Kits v2

- Use existing kits
  - Restrict kit sources
- Image overrides for built-in agents
  - Choose a template
  - Build a custom template
- Kit kinds
  - kind: mixin
  - kind: sandbox
- Top-level fields
- Arguments
  - Pass arguments to kits
- Sandbox block
- Agent instructions
- Credentials
  - apiKey
  - oauth
- Network
- Ports
- Environment
- Setup
  - Execution order
  - install
  - startup
  - files
  - Shell initialization and service logs
- Static files
- Volumes
- Fork an existing agent

- Install an internal CA certificate
- Sandbox-managed agent configuration
- Validate and inspect kits
- Packaging and distribution
- Sign and verify kits
  - Inspect provenance
  - Require signed kits
- Schema versions
- Move an environment to v3

{{< summary-bar feature_name="Docker Sandboxes sbx" >}}

V2 kits remain supported and are not deprecated. Built-in agent names such as `claude` and `codex` select v2 kits and work with v2 mixins. This page covers their usage and specification. For new kits, Docker recommends v3. V1 also remains supported.

Use existing kits

A v2 kit contains `spec.yaml` with `schemaVersion: "2"` and an optional `files/` tree. A sandbox kit defines the agent environment. A mixin adds tools or configuration to it. Pass a sandbox kit in place of the agent name and add mixins with `--kit`:

```
$ sbx run ./my-agent --name my-project --kit ./team-config
$ sbx run claude --name claude-project --kit ./team-config
```

`sbx run` uses your current directory as the workspace. Append a project path to use another directory. To create without launching the agent, use `sbx create`; include a workspace path or `.` to mount a directory. References can be local directories, ZIP files, OCI artifacts, or Git URLs. Start relative paths with `./` or `../`. For Docker Hub kits, you can omit `docker.io/` and use `<NAMESPACE>/<KIT>:<TAG>`. In Git URLs, `ref` selects a revision and `dir` the kit directory. Quote URLs containing `&`:

```
$ sbx run "git+https://github.com/<ORG>/<REPOSITORY>.git#ref=<COMMIT>&dir=my-agent"
```

`git+ssh://` URLs work with your local SSH agent and Git credentials. For private registries, see Registry credentials.

Kit selection with `--kit` applies at creation. Recreate the sandbox to change its kit set, except for the limited updates supported by `sbx kit add`. That command restarts the sandbox while preserving packages, images, volumes, and agent history. Kits can't be removed from a running sandbox.

Restrict kit sources

See Restrict kit sources for source policies. `kit.allowLocalKits` also governs v2 ZIP files.

Image overrides for built-in agents

Use `--template` to replace a built-in agent's image while keeping its configuration and launch command. The replacement image must support the same agent. For example, an image used with `claude` must have Claude Code installed.

To define an environment with its own launch command and sandbox settings, see Build an agent workload for the v3 workflow.

Choose a template

Docker publishes agent images as `docker/sandbox-templates:<variant>`. Choose the variant that matches your agent. See Base images for the available variants.

Variants with a `-docker` suffix, such as `claude-code-docker`, include Docker Engine for building and running containers inside the sandbox. Built-in agents use these variants by default when you don't specify a custom template.

If you don't need Docker inside the sandbox, select a variant without the suffix. It uses fewer resources and doesn't require privileged mode:

```
$ sbx run claude --template docker.io/docker/sandbox-templates:claude-code
```

Include the registry domain in `--template` image references. Unlike kit references, template references don't automatically expand to include `docker.io`.

Build a custom template

Building a custom template requires Docker Desktop.

Extend the Docker-provided image for the agent you plan to run. For example, this Dockerfile adds Rust and protocol buffer tools to the Claude Code image:

```
FROM docker/sandbox-templates:claude-code
USER root
RUN apt-get update && apt-get install -y protobuf-compiler
USER agent
RUN curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh -s -- -y
```

Install system packages as `root`, then switch back to `agent` before installing tools in the agent's home directory.

Build the image and push it to a registry. Replace `<NAMESPACE>` with a Docker Hub namespace you can push to:

```
$ docker build -t docker.io/<NAMESPACE>/my-template:v1 --push .
```

For registry credentials and loading a locally built image, see Load a template.

Run the sandbox with your image:

```
$ sbx run claude --template docker.io/<NAMESPACE>/my-template:v1
```

Because this image extends `claude-code`, use it with `claude`. For an image based on `codex`, use `codex`. For one based on `shell`, use `shell` to open Bash without an agent.

If your added tools need network access, allow the domains they use in the sandbox's network policy, unless you use the `allow-all` policy:

```
$ sbx policy allow network "*.example.com:443,example.com:443"
```

Kit kinds

`kind: mixin`

A mixin layers capabilities onto an existing sandbox. It must not declare a `sandbox:` block, `extends:`, or `mixins:`. A mixin can declare `requires:` to pin the base agent it is designed for:

```
schemaVersion: "2"
kind: mixin
```

```
name: github-tools
requires:
  agent: claude
```

`requires.agent` takes one base-agent name. It is validated as a kit name and enforced during composition.

`kind: sandbox`

A sandbox kit defines a full agent. A root sandbox must declare a `sandbox:` block. A sandbox that uses `extends:` can inherit the parent image and omit its own `sandbox:` block:

```
schemaVersion: "2"
kind: sandbox
name: claude-safe
extends: claude
```

`extends:` is sandbox-only. The parent must resolve to a sandbox kit. `mixins:` is also sandbox-only and accepted by the parser, but runtime composition support is pending.

Top-level fields

For the normative grammar, see the v2 specification.

Field	Required	Description
`schemaVersion`	Yes	Spec schema version. Use `"2"` for this grammar.
`kind`	Yes	`mixin` for kits that extend an agent; `sandbox` for kits that define one.
`name`	Yes	Unique identifier. Lowercase alphanumeric with hyphens, 1 to 64 characters.
`version`	No	Kit version.
`displayName`	No	Human-readable name.
`description`	No	Short description.
`sourceURL`	No	Source repository or documentation URL.
`licenses`	No	SPDX license identifiers.

Field	Required	Description
`locked`	No	Dotted paths child kits may not override.
`security`	No	Container security settings. `security.privileged: true` runs the container in privileged mode.
`args`	No	Arguments supplied when the kit is loaded. Schema v2 only.

A kit also declares behavior blocks such as `agentInstructions`, `permissions`, `ports`, `credentials`, `environment`, `setup`, and `volumes`.

Arguments

A schema v2 kit can declare arguments and reference them anywhere in `spec.yaml` or under `files/` as `${{ kit.args.<name> }}`. Substitution happens before the spec is decoded.

```
args:
  version:
    default: latest
    description: Tool version to install
    pattern: '^(latest|[0-9]+\.[0-9]+\.[0-9]+)$'
  channel:
    default: stable
    enum: [stable, beta, nightly]
  target:
    required: true
    description: Build target

environment:
  variables:
    TOOL_VERSION: "${{ kit.args.version }}"
```

Don't use kit arguments for API tokens, passwords, or other secrets. Use Credentials to provide sensitive values to a sandbox.

Field	Description
Argument name	Starts with a letter or underscore and contains only letters, digits, underscores, and hyphens.

Field	Description
`default`	String to use when the caller supplies no value. Mutually exclusive with `required: true`.
`required`	Set to `true` when the caller must supply a value. Mutually exclusive with `default`.
`description`	Optional help text shown when a required value is missing.
`enum`	Optional list of accepted values. Mutually exclusive with `pattern`.
`pattern`	Optional Go RE2 regular expression matched against the complete value. Mutually exclusive with `enum`.

Each argument must declare either `default`, including an empty-string default, or `required: true`.

Argument values are strings, but substitution happens before YAML decoding. Quote a placeholder in a string-valued field so a value such as `1.20` isn't decoded as a number.

Pass arguments to kits

Use `--kit-arg name=value` for every kit declaring that argument, or prefix with the kit's `name` to target one kit. Scoped values override shared values:

```
$ sbx run ./my-agent --kit ./my-mixin --kit-arg channel=stable \
    --kit-arg my-mixin.channel=beta
```

`--kit-args-file <FILE>` reads `name=value` entries, ignoring blank lines and `#` comments. Later files override earlier files; `--kit-arg` overrides files. For repeated CLI keys, the last value wins. Missing required values, unknown arguments, undeclared placeholders, and invalid values fail before creation. Pass the same flags to `sbx kit validate` or `sbx kit inspect` when needed. Argument values can remain in shell history and are stored unencrypted in argument files.

Sandbox block

```
sandbox:
  image: <image-ref>
  build:
    context: .
    dockerfile: Dockerfile
    args:
      AGENT_VERSION: "1.0.0"
```

```
    target: runtime
    platforms:
      - linux/amd64
  entrypoint: [my-agent, "--flag"]
  command:
    default: ["--task-mode"]
    interactive: []
  resources:
    cpu: 2
    memory: 4g
    gpu: "1"
```

Field	Required	Description
`sandbox.image`	When `extends:` is omitted	Docker image reference.
`sandbox.build`	No	Build configuration. Runtime support is pending, so a kit with `build:` must also set `image:`.
`sandbox.entrypoint`	No	Fixed process prefix as a string array. The first element is the agent binary.
`sandbox.command`	No	Mode-specific argument tail. Use a list shorthand for `default`, or a mapping with `default` and `interactive`.
`sandbox.resources`	No	Optional CPU, memory, and GPU constraints. Memory uses byte-size strings such as `4096m` or `4g`.

The effective command is `entrypoint` plus `command.default` for non-interactive launches, and `entrypoint` plus `command.interactive` for TTY sessions. If `interactive` is omitted, it falls back to `default`.

For a kit that uses `extends:`, `sandbox.command` replaces the full inherited argument tail, including flags after the binary in the parent's `sandbox.entrypoint`. It doesn't append to that tail. Define every argument the child needs. For example, a child of `claude` that adds `--settings` must also include `--dangerously-skip-permissions` to preserve that behavior.

The agent's container image must provide:

- A non-root `agent` user at UID 1000 with passwordless sudo.
- A `/home/agent/` home directory owned by `agent`.
- HTTP proxy environment variables (`HTTP_PROXY`, `HTTPS_PROXY`, `NO_PROXY`) preserved across sudo.
- The agent binary, either baked in or installed with `setup.install`.

Build on top of `docker/sandbox-templates:shell-docker` to get these base requirements.

Agent instructions

Declare these fields under `agentInstructions`:

Field	Description
`filename`	AI profile filename. Meaningful for `kind: sandbox`; ignored with a warning for `kind: mixin`.
`content`	Markdown instructions. For a sandbox, inlined into the profile. For a mixin, written to kit memory.

For mixins, the engine writes `content` to `<dir-of-AI-file>/kits-memory/<kit-name>.md` and adds a `## Kits` pointer section to the base AI file. This keeps each mixin's instructions in a separate file.

The generated profile lives in the parent directory of the mounted workspace inside the sandbox. It sits outside the mount and doesn't replace an instruction file in the project. The sandbox kit's inline instructions go directly into that profile.

Credentials

A kit declares the credentials it needs and how the proxy injects them into outbound requests. It does not declare a host discovery source. The user provides the value through the secret store or the first-run prompt, and a credential binding authorizes its use. A kit can't read arbitrary host environment variables or files.

`credentials` is a list; each entry names a `service` and configures one or more auth mechanisms.

Field	Description
`service`	Credential identifier, matched against the value stored with `sbx secret set`. Lowercase kebab-case.
`description`	Optional. Shown to the user when approving a binding.
`required`	Marks the credential as essential to the agent. If it has no binding, `sbx` warns and starts with the credential withheld. Default `false`.
`provider`	Reserved for a provider registry. Accepted with a warning and no runtime effect.
`apiKey`	API-key injection (see apiKey).

Field	Description
`oauth`	OAuth interception (see oauth).

Each service must declare `apiKey`, `oauth`, or both. When both resolve at runtime, the API key takes precedence and OAuth acts as the fallback.

apiKey

Field	Description
`name`	Environment variable name for the credential (for example, `ANTHROPIC_API_KEY`).
`proxyManaged`	If `true`, `sbx` sets `name` inside the container to the `proxy-managed` sentinel. Default `false`.
`inject[].domain`	Domain to inject the credential into. Must also be allowed in `permissions.network`.
`inject[].header`	HTTP header the proxy sets (for example, `x-api-key`, `Authorization`).
`inject[].format`	Header value format, with one `%s` placeholder (for example, `"%s"` or `"Bearer %s"`). Mutually exclusive with `scheme`.
`inject[].scheme`	Shorthand for common auth schemes. `bearer` expands to `Authorization: Bearer %s`; `basic` requires `username`. Mutually exclusive with `format`.
`inject[].username`	Username for HTTP Basic auth, for example `x-access-token` for Git over HTTPS.

oauth

For agents that authenticate with OAuth (for example, Claude Code), the proxy intercepts token responses and replaces real tokens with sentinels, then swaps the real token back in on outbound requests. By default, the token never enters the sandbox. Setting `passthrough: true` opts out of sentinel masking and sends the real token response into the sandbox.

Field	Description
`tokenEndpoint.host` / `path`	The OAuth token endpoint the proxy intercepts.
`sentinels.accessToken` / `refreshToken`	Sentinel values written into the container in place of the real tokens.
`credentialFile.path`	Where to write the credential file inside the container (`~` expands).
`credentialFile.structure`	Declarative JSON shape. Supports `{{.AccessToken}}`, `{{.RefreshToken}}`, `{{.ExpiresAt}}`, and `{{.Scopes}}`.

Field	Description
`credentialFile.template`	Go template. Supports `{{.AccessToken}}`, `{{.RefreshToken}}`, `{{.ExpiresAt}}`, `{{.Scopes}}`, and `{{.ScopesJSON}}`.
`resourceHosts`	API hosts where the proxy attaches the token on outbound requests, distinct from the token endpoint host.
`skipIfEnv`	Accepted for compatibility, but ignored for schema v2. A v2 binding is authoritative instead of host environment variables.
`responseFields`	Overrides the default field names the proxy reads from the token response.
`passthrough`	If `true`, the proxy passes the token response through unchanged instead of replacing the tokens with sentinels.

`credentialFile.structure` provides a declarative alternative to `credentialFile.template`. The engine renders it as well-formed JSON. If both fields are set, `structure` takes precedence.

Network

Network egress is declared under `permissions.network`. Credentials no longer carry their own domain mapping — the proxy injects a credential only into the domains its `apiKey.inject` lists, and every domain the sandbox reaches must be allowed here.

Field	Description
`permissions.network.allow`	Domains the sandbox can reach.
`permissions.network.deny`	Domains the sandbox is blocked from reaching. Deny takes precedence over allow, including across composed kits.

Allow and deny patterns:

Pattern	Example	Status
Exact host	`api.example.com`	Enforced
Exact host and port	`api.example.com:8080`	Enforced
Single-label wildcard	`*.example.com`	Enforced
Multi-label wildcard	`**.example.com`	Parsed; enforcement pending
Port range	`api.example.com:80-443`	Parsed; enforcement pending
Port wildcard	`api.example.com:*`	Parsed; enforcement pending
CIDR	`10.0.0.0/8`	Parsed; enforcement pending

In v1 this was the `network:` block (`allowedDomains` / `deniedDomains`, plus `serviceDomains` / `serviceAuth`). In v2, those fields are decode errors.

Ports

Declare `ports` as a list of entries to expose sandbox services to the host:

Field	Description
`container`	Container port, 1 to 65535.
`protocol`	`tcp` or `udp`. Empty publishes one family; see below.
`name`	Optional label surfaced by tools that list published port bindings.

Host ports are allocated ephemerally. Leave `protocol` empty unless the service listens on IPv6: an empty value publishes IPv4 only (`127.0.0.1`), which is what a service bound to `0.0.0.0` needs, while `tcp` publishes both `127.0.0.1` and `::1` — and a client arriving over `::1` is accepted and then reset if nothing in the sandbox is listening there. Users can pin host ports with `sbx ports --publish <host>:<container>`.

Environment

Field	Description
`environment.variables`	Key-value pairs set directly in the container.

Do not set `DASH_`, `SBX_`, or `DOCKER_` variables, and avoid overriding `HOME`, `USER`, `SHELL`, `PATH`, `LD_PRELOAD`, and `LD_LIBRARY_PATH`. The runtime reserves these names and may override them.

Setup

`setup.install`, `setup.startup`, and `setup.files` are lists of commands or files with the fields described here.

Execution order

When a sandbox is created, kit content is applied in this order:

1. Network permissions and environment variables.
2. Static files under `files/home/`.
3. `setup.install` commands, in declaration order.
4. `setup.files` entries.
5. `setup.startup` commands are registered for each sandbox start.
6. Static files under `files/workspace/`, after the workspace is ready. With `--clone`, this means after the repository has been cloned.

For stacked kits, entries in each stage are applied in `--kit` order. An install command can consume a bundled file from `files/home/`, but not one from `files/workspace/` or `setup.files`, because those files land later.

`sbx kit add` recreates the sandbox rather than modifying it in place. It supports mixin kits limited to `environment.variables`, `setup.install`, `setup.startup`, and `permissions.network.allow`, which follow the same order as sandbox creation. Startup commands run after the kit is added and on each subsequent sandbox start. The command rejects kits that declare static files or `setup.files`. To use those fields, create the sandbox with the kit included.

install

Runs synchronously when a kit is applied, either during sandbox creation or through `sbx kit add`. Shell strings are passed to `sh -c`.

Kit install commands start in the template image's configured `WORKDIR`. Docker-provided templates use `/home/agent/workspace`, which isn't necessarily the primary workspace in a direct-mounted or clone-mode sandbox. Don't rely on the current directory to locate workspace files. Use absolute paths for bundled assets from `files/home/`.

Field	Default	Description
`command`	—	Shell command string.
`user`	`"0"`	User to run as. `"0"` = root.
`description`	—	Human-readable description.

startup

Runs at every sandbox start. String array, not interpreted by a shell. A `$WORKSPACE_DIR` reference in the array is passed unchanged. To expand it, run the command through a shell, for example `["sh", "-c", "..."]`.

Field	Default	Description
`command`	—	Command and args as a string array.

Field	Default	Description
`user`	`"1000"`	User to run as. `"1000"` = agent.
`background`	`false`	Block later startup commands until this command finishes. Set to `true` to let later commands run without waiting.
`description`	—	Human-readable description.

Startup commands are non-interactive. They run before the agent attaches, with no terminal connected, so they can't prompt the user (for example, an interactive `aws login` will hang or fail). They also don't gate the agent's entrypoint: the agent launches once startup commands have been dispatched, regardless of `background`. A value of `false` waits within the startup dispatcher before it runs the next command; it doesn't delay the agent entrypoint. Use startup commands for work that can run alongside the agent. Use `setup.files` for any value that needs to land on disk before the agent runs.

Startup commands must be idempotent. They run on every sandbox start and replay on container restarts, so a command that fails or misbehaves on a second invocation breaks the restart path. Guard work with existence checks, use upserts instead of inserts, and prefer commands that converge to the same end state regardless of how many times they run.

files

Files written at sandbox start, with runtime substitution.

Field	Default	Description
`path`	—	Absolute container path.
`content`	—	File content. Write `${WORKDIR}` for the workspace path.
`mode`	`"0644"`	File permissions in octal.
`onlyIfMissing`	`false`	Skip if the file already exists.

The runtime writes these files as the agent user with UID 1000. The target path must be writable by that user. To write to a root-owned path such as `/etc`, use an `install` command, which runs as root by default. Set ownership in the install command if the agent needs to modify the file later.

Write `${WORKDIR}` in `content` and `$WORKSPACE_DIR` in commands. Three names look alike and mean different things:

Name	What it is	Where it applies
`${WORKDIR}`	Placeholder for the workspace path	`setup.files[].content` only

Name	What it is	Where it applies
`WORKSPACE_DIR`	Environment variable set by the runtime to the workspace path	Shell commands, and the agent
`WORKDIR`	The template image's Dockerfile working directory	Where commands start, see install

The runtime replaces `${WORKDIR}` when it creates the sandbox, only inside `content`, never in `path` or in commands. `WORKSPACE_DIR` is an environment variable, so a `$WORKSPACE_DIR` reference expands only when a shell runs the command. `install` strings run through `sh -c`. `startup` arrays run without a shell. A Dockerfile `WORKDIR` doesn't define an environment variable.

Shell initialization and service logs

With Docker templates, append shell initialization to `/etc/sandbox-persistent.sh` in an install command. Keep existing content and omit completion scripts: interactive and non-interactive Bash commands source this file. For a background service, redirect startup output to a file and read it with `sbx exec`. Use `background: true` instead of a trailing `&`.

Static files

```
my-kit/files/
├── home/       → /home/agent/
└── workspace/  → primary workspace path
```

Kit path	Container destination
`files/home/`	`/home/agent/` (config files, dotfiles)
`files/workspace/`	The primary workspace path

Parent directories are created automatically. Existing files are overwritten. Absolute paths and path-traversal sequences (`../../`) are rejected.

Static files can supply linter settings, helper scripts, or agent skills. For example, a Claude Code project skill belongs at `files/workspace/.claude/skills/<NAME>/SKILL.md`.

Volumes

Declare `volumes` as a list of mounts with these fields:

Field	Description
`path`	Required absolute container path.
`type`	Empty for a block-backed volume, or `tmpfs` for RAM-backed storage.
`size`	Optional byte-size string.
`mode`	Optional octal permissions.

Volumes are applied only when a sandbox is created. `sbx kit add` cannot attach volumes to a running container.

Fork an existing agent

Sandbox kits (`kind: sandbox`) define a full agent from scratch. The most common variant is a fork of a built-in agent. Use `extends:` to inherit the parent's complete configuration and declare only the fields you want to change. This example replaces the built-in `claude` entrypoint so Claude Code uses manual permission mode instead of bypassing approval prompts:

```
schemaVersion: "2"
kind: sandbox
name: claude-safe
displayName: Claude Code (with approval prompts)
description: Claude Code in manual permission mode

extends: claude

sandbox:
  entrypoint: [claude, "--permission-mode", "manual"]
```

The child inherits the built-in image, credentials, network permissions, persistent volumes, settings, MCP integration, agent instructions, setup entries, and environment variables. Its `sandbox.entrypoint` replaces the inherited entrypoint.

Launch by passing the sandbox kit in place of a built-in agent name:

```
$ sbx run ./claude-safe
```

Install an internal CA certificate

Put each PEM-encoded root certificate under `files/home/` with a `.crt` extension. For `files/home/internal-ca.crt`, use:

```
schemaVersion: "2"
kind: mixin
name: internal-ca
setup:
  install:
    - command: "install -m 0644 /home/agent/internal-ca.crt /usr/local/share/ca-
certificates/internal-ca.crt && update-ca-certificates"
      user: "0"
```

This updates the system trust store. For several CAs, install every certificate before running `update-ca-certificates`.

Sandbox-managed agent configuration

Built-in agent kits reserve the following paths for sandbox setup. Treat these paths as sandbox-managed, even if a file is only needed for a particular feature. Don't target them with static files, `setup.files`, or install commands. Later setup can replace your content or depend on settings that your file removes. In this table, `~` is `/home/agent`.

Built-in agent kit	Managed configuration paths
`claude`	`~/.claude.json`, `~/.claude/settings.json`, `~/.claude/.config.json`
`codex`	`~/.codex/config.toml`
`copilot`	`~/.copilot/config.json`
`cursor`	`~/.cursor/cli-config.json`
`devin`	`~/.config/devin/config.json`, `~/.config/devin/mcp_config.json`
`gemini`	`~/.gemini/settings.json`
`kiro`	`~/.kiro/settings/mcp.json`
`opencode`	`~/.config/opencode/opencode.json`

Use separate settings files when supported: Claude Code accepts `--settings`, and OpenCode reads `OPENCODE_CONFIG`. Don't use `setup.startup` for settings the agent must read during initialization; startup commands don't gate the entrypoint.

Validate and inspect kits

Use these commands to check a kit before sharing it:

- `sbx kit validate <path>` — check that a kit directory or ZIP is well-formed.
- `sbx kit inspect <path>` — display kit details. Add `--json` for machine-readable output.

Add `--offline` to validate or inspect a local directory or ZIP without starting the daemon or signing in. Offline checks skip kit source and signature policies; they don't establish whether the kit is permitted to load.

Packaging and distribution

- `sbx kit pack <path> -o <file.zip>` — package a directory as a ZIP file for sharing.
- `sbx kit push <path> <ref>` — publish to an OCI registry (for example, `ghcr.io/myorg/my-kit:1.0`).
- `sbx kit pull <ref>` — download a kit to the working directory as `.zip` for schema v1 or `.tar.gz` for schema v2. Use `--output <FILE>` to choose the output path.

For Docker Hub, `sbx kit pull` and `sbx kit push` use the session from `sbx login`. For other registries, they prefer credentials stored with `sbx secret set --registry`. Both commands fall back to the Docker credential store, so credentials from `docker login` also work.

Sign and verify kits

Use cosign-compatible Sigstore signatures to verify who approved a kit and that its signed content hasn't changed. Signing is keyless by default. Verify a keyless signature with the certificate identity and OpenID Connect (OIDC) issuer:

```
$ sbx kit sign ./my-kit/
$ sbx kit verify \
    --certificate-identity user@example.com \
    --certificate-oidc-issuer https://accounts.google.com \
    ./my-kit/
```

For key-based signing, use an ECDSA P-256 key pair:

```
$ sbx kit sign --key cosign.key ./my-kit/
$ sbx kit verify --key cosign.pub ./my-kit/
```

For a local directory, `sbx kit sign` writes a `kit.sig.bundle` file next to `spec.yaml`. Commit this file so consumers can verify a kit loaded from the Git repository.

Git line-ending conversion and clean filters can invalidate a source signature. If `sbx kit sign` warns about these transformations, ensure Git preserves the signed files unchanged before sharing the Git reference.

For an OCI kit, the signature is stored as an OCI referrer. You can sign an OCI kit after pushing it, or push and sign it in one step:

```
$ sbx kit push ./my-kit/ ghcr.io/myorg/my-kit:1.0 --sign
```

ZIP kits can't carry verifiable signatures.

Inspect provenance

Pushing a kit also attaches a provenance attestation that records its content digests, declared sandbox image, and source Git commit when pushed from a working tree. Inspect it to check the source of a published kit:

```
$ sbx kit provenance ghcr.io/myorg/my-kit:1.0
```

Provenance is unsigned by default. Use `sbx kit push --sign` to sign both the kit manifest and its provenance when publishing.

Require signed kits

Set `kit.trustedSigners` to the identities or keys you trust before requiring signatures. Otherwise, `sbx` uses the default policy, which trusts Docker employee identities attested by Google's OpenID Connect issuer. A keyless policy must specify both the certificate identity and its OpenID Connect issuer:

```
$ sbx settings set kit.trustedSigners \
    '[{"identity":"release-bot@example.com","issuer":"https://accounts.google.com"}]'
$ sbx settings set kit.requireSignature true
```

To trust a key-based signature, set the policy to the public key path:

```
$ sbx settings set kit.trustedSigners '[{"key":"/path/to/cosign.pub"}]'
$ sbx settings set kit.requireSignature true
```

When `kit.requireSignature` is `true`, `sbx` rejects unsigned kits, signatures that don't match `kit.trustedSigners`, and ZIP kits. This policy applies when a kit is loaded from a local directory, Git repository, or OCI registry.

The signature covers `spec.yaml` and the kit's `files/` content, but not mutable dependencies such as image tags or content downloaded by install and startup commands. Pin those dependencies by digest or checksum when they must remain immutable.

Schema versions

Schema v2 is supported starting with Docker Sandboxes version 0.36. Use `schemaVersion: "2"` for the syntax on this page. Version `"1"` also remains accepted. V3 is a separate format for environments built entirely with v3 workloads and mixins. V3 kits can't compose with v1 or v2 kits. See Kits v3 for that workflow.

When migrating to `schemaVersion: "2"`, replace v1 fields with their v2 equivalents:

v1	v2
`credentials.sources.<id>`	`credentials:` list entry with `service`
`network.allowedDomains` / `deniedDomains`	`permissions.network.allow` / `deny`
`network.serviceDomains` / `serviceAuth`	`credentials[].apiKey.inject`
`network.publishedPorts` / `publishedPorts`	top-level `ports`
standalone `oauth:` block	`credentials[].oauth`
`oauth.skipIfEnv`	Accepted but ignored
`environment.proxyManaged`	`credentials[].apiKey.proxyManaged`
`memory` / `agentContext`	`agentInstructions.content`
`kind: agent` / `agent:` block	`kind: sandbox` / `sandbox:` block
`sandbox.aiFilename`	`agentInstructions.filename`
`sandbox.entrypoint.run`	`sandbox.entrypoint`
`sandbox.entrypoint.args`	`sandbox.command.default`
`sandbox.entrypoint.ttyArgs`	`sandbox.command.interactive`
`tmpfs:`	`volumes:` entries with `type: tmpfs`
`volumes:` (mapping form)	`volumes:` sequence (`- path: <path>`)
`commands:` / `commands.initFiles`	`setup:` / `setup.files`
`settings:` / `kitDir` / `persistence`	Removed

Credential discovery also moved out of the kit in v2: a kit declares which credentials it needs and how to inject them, but where each value comes from is controlled by the user through credential bindings.

> [!NOTE] `mixins` and `sandbox.build` are accepted by the parser, but runtime support is pending. A kit that sets `sandbox.build` must also set `sandbox.image`.

Move an environment to v3

Select a v3 workload, convert or replace its mixins, and create a separate sandbox. Every selected kit must use v3. Running an existing sandbox keeps its recorded kit configuration.

For an environment file, see Add v3 kits. Keep its `schemaVersion: "1"`: the environment file version is independent of kit descriptor versions.

Changing `schemaVersion` alone doesn't convert a kit. Separate reusable image content from sandbox initialization, and declare runtime capabilities:

V2 surface	V3 equivalent
`kind: sandbox`	`kind: workload` with a Dockerfile recipe
`sandbox.image`	Dockerfile `FROM`
`sandbox.entrypoint`, `sandbox.command`, `environment.variables`	Dockerfile `ENTRYPOINT`, `CMD`, and `ENV`
`extends`	A mixin for composition, or a derived workload image with its own descriptor
`setup.install`	Dockerfile `RUN` for reusable content; lifecycle `install` for sandbox initialization
`setup.startup` and `setup.files`	Lifecycle capability `startup` and `files`
`setup.files[].onlyIfMissing: true`	Lifecycle `files[].overwrite: false`
Automatic `files/home/` and `files/workspace/` injection	Dockerfile `COPY`, with lifecycle hooks for destinations provided by runtime mounts
`permissions.network` and `credentials`	Network-policy and credential capabilities
`agentInstructions`	Agent-context capability
`${WORKDIR}` in `setup.files[].content`	`${{ kit.env.WORKSPACE_DIR }}` in lifecycle `files` content

Follow the v3 authoring guidance when converting runtime setup and capability declarations. Use a kit set to combine published v3 components with your settings. To rebuild the agent environment

from a base image, follow Build an agent workload.

□□□ #2
□ □□□ □□□ 2026-10-31 19:00:20 UTC
□ □□□ □□□ 2026-10-31 19:00:25 UTC