a Markdown snippet suitable for pasting into a GitHub issue.

Sandbox is not responding

If `sbx ls` shows `running (not responding)`, Docker Sandboxes has detected that the sandbox isn't responding to commands. Stop and start that sandbox, then retry your command:

```
$ sbx stop <SANDBOX_NAME>
$ sbx start <SANDBOX_NAME>
```

Files saved in the sandbox are kept when you stop and start it.

Restart the sandbox daemon

If sandbox commands hang, fail to connect to the daemon, or keep returning daemon errors, restart the sandbox daemon before resetting sandbox state:

```
$ sbx daemon restart
```

Then retry the command that failed. Restarting the daemon doesn't delete sandbox data. If the issue persists or state is corrupted, use `sbx reset`.

Resetting sandboxes

If you hit persistent issues or corrupted state, run `sbx reset` to stop all VMs and delete all sandbox data. Create fresh sandboxes afterwards.

Sandbox doesn't contain my project files

Starting with `sbx` version 0.42.0, the workspace path is optional for `sbx create`. When you omit it, the command creates a mountless sandbox. For example, these commands create and attach to a sandbox without mounting your host project files:

```
$ sbx create --name <sandbox-name> <agent>
$ sbx run --name <sandbox-name>
```

By contrast, `sbx run` mounts the current directory when you don't pass a workspace path:

```
$ sbx run <agent>
```

A sandbox's workspace configuration is fixed when the sandbox is created. To reuse the name of an existing mountless sandbox, first copy out any files you want to keep, then remove and recreate it with a workspace path:

```
$ sbx rm <sandbox-name>
$ sbx run --name <sandbox-name> <agent>
```

See Choose a workspace for mountless, direct, and clone-mode behavior.

Kiro, Copilot, or Droid shorthand fails

In Docker Sandboxes v0.42, `sbx run kiro`, `sbx run copilot`, and `sbx run droid` fail because these agents moved from built-in agents to public kits and their shorthand names aren't resolved in this release.

Upgrade Docker Sandboxes to v0.43.0 or later to launch these agents by name again. If you need to stay on v0.42, use the full kit reference for your agent:

```
$ sbx run docker.io/sbx/kiro-kit:latest
$ sbx run docker.io/sbx/copilot-kit:latest
$ sbx run docker.io/sbx/droid-kit:latest
```

Agent can't install packages or reach an API

Sandboxes use network access rules to control outbound traffic. If the agent fails to install packages or call an external API, the target domain is likely not in the allow list. Check which requests are being blocked:

```
$ sbx policy log
```

Then allow the domains your workflow needs:

```
$ sbx policy allow network "*.npmjs.org,*.pypi.org,files.pythonhosted.org"
```

To allow all outbound traffic instead:

```
$ sbx policy allow network "**"
```

If `sbx policy allow` doesn't unblock the request, your organization may manage sandbox policies centrally and take precedence over local rules. See Organization policies.

Kit fails to install: source not in allowlist

If loading a kit fails with a message like its source is not in your allowlist:

```
$ sbx run claude --kit "git+https://github.com/docker/sbx-kits-contrib.git#dir=vale"
```

`sbx` restricts kit installs to an allowlist of sources, which defaults to Docker Hub (`docker.io/`) only. Add the kit's publisher to the `kit.allowedSources` setting, keeping the entries you want to retain:

```
$ sbx settings set kit.allowedSources '["docker.io/","github.com/docker/"]'
```

Then run the command again. For details, including how to allow local kits or any remote source, see Restrict kit sources.

Kit install or startup command fails

If `sbx create` or `sbx run` shows ✗ on a kit's install command, or warns that a startup command failed, the full output is inside the sandbox. Read it with `sbx exec`:

```
$ sbx exec <sandbox-name> cat /var/log/sbx-kits/<kit>/install-<n>.log
$ sbx exec <sandbox-name> cat /var/log/sbx-kit-startup.log
```

`<n>` is the install command's position in the kit's `install` list, starting at 0. To stream every install command's output and the startup log while the sandbox is created, run `sbx -D create` or `sbx -D run`. See Debug a kit for how to read the output and fix the common causes.

SSH and other non-HTTP connections fail

Non-HTTP TCP connections such as SSH can be allowed by adding a policy rule for the destination. Hostname rules work for these connections because the sandbox recovers the hostname from its DNS resolver when the protocol doesn't include one:

```
$ sbx policy allow network "myhost:22"
```

If the destination is reached by IP address without a DNS lookup, the hostname can't be recovered. Use an address-based rule in that case:

```
$ sbx policy allow network "10.1.2.3:22"
```

UDP requires experimental UDP egress and UDP allow rules. ICMP is blocked and can't be unblocked with policy rules.

For Git operations over SSH, you can either add an allow rule for the Git server's hostname or IP address, or use HTTPS URLs instead:

```
$ git clone https://github.com/owner/repo.git
```

Can't reach a service running on the host

If a request to `127.0.0.1` or a local network IP returns "connection refused" from inside a sandbox, the address is not reachable from within the sandbox VM. See Accessing host services from a sandbox.

Docker authentication failure

If you see a message like `You are not authenticated to Docker`, your login session has expired. In an interactive terminal, the CLI prompts you to sign in again. In non-interactive environments such as scripts or CI, run `sbx login` to re-authenticate.

Agent authentication failure

If the agent can't reach its model provider or you see API key errors, the key is likely invalid, expired, or not configured. Verify it's set in your shell configuration file and that you sourced it or opened a new terminal.

For agents that use the credential proxy, make sure you haven't set the API key to an invalid value inside the sandbox — the proxy injects credentials automatically on outbound requests.

If credentials are configured correctly but API calls still fail, check `sbx policy log` and look at the **PROXY** column. Requests routed through the `transparent` proxy don't get credential injection. This can happen when a client inside the sandbox (such as a process in a Docker container) isn't configured to use the forward proxy. See Monitoring network activity for details.

MCP server streams stall

If a remote MCP server's HTTP/2 handling stalls long-lived streams, register it with `--disable-http2` to use HTTP/1.1:

```
$ sbx mcp add acme --url https://mcp.acme.com/mcp --disable-http2
```

Replace the example URL with your MCP endpoint. The setting applies to later connections to this server. The flag requires `--url` and can't be used with `--command` or `--local`. For registration options, see Register an MCP server.

API calls fail with a certificate error

If your organization uses a proxy that inspects HTTPS traffic, agent requests can fail with a certificate error such as `SSL certificate problem: self-signed certificate in certificate chain`. Install your organization's internal root CA inside the sandbox so the agent and its SDKs trust certificates signed by the proxy. Certificate errors can stop a request before the credential proxy can inject credentials.

For repeatable setup with a built-in agent, create a v2 mixin kit that installs the CA when the sandbox is created. See Install an internal CA certificate for an example kit.

Use a PEM-encoded certificate with a `.crt` extension. If traffic can be signed by more than one internal proxy, install each proxy's root CA before running `update-ca-certificates`.

Create a sandbox with the kit:

```
$ sbx run claude --kit ./internal-ca/
```

To update an existing sandbox, copy the certificate into the sandbox and update the trust store:

```
$ sbx cp ./internal-ca.crt <sandbox-name>:/tmp/internal-ca.crt
$ sbx exec <sandbox-name> -- sudo install -m 0644 /tmp/internal-ca.crt /usr/local/share/ca-certificates/internal-ca.crt
$ sbx exec <sandbox-name> -- sudo update-ca-certificates
```

> [!IMPORTANT] Install the CA into the system trust store with `update-ca-certificates`, as shown above. Don't override the sandbox's TLS trust variables (such as `SSL_CERT_FILE`) to point at only your internal CA. Doing so replaces the system bundle and breaks the trust the credential proxy depends on, so requests on the `forward` egress path fail.

If API calls still fail after installing the CA, run `sbx policy log` and check the egress path in the **PROXY** column:

- `forward`: the credential proxy terminates TLS and presents its own certificate, which the sandbox already trusts. Requests on this path don't need the internal CA, and overriding

the sandbox's trust variables breaks them, as described above.
- `forward-bypass` and `transparent`: the proxy forwards packets to the upstream proxy without terminating TLS, so the sandbox sees your organization's certificate directly. These paths are where installing the internal CA applies. The only difference between them is whether the client knows it's talking to a proxy.

Certificate errors during Docker builds

Containers started by the sandbox's Docker Engine have their own trust stores. They don't inherit the sandbox's installed certificates. If an HTTPS download in a Dockerfile fails with `self signed certificate in certificate chain`, install the proxy CA in the build image before the download.

From a shell inside the sandbox, write its proxy CA into your build context:

```
$ printf '%s' "$PROXY_CA_CERT_B64" | base64 -d > sbx-proxy-ca.crt
```

For a Debian-based image with `ca-certificates` installed, copy the certificate and update the trust store before commands that make HTTPS requests:

```
FROM python:3.13-slim
COPY sbx-proxy-ca.crt /usr/local/share/ca-certificates/sbx-proxy-ca.crt
RUN update-ca-certificates
```

Build inside the sandbox, passing its proxy settings to the build:

```
$ docker build --build-arg HTTP_PROXY --build-arg HTTPS_PROXY --build-arg NO_PROXY -t my-app .
```

If your organization also inspects TLS, install its root CA in the build image as a separate `.crt` file before `update-ca-certificates`. Keep the system CA bundle intact so the image trusts both proxies and public certificate authorities. The earlier certificate troubleshooting explains which proxy paths need each CA.

Keep the generated proxy certificate out of version control. Regenerate it and rebuild the image if the sandbox proxy CA changes.

Sandbox runs out of disk space

The sandbox root (`/`) filesystem defaults to 20 GB. To increase it, set `DOCKER_SANDBOXES_ROOT_SIZE` before creating the sandbox:

```
$ DOCKER_SANDBOXES_ROOT_SIZE=40g sbx run claude
```

`DOCKER_SANDBOXES_ROOT_SIZE` controls the root filesystem size. The Docker data disk at `/var/lib/docker` is independent and defaults to 10 GB. To change the Docker data disk size for a sandbox, set `DOCKER_SANDBOXES_DOCKER_SIZE` when you create it:

```
$ DOCKER_SANDBOXES_DOCKER_SIZE=20g sbx run claude
```

The Docker data disk must be at least 512 MiB. The environment variable doesn't resize existing volumes.

For a clone-mode sandbox, set `DOCKER_SANDBOXES_CLONED_WORKSPACE_SIZE` before creating the sandbox to configure the cloned workspace volume capacity. The variable accepts human-readable size strings such as `100g`:

```
$ DOCKER_SANDBOXES_CLONED_WORKSPACE_SIZE=100g sbx run --clone claude .
```

Filesystem operations are slow in large repositories

Filesystem operations such as `git status`, `git log`, or directory scans can be noticeably slow when you pass a workspace path and use direct mode. Virtiofs caching speeds up these workloads. Clone-mode sandboxes always enable it, so this tuning applies only to direct mode.

Virtiofs caching is enabled by default on all operating systems. If you experience Git index corruption or unexpected file content, disable caching with the kill switch and recreate the sandbox:

```
$ DOCKER_SANDBOXES_ENABLE_VIRTIOFS_CACHE=0 sbx run <agent>
```

Clone mode reports "not in a Git repository" on WSL

On Windows, running `sbx run --clone` against a repository on a WSL filesystem (a `\\wsl.localhost\...` path) can fail even though the directory is a valid Git repository:

```
> sbx run --clone claude \\wsl.localhost\Ubuntu\home\you\repo
error: --clone requires a Git repository, but \\wsl.localhost\Ubuntu\home\you\repo is not in a
```

```
Git repository
```

The cause is Git's dubious ownership check. When Git on Windows accesses a repository owned by a different user across the WSL boundary, it refuses to operate on it, so the underlying repository detection fails:

```
> git -C \\wsl.localhost\Ubuntu\home\you\repo rev-parse --show-toplevel
fatal: detected dubious ownership in repository at '//wsl.localhost/Ubuntu/home/you/repo'
```

Add the repository to Git's `safe.directory` list to allow access, then run the command again:

```
> git config --global --add safe.directory '%(prefix)///wsl.localhost/Ubuntu/home/you/repo'
> sbx run --clone claude \\wsl.localhost\Ubuntu\home\you\repo
```

SSH agent socket is missing

If `SSH_AUTH_SOCK` is set inside a sandbox but `ssh-add -L` reports `No such file or directory`, check whether your custom template includes `socat`. Docker Sandboxes uses it to create the socket that forwards requests to your host SSH agent. Installing OpenSSH client tools alone isn't enough.

For an Ubuntu-based custom template, add `socat` to the packages installed as root in your Dockerfile, then rebuild the template and create a sandbox from it. Docker-provided sandbox templates already include `socat`.

If the socket exists but forwarding still fails, check the SSH agent settings.

Sandbox commits aren't signed

Docker Sandboxes can sign Git commits with SSH keys from your host agent. For setup steps, see Commit signing.

Forwarding is enabled by default. Check `ssh.agentForwardingEnabled` and `ssh.agentSocketPath` to confirm that forwarding is enabled and inspect the socket selection:

```
$ sbx settings get ssh.agentForwardingEnabled
$ sbx settings get ssh.agentSocketPath
```

If you use each client's current `SSH_AUTH_SOCK`, reconnect from a shell where it points to the intended agent. If `ssh.agentSocketPath` returns a path, confirm that it points to an active host

agent. After changing forwarding or the socket selection, run `sbx daemon restart`.

If `ssh-add -L` prints `The agent has no identities.`, the sandbox can reach the forwarded agent, but the host agent doesn't have a loaded key. Load the signing key into your host SSH agent:

```
$ ssh-add ~/.ssh/id_ed25519
```

If commit signing works on the host but fails in a sandbox, check whether Git is configured to sign with a host file path such as `/Users/me/.ssh/id_ed25519.pub`. The sandbox uses the forwarded SSH agent, not the host key file path. Use the inline public key form instead:

```
$ git config --global gpg.format ssh
$ git config --global user.signingkey "key::$(ssh-add -L | head -n 1)"
```

If Git reports that `ssh-keygen` is missing, use a sandbox template that includes OpenSSH client tools.

If `git log --show-signature` reports that `gpg.ssh.allowedSignersFile` needs to be configured, Git can't verify the SSH signature locally. This verification config isn't required to create signed commits. GitHub uses the SSH signing keys configured in your GitHub account to verify commits.

GPG and S/MIME signing keys aren't available inside the sandbox. If your repository or organization requires GPG or S/MIME signatures, or if SSH signing isn't configured, use one of these workarounds:

- Commit outside the sandbox. Let the agent make changes without committing, then commit and sign from your host terminal.
- Sign after the fact. Let the agent commit inside the sandbox, then re-sign the commits on your host:

  ```
  $ git rebase --exec 'git commit --amend --no-edit -S' origin/main
  ```

  This replays each commit on the branch and re-signs it with your local signing key.

Daemon fails to start after downgrading

If you downgrade `sbx` to a version older than the one that last managed your local state, the daemon may fail to start with a database version mismatch:

```
error: failed to start backend in-process: start backend: creating containerd
server: ... database is at major version 6, but this binary only supports up
to major version 1
```

A newer version of `sbx` upgraded the local database to a schema that older binaries don't understand. To recover, reset all sandbox state:

```
$ sbx reset --preserve-secrets
```

This stops all VMs and deletes all sandbox data. You'll need to create new sandboxes afterwards. The `--preserve-secrets` flag keeps any secrets you've set so you don't have to reconfigure them.

Removing all state

As a last resort, if `sbx reset` doesn't resolve your issue, you can remove the `sbx` state directory entirely. This deletes all sandbox data, configuration, and cached images. Stop all running sandboxes first with `sbx reset`.

{{< tabs >}} {{< tab name="macOS" >}}

```
$ rm -rf ~/Library/Application\ Support/com.docker.sandboxes/
```

{{< /tab >}} {{< tab name="Windows" >}}

```
> Remove-Item -Recurse -Force "$env:LOCALAPPDATA\DockerSandboxes"
```

{{< /tab >}} {{< tab name="Linux" >}}

Sandbox state on Linux follows the XDG Base Directory specification and is spread across three directories:

```
$ rm -rf ~/.local/state/sandboxes/
$ rm -rf ~/.cache/sandboxes/
$ rm -rf ~/.config/sandboxes/
```

If you have set custom `XDG_STATE_HOME`, `XDG_CACHE_HOME`, or `XDG_CONFIG_HOME` environment variables, replace `~/.local/state`, `~/.cache`, and `~/.config` with the corresponding values.

{{< /tab >}} {{< /tabs >}}

Enable automatic diagnostics uploads

To opt in to automatic diagnostics uploads after certain daemon errors, set `diagnostics.autoUpload` to `yes`:

```
$ sbx settings set diagnostics.autoUpload yes
```

Automatic bundles include basic system information and client, daemon, crash, and MCP logs. Docker Sandboxes redacts recognized identity values and credential patterns, but collected logs can still contain user content. Failed uploads remain in a local queue for a later retry.

Report an issue

If you've exhausted the steps above and the problem persists, file a GitHub issue at github.com/docker/sbx-releases/issues.

To help Docker investigate, generate a diagnostics bundle and share it when reporting the issue:

```
$ sbx diagnose --upload
```

The bundle contains daemon logs, diagnostic check results, and basic system information. When `--upload` is confirmed, the bundle is uploaded to Docker support and the command prints a diagnostics ID. Include this ID in your issue so the team can correlate it with the uploaded bundle.

Without terminal input, `sbx diagnose --upload` saves the bundle locally and skips the upload. Check for a diagnostics ID to confirm that the upload succeeded.

□□□ #2
□ □□□ □□□ 2026-10-31 19:02:19 UTC
□ □□□ □□□ 2026-10-31 19:02:21 UTC