Sandbox environment files

- Start an environment
  - Protect the environment file
- Commands
  - Set user defaults
  - Reference directories
    - Project directory
    - File directory
  - Parameterize an environment
- Common workflows
  - Add v3 kits
  - Combine team defaults and personal settings
  - Work across multiple repositories
  - Reuse an environment in automation
- Use a cloud environment
- Review an environment plan
  - Approve or skip host commands
- Update an environment
- Remove an environment
  - Clean up after a failed create
- File reference
  - Top-level fields
  - args
  - kits
  - workspace
  - additionalWorkspaces
  - mounts
  - sandboxOptions
  - lifecycle

- secrets
- bindings
- registries
- mcp
  - Approve local MCP commands
  - Recover a conflicting MCP registration
- ports

A sandbox environment file captures the setup for a local or cloud sandbox in a `sbxenv.yaml` file. Share the file with project contributors so they use the same agent, tools, resources, and credentials without reproducing CLI flags and setup steps.

Kits package software and declare what it needs to run. An environment file selects those kits and configures the sandbox that uses them, including its workspace, resources, credentials, and lifecycle commands. You can change or share that configuration without repackaging the tools. See Kits for how kits supply agents, tools, and their runtime requirements.

> `sbx env` is experimental. The command interface and file format may change.

The examples on this page use local sandboxes unless stated otherwise. For cloud configuration and lifecycle differences, see Use a cloud environment.

Start an environment

Keep the environment file outside the directories you mount into the sandbox. That includes the primary workspace and every `additionalWorkspaces` mount. For example, place it beside your project:

```
web-app-env/
├── sbxenv.yaml
└── web-app/
```

Create `web-app-env/sbxenv.yaml`. This example gives the agent a shared environment variable and the Playwright browser-testing tools. It also publishes the application's development port:

```
schemaVersion: "1"
name: web-app
```

```
agent: claude
workspace: ./web-app

kits:
  - docker.io/sbx/playwright-kit:latest

env:
  NODE_ENV: test

ports:
  - sandbox: 3000
    host: 3000
```

> This example uses kits v2, the format used by built-in agents. For v3, see Add v3 kits.

From `web-app-env`, run the environment:

```
$ sbx env run
```

`sbx` shows an environment plan and asks you to approve it. If you approve the plan, the `web-app` directory becomes the workspace, while `sbxenv.yaml` remains outside the sandbox. If the environment doesn't exist, `sbx` creates a sandbox named `web-app`, installs Playwright and Chromium, and publishes sandbox port `3000` on the host. It then attaches to the agent. Later runs attach to the existing sandbox.

Protect the environment file

Environment files can declare lifecycle, secret, and MCP commands that run on your host. Keep them outside writable sandbox mounts so the agent cannot change what a later invocation runs. This applies to both the primary workspace and `additionalWorkspaces`.

If you need to keep an environment file in a direct-mounted workspace, place it directly in the workspace root. `sbx` mounts the file read-only by default, but a file in a subdirectory can still be replaced by renaming its parent directory.

For a local environment where you intend the agent to edit the file, set `sandboxOptions.writableEnvFiles: true`. The plan reports that the file is writable. Use this option only when you trust the agent to change commands that future invocations run on your host.

Commands

Command	Description
`sbx env plan [PATH...]`	Shows what applying the environment would change without changing or approving it
`sbx env run` `[PATH...]`	Applies the approved plan, creates the environment if needed, and attaches
`sbx env create` `[PATH...]`	Applies the approved plan and creates the environment without attaching
`sbx env exec` `[PATH...] -- COMMAND [ARG...]`	Runs a command in an existing environment without running lifecycle commands
`sbx env rm` `[PATH...]`	Shows a destroy plan, then removes the sandbox and resources named in the plan

To use an environment file, pass its path to `sbx env`. If you pass a directory, `sbx` looks for `sbxenv.yaml` inside it. If you don't pass a path, `sbx` looks in the directory you run the command from and loads your user defaults, if present.

Every `sbx env` subcommand accepts `--name`. This flag sets the sandbox name for that command, overriding the `name` field in the file or the automatically generated name:

```
$ sbx env create --name web-app-test
$ sbx env exec --name web-app-test -- npm test
$ sbx env rm --name web-app-test
```

Use the same file paths for each command. If you set `--name`, use that same name for every command that manages the sandbox.

Set user defaults

Create `~/.sbxenv.yaml` to share settings across your projects. For example, this file selects Claude as the agent and mounts the directory you run `sbx env` from:

```
schemaVersion: "1"
agent: claude
workspace: ${{ env.projectDir }}
```

With this file saved in your home directory, run:

```
$ cd /projects/web-app
$ sbx env run
```

The sandbox mounts `/projects/web-app` as its workspace. Run the same command from `/projects/api`, and it mounts `/projects/api` instead. You don't need a separate `sbxenv.yaml` in either project.

If the project has a `sbxenv.yaml`, `sbx` combines it with your user defaults. Project settings override individual default values. Lists such as `ports` and `mcp.servers` combine entries from both files. If you pass a file or directory path to the command, `sbx` skips `~/.sbxenv.yaml`.

In `~/.sbxenv.yaml`, you can set `workspace` to `${{ env.projectDir }}` or a subdirectory such as `${{ env.projectDir }}/src`. Other workspace paths aren't accepted in this file.

The user defaults file cannot set `name`. Set the sandbox name in a project environment file or with `--name`.

Reference directories

You can use `${{ env.projectDir }}` and `${{ env.fileDir }}` in environment files to insert absolute directory paths. They refer to directories on the host.

Project directory

`${{ env.projectDir }}` is the absolute path to your project directory. `sbx` chooses this directory from the command you run:

- `sbx env run`: the directory you run the command from.
- `sbx env run /projects/web-app`: `/projects/web-app`.
- `sbx env run /projects/web-app/custom.yaml`: `/projects/web-app`, the directory containing the file.

If you pass several paths, the first one sets the project directory. Every file loaded by that command uses the same value for `env.projectDir`.

Use this reference in shared settings that need to point to each project's files. For example, `workspace: ${{ env.projectDir }}/src` mounts the `src` directory in whichever project you select.

File directory

`${{ env.fileDir }}` is the absolute path to the directory containing the environment file where you write the reference. For example, inside `/shared/environment.yaml`, its value is `/shared`.

Use this reference when an environment file needs to locate files stored alongside it. For example, suppose your setup script is `/shared/scripts/setup.sh`. Add this lifecycle command to `/shared/environment.yaml` to run the script from `/shared`:

```
lifecycle:
  initialize:
    - command: ./scripts/setup.sh
      workdir: ${{ env.fileDir }}
```

The command runs from `/shared`, even if you use this environment file with a project in another directory.

Relative workspace paths already use the directory containing the environment file. For example, `workspace: ./src` in `/shared/environment.yaml` mounts `/shared/src`.

Both directory references can appear in YAML values, but not in field names or inside the `args` block.

Parameterize an environment

Declare inputs in a top-level `args` block when values need to vary between uses of the same environment file. Each argument must have exactly one of `default` or `required: true`:

```
schemaVersion: "1"
name: web-app
agent: claude

args:
  channel:
    default: stable
    description: Release channel
    enum:
      - stable
      - beta
  endpoint:
    required: true
    description: API endpoint
  cpus:
    default: "4"
    pattern: "[1-9][0-9]*"
```

```
env:
  RELEASE_CHANNEL: ${{ env.args.channel }}
  API_ENDPOINT: ${{ env.args.endpoint }}

sandboxOptions:
  cpus: ${{ env.args.cpus }}
```

Reference a declared argument as `${{ env.args.NAME }}` anywhere a YAML value can appear. References can't be used in field names or within the `args` block. An unquoted reference is interpreted as a YAML value after substitution, so the `cpus` value in this example becomes an integer. Quote a reference to preserve it as a string.

All `sbx env` commands accept repeatable `--env-arg NAME=VALUE` flags. Values provided with a flag replace defaults from the environment file:

```
$ sbx env run --env-arg endpoint=https://api.example.com --env-arg channel=beta
```

Use `--env-args-file` to load values from a file. Each non-empty, non-comment line must have the form `NAME=VALUE`:

```
# production.args
channel=beta
endpoint=https://api.example.com
```

```
$ sbx env run --env-args-file production.args
```

You can pass multiple argument files. Later files take precedence over earlier files, and `--env-arg` flags take precedence over every argument file. Values can contain `=`, and values in an argument file are read literally rather than expanded by a shell.

Argument references and the two directory references are the only variable expressions expanded in an environment file. Shell-style expressions such as `${VAR}` aren't expanded from the host environment. Other dollar signs remain literal, so a value such as `$PATH:/opt/bin` is passed unchanged. Use `$${{ env.args.NAME }}` to produce the literal text `${{ env.args.NAME }}`. Substituted values aren't expanded a second time.

Common workflows

The following examples combine environment file fields into configurations you can adapt for a project.

Add v3 kits

To add v3 mixins, include a v3 workload in `kits` and set `agent` to the workload's name.

For example, save this as `web-app-env/v3.sbxenv.yaml` to use a v3 Claude Code workload with the v3 Playwright mixin:

```
schemaVersion: "1"
name: web-app-v3
agent: sbx-kit-claude
workspace: ./web-app

kits:
  - docker.io/docker/sbx-kit-claude:2.1.267
  - docker.io/sbx/playwright:1.61.1
```

From `web-app-env`, run:

```
$ sbx env run v3.sbxenv.yaml
```

The `agent` value, `sbx-kit-claude`, matches the last component of the workload's repository name. The `web-app-v3` name creates a separate sandbox. For other agents, see the published v3 workloads.

Combine team defaults and personal settings

Keep the shared configuration in a version-controlled environment directory outside the mounted workspace. Put machine-specific settings in a file excluded from version control. For example, commit `base.sbxenv.yaml` beside the `web-app` directory:

```
schemaVersion: "1"
name: web-app
agent: claude
workspace: ./web-app

env:
  NODE_ENV: development
```

```
sandboxOptions:
  cpus: 4
  memory: 8g
```

Add `local.sbxenv.yaml` to `.gitignore`, then use it for personal settings:

```
env:
  LOG_LEVEL: debug

sandboxOptions:
  memory: 12g
```

Pass both files in merge order:

```
$ sbx env run base.sbxenv.yaml local.sbxenv.yaml
```

Nested mappings merge by key, lists concatenate, and values from later files replace earlier scalar values. In this example, the sandbox has four CPUs, 12 GB of memory, and both environment variables. Each relative workspace path resolves from the directory of the file that declares it.

Each source file must contain valid fields and values before merging. A later override can't repair an invalid value in an earlier file. This also applies to user defaults loaded from `~/.sbxenv.yaml`.

Work across multiple repositories

Mount related repositories alongside the primary project when the agent needs to coordinate changes or consult shared code and documentation:

```
# sbxenv.yaml in the directory above the repositories
schemaVersion: "1"
name: web-platform
agent: codex

workspace: ./web-app

additionalWorkspaces:
  - path: ./shared-components
  - path: ./architecture-docs
    readOnly: true
```

The agent starts in `web-app`, can modify `shared-components`, and can read `architecture-docs` without changing it. The environment file stays outside all three workspaces. Relative paths resolve from the directory of the environment file that declares them. Additional workspaces are mounted directly even when the primary workspace uses clone mode.

Reuse an environment in automation

Use the same committed environment for interactive development and automated tasks. Developers attach to the agent with `run`:

```
$ sbx env run
```

Automation can create the sandbox without attaching, run commands in it, and remove it afterward:

```
$ sbx env create --force
$ sbx env exec -- npm test
$ sbx env rm --force
```

For `create` and `run`, `--force` approves the plan for that invocation without recording consent for later invocations. Use the flag for each unattended `create` or `run`. For `rm`, `--force` approves the destroy plan and removes the sandbox even when it is in use.

Commands and vault references under `secrets` resolve on the host, so the automation runner must provide the referenced tools and authentication. The secret values remain outside the environment file.

Use a cloud environment

Use `sbx --cloud env` to create a sandbox on Docker-managed infrastructure from an environment file. Cloud sandboxes don't mount or synchronize local project files, and local credentials aren't available to them. You copy files into the sandbox or clone a repository there, then retrieve your work before removing it.

For a walkthrough, supported settings, and lifecycle differences, see Use environment files in the cloud.

Review an environment plan

`sbx env create`, `sbx env run`, and `sbx env rm` show the changes an environment makes outside its sandbox and ask for approval before applying them. The plan includes host commands, credentials, bindings, MCP registrations, directories, kits, published ports, sandbox options, and environment variables.

Run `sbx env plan` to inspect the apply plan without changing, approving, or recording anything:

```
$ sbx env plan
```

The plan compares the environment file with the environment's last applied state and the resources on the host. It omits resources that are unchanged and already approved. Literal secret values appear as SHA-256 digests. Secret references, host commands, environment variables, ports, paths, and binding domains remain visible so you can review them.

Changes appear under the same keys and nesting as the environment file. For example, this plan changes `GOFLAGS` from `-mod=mod` to `-mod=readonly`:

```
   env:
~    GOFLAGS: -mod=mod -> -mod=readonly
```

Read the symbol beside each entry before approving the plan:

Symbol	Effect of applying the entry
`+`	Add a resource
`~`	Change a resource
`-`	Destroy a resource
`>`	Run a host command again
`!`	Stop tracking a resource that is no longer declared, without deleting it

Interactive approval is recorded for the environment under the `sbx` state directory. Plans without host commands apply silently on later invocations until the environment changes or a resource is missing. An approval provided with `--force` applies only to that invocation.

Approve or skip host commands

Plans containing lifecycle commands, credential `command` sources, or local MCP commands require approval for every invocation by default, even when the command text hasn't changed. Approve one invocation with `--force`, skip lifecycle and MCP commands with `--skip-host-commands`, or turn on `env.rememberHostCommands` to remember approval until the commands change:

```
$ sbx settings set env.rememberHostCommands true
```

To run an environment without lifecycle or local MCP commands, use:

```
$ sbx env run --skip-host-commands
```

The flag also disables previously approved MCP commands before asking you to approve the plan. They remain disabled even if you cancel the plan or restart the daemon. To enable them again, run `sbx env run` without the flag and approve the plan. Credential `command` sources still run when their secrets are needed.

Update an environment

For an existing sandbox, `sbx env run` applies updated `env` values to the new agent session and reconciles declared MCP servers. Changes to workspaces, kit host mounts, kits, ports, secrets, bindings, and `sandboxOptions` take effect only when the sandbox is next created. Remove the environment with `sbx env rm`, then create it again to apply those changes.

Remove an environment

Secrets and registry credentials are sandbox-scoped. Credential bindings and remote MCP server registrations are host-global and can be shared by multiple sandboxes. Local MCP commands declared by the environment belong to that environment and sandbox.

`sbx env rm` builds a destroy plan from the resources on the host. The plan includes all credentials stored at the sandbox's scope, including credentials that the environment file no longer declares. After approval, `sbx` removes only the resources named in the plan.

Global credential bindings remain unless you pass `--prune-bindings`. Remote MCP registrations remain available to other sandboxes. Removing a local environment revokes and removes the MCP commands it registered.

Clean up after a failed create

Secret provisioning, binding updates, and MCP server registration occur before the sandbox is created. If sandbox creation fails, scoped secrets remain, and bindings and MCP registrations may also remain. Run `sbx env rm` with the same paths to remove the scoped secrets. Pass `--prune-bindings` if you also want to remove the declared global bindings. Remote MCP registrations remain after cleanup. If an environment MCP command registration cannot be removed, follow Recover a conflicting MCP registration.

File reference

Top-level fields

Field	Type	Required	Default	Description
`schemaVersion`	string	Yes	None	Schema version. The supported value is `"1"`
`name`	string	No	`<agent>-<workspace-basename>`	Sandbox name, overridden by `--name`
`agent`	string	Yes	None	Built-in agent or the name of an agent kit
`args`	map	No	None	Environment arguments. See `args`
`kits`	list	No	None	Kits to install at creation. See `kits`
`workspace`	string or object	No	No host mount	Primary workspace. See `workspace`
`additionalWorkspaces`	list	No	None	Extra directories to mount. See `additionalWorkspaces`
`mounts`	map of strings	No	None	Host directories for kit-declared mounts. See `mounts`
`env`	map of strings	No	None	Environment variables for the sandbox
`sandboxOptions`	object	No	None	Creation options. See `sandboxOptions`
`secrets`	map	No	None	Service credentials. See `secrets`
`bindings`	map	No	None	Credential injection approvals. See `bindings`
`registries`	map	No	None	Registry pull credentials. See `registries`
`mcp`	object	No	None	MCP servers. See `mcp`

Field	Type	Required	Default	Description
`ports`	list	No	None	Port mappings. See `ports`
`lifecycle`	object	No	None	Host commands. See `lifecycle`

args

`args` maps argument names to their declarations. Names must start with a letter or underscore and can contain letters, numbers, underscores, and hyphens. Each declaration must set exactly one of `default` or `required: true`.

Field	Type	Default	Description
`default`	string	None	Value used when the command doesn't supply the argument
`required`	boolean	`false`	Require the command to supply the argument
`description`	string	None	Explanation shown in command output
`enum`	list of strings	None	Values accepted for the argument
`pattern`	string	None	Go (`RE2`) expression matched against the complete argument value

`enum` and `pattern` can't be used together.

kits

`kits` accepts local directories, ZIP archives, OCI registry references, and Git URLs prefixed with `git+https://` or `git+ssh://`. Kits can install tools, configure the sandbox, and give the agent project-specific instructions. Built-in agents use v2 mixins. For v3 workloads and mixins, see Add v3 kits.

An environment file selects kits and configures a sandbox's host resources. A kit descriptor defines the package itself. The environment file's `schemaVersion` is independent of the schema version in a kit descriptor.

Explicit relative paths resolve from the directory of the environment file that declares them. These include `.`, `..`, paths that start with `./` or `../`, and relative paths that end in `.zip`. Bare references such as `organization/kit` remain registry references.

Use an object entry to pass arguments to a kit. Set `source` to the kit reference and map the kit's argument names to values under `args`:

```
kits:
  - source: ./kits/tool
    args:
      version: ${{ env.args.channel }}
```

Remote kit sources must match the `kit.allowedSources` setting. Docker Hub is allowed by default. To use Git kits from `docker/sbx-kits-contrib`, add its source:

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

The setting replaces the complete allowlist, so include any existing sources you want to keep. For reproducible setup, pin Git kits with the `ref` URL parameter and OCI kits with an immutable tag or digest.

`workspace`

When specified as a string, `workspace` is the path. Use the object form for clone mode. Omit `workspace` to create a sandbox without a host bind mount. Set `workspace: .` to mount the directory that contains the environment file that declares it.

For environment file placement and write protection, see Protect the environment file.

Field	Type	Required	Default	Description
`path`	string	Yes	None	Workspace directory. Relative paths resolve from the declaring file's directory
`clone`	boolean	No	`false`	Use a private clone, equivalent to `sbx create --clone`

You can override `workspace.clone` for one `create` or `run` invocation with `--clone` or `--clone=false`.

`additionalWorkspaces`

Each additional workspace is mounted after the primary workspace. Relative paths resolve from the directory of the environment file that declares them.

Field	Type	Required	Default	Description

`path`	string	Yes	None	Directory to mount
`readOnly`	boolean	No	`false`	Mount the directory read-only

`mounts`

`mounts` selects host directories for kit-declared host mounts in local sandboxes. Map each kit-declared sandbox path to an existing host directory. Relative host paths resolve from the environment file that declares them:

```
mounts:
  /home/agent/history: ./history
```

The target must match a `host-mount@1` path declared by one of the selected kits. Bindings take effect when the sandbox is created.

`sandboxOptions`

Field	Type	Default	Description
`template`	string	None	Custom sandbox template image
`memory`	string	None	Memory limit, such as `8g` or `512m`
`cpus`	integer	`0`	Number of CPUs. `0` selects the host default
`pullPolicy`	string	`always`	Image pull policy: `always`, `missing`, or `never`
`profile`	string	None	Governance profile name
`skills`	string	Daemon default	Shared agent skills store access: `off`, `readonly`, or `readwrite`
`display`	boolean	`false`	Provision a display socket for graphical applications
`gpu`	boolean	`false`	Pass the host GPU through to the sandbox
`usb`	list of strings	None	USB device selectors to pass through to the sandbox
`writableEnvFiles`	boolean	`false`	Let the sandbox write to shared environment files; local only

For local sandboxes, `cpus: 0` allocates all host CPUs, except on Linux arm64 hosts, where the default is capped at 16. Set `cpus` to an explicit count to request a larger allocation, up to the number of available host CPUs. When creating a sandbox directly with `sbx create` or `sbx run`, use `--cpus` for the same override. Cloud resource limits must match a cloud size.

`skills` controls access to the shared agent skills store. Set it to `off` to omit the mount, `readonly` to mount the store read-only, or `readwrite` to let the sandbox modify shared skills. If omitted, it uses the daemon's default, which is `readonly` unless your organization overrides it.

`lifecycle`

The `lifecycle` block declares commands that run on the host with your user privileges. Use lifecycle commands for work that must happen outside the sandbox, such as creating a workspace, seeding fixtures, or archiving state.

```
lifecycle:
  initialize:
    - name: Prepare workspace
      command: test -d web-app || git clone https://github.com/example/web-app
      timeout: 5m
  postCreate:
    - command: ./scripts/seed-fixtures.sh
      workdir: web-app
  preRemove:
    - command: ./scripts/archive-state.sh
```

Lifecycle phases run at the following points:

Phase	Timing
`initialize`	Before other create or run actions. Runs for every `create` and `run`
`postCreate`	After a new sandbox is created. For `run`, before attachment. Does not run when attaching to an existing sandbox
`preRemove`	After you approve removal and before `sbx` deletes resources. A failure produces a warning and removal continues

`sbx env exec` doesn't run lifecycle commands. Commands within a phase run in order and stop at the first failure. Make `initialize` commands safe to run more than once.

After `preRemove` commands finish, `sbx` generates the destroy plan again. Removal stops if the commands introduced changes that weren't included in the approved plan.

Each command requires `command` and accepts the following fields:

Field	Type	Default	Description
`name`	string	Command text	Label shown in progress and plan output
`command`	string	None	Command passed to the user's shell
`workdir`	string	Project directory	Host working directory. Relative paths resolve from the first file's directory
`timeout`	string	None	Maximum runtime, such as `90s` or `5m`

Commands inherit the environment of the `sbx` process and receive the following variables:

Variable	Value
`SBX_LIFECYCLE_PHASE`	Phase running the command: `initialize`, `postCreate`, or `preRemove`
`SBX_ENV_FILE`	Absolute path to the last environment file in merge order
`SBX_ENV_FILES`	Absolute paths to all environment files in merge order, separated by `:` on macOS and Linux or `;` on Windows
`SBX_ENV_DIR`	Project directory, used as the default host working directory
`SBX_SANDBOX_NAME`	Resolved sandbox name, including any `--name` override
`SBX_AGENT`	Resolved agent name
`SBX_WORKSPACE`	Primary workspace path on the host. Empty when no workspace is declared, including cloud environments

For cloud environments, `sbx` commands run by lifecycle commands inherit cloud mode and the selected endpoint. For example, `sbx cp` in a lifecycle command targets the cloud even without `--cloud`.

The environment file's `env` values and resolved secrets aren't passed to host commands.

`secrets`

`secrets` maps service names to secret sources. Each entry must set exactly one of `value`, `ref`, or `command`. The secret is stored at the sandbox scope when the environment is created.

Field	Type	Default	Description
`value`	string	None	Literal secret value

Field	Type	Default	Description
`ref`	string	None	Vault URI, such as `op://Vault/Item/field`
`command`	string	None	Host shell command whose standard output becomes the secret
`snapshot`	boolean	`false`	Resolve `ref` or `command` once on the host and store the result as a literal
`refresh`	string	None	Resolution policy for `ref` or `command`, such as `on-demand` or `55m`
`backend`	string	Automatic	Resolver for `ref`: `sdk` or `cli`
`noVerify`	boolean	`false`	Skip verifying that a `ref` or `command` resolves during provisioning

> A literal `value` is visible to anyone with read access to the file. Use a vault URI with `ref` or obtain the value at runtime with `command`.

```
secrets:
  anthropic:
    ref: op://Private/Anthropic/api-key
    refresh: 55m
  github:
    command: gh auth token
```

Secret commands execute from a fresh temporary directory on the host. Relative helper paths resolve from that directory. Keep helpers and their dependencies outside writable sandbox mounts. Use an absolute path, an absolute host `PATH` entry, or an explicit change to the helper's private directory. See dynamic secret sources.

For a cloud environment, set `snapshot: true` on a `ref` or `command` source:

```
secrets:
  github:
    command: gh auth token
    snapshot: true
```

The command runs on the host after plan approval. The resolved value is stored as a literal secret and doesn't refresh. Recreate the environment to rotate it. Snapshots also work with local

environments. A snapshot can't set `refresh` or `noVerify`. Cloud snapshots use CLI resolvers for vault references and don't support `backend: sdk`.

bindings

`bindings` approves credential injection domains for each service. The environment merges these approvals into the user's global `credentials.yaml`. Each service can contain an `apiKey` block, an `oauth` block, or both. Each block contains a `domains` list:

```
bindings:
  github:
    apiKey:
      domains:
        - api.github.com
```

`sbx env rm` preserves global bindings by default. Pass `--prune-bindings` to remove every service binding declared by the environment file.

> `--prune-bindings` deletes the complete global binding entry for every service declared in the environment file. This can affect other sandboxes that share those service bindings.

registries

`registries` maps registry hostnames to pull credentials. Each entry requires `secret` and accepts an optional `username`. Both fields accept a secret source with exactly one of `value`, `ref`, or `command`.

When `username` is omitted, `sbx` stores a token-only credential. Registries such as GHCR and GitLab accept token-only credentials.

```
registries:
  ghcr.io:
    secret:
      command: gh auth token
```

mcp

The `mcp.servers` list registers servers with the built-in MCP gateway and adds them to the sandbox. Remote servers registered with `url` remain host-global and survive `sbx env rm`. In local

environments, servers registered with `command` belong to the environment and can run only for its sandbox.

Field	Type	Required	Default	Description
`name`	string	Yes	None	Server name
`url`	string	No	None	Remote server URL, registry reference, or OCI reference
`command`	string	No	None	Command for a local stdio server
`args`	list of strings	No	None	Arguments passed to `command`

Each server must set exactly one of `url` or `command`.

Approve local MCP commands

A `command` server runs on your host with your privileges whenever the sandbox's MCP gateway starts, including after restarts. The environment plan shows the resolved executable and arguments before registration. Approval pins the executable path, not the contents of executables, scripts, or packages they load. Keep these files outside writable sandbox mounts.

If an `initialize` command creates the executable or changes its resolved path, `sbx` asks you to approve the resolved MCP command after initialization. Declining that approval leaves the completed initialization work in place but does not register the command. `--force` approves both stages for that invocation.

Removing a command from the file and applying the environment revokes its registration. `sbx env rm` removes the environment's command registrations.

Recover a conflicting MCP registration

An environment cannot take over a command registered manually or by another environment, even when its executable and arguments match. Choose a different server name, or remove the conflicting registration before applying the file.

For a legacy or damaged registration, stop the daemon before removing only its saved registration:

```
$ sbx daemon stop
$ sbx mcp rm --registration-only -- <SERVER_NAME>
```

Stopping the daemon interrupts its sandbox gateways. Registration-only removal does not delete saved credentials. Apply the environment again to register and approve the command for its sandbox.

`ports`

`ports` publishes sandbox ports when the environment is created. Ports exposed by a kit but omitted from this list receive an ephemeral host port.

Field	Type	Required	Default	Description
`sandbox`	integer	Yes	None	Sandbox port from 1 through 65535
`host`	integer	No	Ephemeral	Host port from 1 through 65535
`protocol`	string	No	`tcp4`, or `tcp6` for IPv6 `hostIP`	`tcp`, `tcp4`, `tcp6`, `udp`, `udp4`, or `udp6`
`hostIP`	string	No	Loopback	Host interface to bind

Set `protocol: tcp` to bind both IPv4 and IPv6. Leave `hostIP` unset for a dual-stack binding because an explicit address binds only its own IP family.

If a port can't be published, sandbox creation fails and removes the new sandbox.

□□□ #2
□ □□□ □□□ 2026-10-31 18:58:34 UTC
□ □□□ □□□ 2026-10-31 18:58:37 UTC